Job ID: printopdf-22

Title: Symbolics Common Lisp Dictionary

Requesting User: genera

Billing Info: {job-billing}
Symbolics Common Lisp Dictionary

≠ number &rest numbers

In your new programs, we recommend that you use function =, which is the Common Lisp equivalent of ≠.

Returns t if number is not numerically equal to any of numbers, and nil otherwise. Either argument can be of any numeric type.

≤ number &rest more-numbers

In your new programs, we recommend that you use function <=, which is the Common Lisp equivalent of ≤.

≤ compares its arguments from left to right. If any argument is greater than the next, ≤ returns nil. But if the arguments are monotonically increasing or equal, the result is t.

Arguments must be noncomplex numbers, but they need not be of the same type. Examples:

(≤ 5) => T
(≤ 1 2 3) => T
(≤ 3 6 2 8) => NIL
(≤ 5 6.3) => T

≥ number &rest more-numbers

In your new programs, we recommend that you use function >= which is the Common Lisp equivalent of ≥.

≥ compares its arguments from left to right. If any argument is less than the next, ≥ returns nil. But if the arguments are monotonically decreasing or equal, the result is t.

Arguments must be noncomplex numbers, but they need not be of the same type. Examples:

(≥ 8) => T
(≥ 3 2 2 1) => T
(≥ 5 4 6 2) => NIL
(≥ 6.02s23 6.02d23) => T

+ &rest numbers

Returns the sum of its arguments. If there are no arguments, it returns 0, which is the identity for this operation. An error signals if any argument is a non-number.
If the arguments are of different numeric types, they are converted to a common
type, which is also the type of the result. See the section "Coercion Rules for
Numbers".

Examples:

\[(+ \Rightarrow 0)\]
\[(+ -8) \Rightarrow -8\]
\[(+ 1 2 3 4) \Rightarrow 10\]
\[(+ 2 5.9) \Rightarrow 7.9\]
\[(+ 5/2 2 2/3) \Rightarrow 31/6\]

When using Genera, the following functions are synonyms of \(+\):

\[\textbf{zl:plus}\]
\[\textbf{zl:}+\]

For a table of related items, see the section "Arithmetic Functions".

\[\textbf{+} \quad \text{Variable}\]

While a form is being evaluated by a read-eval-print loop, \(+\) is bound to the previ-
ous form that was read by the loop. Variable \(++\) is likewise bound to the penulti-
mate evaluated form, and \(+++\) to the form whose evaluation is removed from the
form currently undergoing evaluation.

\[(\text{floor } 5 2) \Rightarrow 2 1\]
\[(\text{eval } +) \Rightarrow 2 1\]

\[\textbf{++} \quad \text{Variable}\]

Holds the previous value of \(+\), that is, the form evaluated two interactions ago.

\[\textbf{+++} \quad \text{Variable}\]

Holds the previous value of \(++\), that is, the form evaluated three interactions ago.

\[\textbf{zl:+}\$ \text{\&rest args} \quad \text{Function}\]

Returns the sum of its arguments. If there are no arguments, it returns \(0\), which
is the identity for this operation.

The following functions are synonyms of \(\text{zl:+}\$:\

\[\textbf{zl:plus}\]
\[\text{+}\]

\[\textbf{- number \&rest more-numbers} \quad \text{Function}\]
With only one argument, returns the negative of its argument. With more than one argument, \texttt{-} returns its first argument minus all of the rest of its arguments. In this way, \texttt{-} serves the dual function of a unary minus and polyadic minus. However, this can cause confusion, particularly when used with \texttt{apply} or given an unexpected number of arguments.

If the arguments are of different numeric types they are converted to a common type, which is also the type of the result. See the section "Coercion Rules for Numbers".

Examples:

\begin{verbatim}
(- 8) => -8
(- 9 3) => 6
(- 9 4 2 1) => 2
(- #C(3 4) 4) => #C(-1 4)
(- 9 5/6) => 49/6
(- 1 2 3 4) => -8
\end{verbatim}

When using Genera, the following function is a synonym of \texttt{-}:

\begin{verbatim}
zl:-$
\end{verbatim}

For a table of related items, see the section "Arithmetic Functions".

\texttt{Variable}

\texttt{-}

While a form is being evaluated by a read-eval-print loop, \texttt{-} is bound to the form itself.

\begin{verbatim}
(print -) prints: (print -)
\end{verbatim}

\texttt{Function}

\texttt{zl:-$ arg &rest args}

With only one argument, returns the negative of its argument. With more than one argument, \texttt{zl:-$} returns its first argument minus all the rest of its arguments.

The following function is a synonym of \texttt{zl:-$}:

\begin{verbatim}
-
\end{verbatim}

\texttt{Function}

\texttt{zl:/ number &rest more-numbers}

In your new programs, we recommend that you use the function \texttt{/}, which is the Common Lisp equivalent of the function \texttt{/}.

With more than one argument, \texttt{/} is the same as \texttt{zl:quotient}; it returns the first argument divided by all of the rest of its arguments. With only one argument, \texttt{(/ x)} is the same as \texttt{(/ 1 x)}.

With integer arguments, \texttt{/} acts like \texttt{truncate}, except that it returns only a single value, the quotient.
Note that in Zetalisp syntax / is the quoting character and must therefore be doubled.

Examples:

```
(zl:/ 3 2) => 1  ;Integer division truncates.
(zl:/ 3 -2) => -1
(zl:/ -3 2) => -1
(zl:/ -3 -2) => 1
(zl:/ 3 2.0) => 1.5
(zl:/ 3 2.0d0) => 1.5d0
(zl:/ 4 2) => 2
(zl:/ 12. 2. 3.) => 2
(zl:/ 4.0) => .25
```

The following function is a synonym of /:

```
zl:/$
```

For a table of related items, see the section "Arithmetic Functions".

/ number &rest more-numbers  \( Function \)

With more than one argument, / successively divides the first argument by all the others and returns the result. With one argument, / returns the reciprocal of the argument: (/ x) is the same as (/ 1 x). If the arguments are of different numeric types, they are converted to a common type, which is also the type of the result. See the section "Coercion Rules for Numbers".

/ follows normal mathematical rules, so if the mathematical quotient of two integers is not an exact integer, the function returns a ratio. To obtain an integer result, use one of these functions: floor, ceiling, truncate, round

```
(/ 4) => 1/4
(/ 4.0) => 0.25
(/ 9 3) => 3
(/ 18 4) => 9/2  ;returns rational number in canonical form
(/ 101 10.0) => 10.1  ;applies coercion rules
(/ 101 10) => 101/10
(/ 24 4 2) => 3
(/ 36. 4. 3.) => 3
(/ 36.0 4.0 3.0) => 3.0
(/ #c(1 1) #c(1 -1)) => #c(0 1)
(/ #c(3 4) 5) => #c(3/5 4/5)
```

For a table of related items, see the section "Arithmetic Functions".

zl:$ arg &rest args  \( Function \)

With more than one argument, zl-user:$ is the same as zl:quotient; it returns the first argument divided by all of the rest of its arguments. With only one argument, (zl-user:$ x) is the same as (zl-user:$ 1 x).
With integer arguments, `zl-user:$/` acts like `truncate`, except that it returns only a single value, the quotient.

Note that in Zetalisp syntax `zl:/$` is the quoting character and must therefore be doubled.

The following function is a synonym of `zl-user:$/`:

```
zl:/
```

`/= number &rest numbers`  

_Returns_ `t` if all arguments are not equal, and `nil` otherwise. Arguments can be of any numeric type; the rules of coercion are applied for arguments of different numeric types.

Two complex numbers are considered _= _ if their real parts are _= _ and their imaginary parts are _= _.

_Examples:_

```
(/= 4) => T
(/= 4 4.0) => nil
(/= 4 #c(4.0 0)) => nil
(/= 4 5) => T
(/= 4 5 6 7) => T
(/= 4 5 6 7 4) => nil
(/= 4 5 4 7 4) => nil
(/= #c(3 2) #c(2 3) #c(2 -3)) => T
(/= #c(3 2) #c(2 3) #c(2 -3) #c(2 3.0)) => nil
```

When using Genera, the following function is a synonym of `/=` :

```
≠
```

For a table of related items, see the section "Numeric Comparison Functions".

`/`  

_While a form is being evaluated by a read-eval-print loop, `/` is bound to a list of the results printed the last time through the loop._

If you are using CLOE, variable `/` is bound to the list of values returned by the last evaluated form. Variable `//` is bound to the list of values returned by the penultimate evaluated form, and variable `///` is bound to the list of values returned by the form evaluated three before the current form.

```
(floor 5 2) => 2, 1
/= => (2 1)
```

`//`  

_Variable_
Holds the previous value of `user::///////////`, that is, the list of results printed two times through the loop ago.

///

Holds the previous value of `user:///////////`, that is, the list of results printed three times through the loop ago.

< number &rest more-numbers  
\textit{Function}

Compares its arguments from left to right. If any argument is not less than the next, < returns \texttt{nil}. But if the arguments are monotonically strictly increasing, the result is \texttt{t}.

Arguments must be noncomplex numbers, but they need not be of the same type. An error is returned if any of the arguments are complex or not numbers.

\textbf{Examples:}

\begin{verbatim}
(< 3 4) => T
(< 1 1.0) => T
(< 0 1/2 2.0 3 4) => T
(< 0 1 3 2 4) => T
(< 5/2 5) => T
(< 3 3.12) => T
\end{verbatim}

When using Genera, the following function is a synonym of <:

\texttt{zl:lessp}

For a table of related items, see the section "Numeric Comparison Functions".

<= number &rest more-numbers  
\textit{Function}

Compares its arguments from left to right. If any argument is greater than the next, <= returns \texttt{nil}. But if the arguments are monotonically increasing or equal, the result is \texttt{t}.

Arguments must be noncomplex numbers, but they need not be of the same type. An error is returned if any of the arguments are complex or not numbers.

\textbf{Examples:}

\begin{verbatim}
(<= 8) => T
(<= 3 4) => T
(<= 1 1) => T
(<= 1 1.0) => T
(<= 0 1/2 2.0 3 4) => T
(<= 0 1 3 2 4) => T
(<= 0 1 3 3 4) => T
(<= 5 5/2) => T
(<= 3 3.0 3.5 4) => T
\end{verbatim}
When using Genera, the following function is a synonym of \( \leq \):
\[
\leq
\]
For a table of related items, see the section "Numeric Comparison Functions".

\[= \text{number} \ \&\text{rest more-numbers}\]

\textit{Function}

Tests for numeric equality of numbers, and works for any type of number. Differs from \texttt{eq} in that non-identical but numerically equal numbers will not be \texttt{eq} but will be \texttt{=}.
Differs from \texttt{eql} in that numerically equal numbers need not be of the same type to be \texttt{=}.
Returns \texttt{t} if all arguments are numerically equal.

\[=\] takes arguments of any numeric type; the arguments can be of dissimilar numeric types.

\textbf{Examples:}
\begin{verbatim}
(= 8) => T
(= 3 4) => NIL
(= 3.0 3.0d0) => T
(= 4 #c(4.0 0.0) #c(4.0d0 0.0d0)) => T
(= 0 0.0) => t
(= #c(1 2) #c(1.0 2.0)) => t
\end{verbatim}

For a discussion of non-numeric equality predicates, see the section "Comparison-performing Predicates".
For a table of related items, see the section "Numeric Comparison Functions".

\[> \text{number} \ \&\text{rest more-numbers}\]

\textit{Function}

Compares its arguments from left to right. If any argument is not greater than the next, \texttt{>} returns \texttt{NIL}.
But if the arguments are monotonically strictly decreasing, the result is \texttt{t}.

Arguments must be noncomplex numbers, but they need not be of the same type.
An error is returned if any of the arguments are complex or not numbers.

\textbf{Examples:}
\begin{verbatim}
(> 4 3.0) => T
(> 4 3 2 1/2 0) => T
(> 4 3 1 2 0) => NIL
(> 4 3) => t
(> 3 3 2) => nil
\end{verbatim}

When using Genera, the following function is a synonym of \texttt{>}: 
\texttt{zl:greaterp}

For a table of related items, see the section "Numeric Comparison Functions".

\[\geq \text{number} \ \&\text{rest more-numbers}\]

\textit{Function}
Compares its arguments from left to right. If any argument is less than the next, \( >= \) returns \texttt{nil}. But if the arguments are monotonically decreasing or equal, the result is \texttt{t}.

Arguments must be noncomplex numbers, but they need not be of the same type. An error is returned if any of the arguments are complex or not numbers.

Examples:

\[
\begin{align*}
(\geq 8) & \Rightarrow T \\
(\geq 4.3.0) & \Rightarrow T \\
(\geq 4.2.3.1.0) & \Rightarrow T \\
(\geq 4.2.3.1.0) & \Rightarrow NIL \\
(\geq 4.3.2.1.2/0) & \Rightarrow T \\
(\geq 4.3) & \Rightarrow t \\
(\geq 3.3.2) & \Rightarrow t
\end{align*}
\]

When using Genera, the following function is a synonym of \( \geq \):

\[
\ge
\]

For a table of related items, see the section "Numeric Comparison Functions".

\begin{verbatim}
zl:\ x y  Function
In your new programs, we recommend that you use either the function \texttt{rem} or \texttt{remainder} which are the Common Lisp equivalents of the function \texttt{zl-}.
\end{verbatim}

\texttt{zl-}\ user:

\begin{verbatim}
\texttt{zl-} \texttt{user:}
\end{verbatim}

Returns the remainder of \( x \) divided by \( y \). \( x \) and \( y \) must be integers.

\begin{verbatim}
\texttt{zl-} \texttt{user:}
\end{verbatim}

acts like \texttt{truncate}, except that it returns only a single value, the remainder.
Examples:

(zl:\ \ 3 2) => 1
(zl:\ \ -3 2) => -1
(zl:\ \ 3 -2) => 1
(zl:\ \ -3 -2) => -1

The following functions are synonyms for **zl-**

Note: In programs using the Zetalisp syntax you would represent **zl-**

as \\

| The function is represented here as **zl-** |

because all objects in this manual are represented as if printed by **prin1** with

**package** bound to the Common Lisp readable. In Common Lisp, the backslash
character (\) is the escape character and must be doubled.

zli: x y &rest args

**Function**
Returns the remainder of \( x \) divided by \( y \). The arguments must be integers.

The following functions are synonyms of \( \% \):

\[
\text{zl:remainder} \\
\text{rem}
\]

We recommend that you use \texttt{rem} in new programs.

\textbf{Note:} In programs using the Zetalisp syntax you would represent \% as \( \backslash \). The function is represented here as \( \backslash \) only because all objects in this manual are represented as if printed by \texttt{prin1} with \texttt{*package*} bound to the Common Lisp readable. In Common Lisp, the backslash character (\( \backslash \)) is the escape character and must be doubled.

\[
\text{zl:}^\times x y
\]

Returns \( x \) raised to the \( y \)th power. The result is an integer if both arguments are integers (even if \( y \) is negative!) and floating-point if either \( x \) or \( y \) or both is floating-point. If the exponent is an integer a repeated-squaring algorithm is used, while if the exponent is floating the result is \((\text{exp} (* y (\log x)))\).

The following functions are synonyms of \texttt{zl:}\(^\times\):

\[
\text{zl:expt} \\
\text{zl:}^\$ 
\]

\[
\text{zl:}^\$ x y
\]

Returns \( x \) raised to the \( y \)th power. The result is an integer if both arguments are integers (even if \( y \) is negative!) and floating-point if either \( x \) or \( y \) or both is floating-point. If the exponent is an integer a repeated-squaring algorithm is used, while if the exponent is floating the result is \((\text{exp} (* y (\log x)))\).

The following functions are synonyms of \texttt{zl:}\(^\$\):

\[
\text{zl:expt} \\
\text{zl:}^\times
\]

\texttt{* \&rest numbers}

Returns the product of its arguments. If there are no arguments, it returns 1, which is the identity for this operation.

If the arguments are of different numeric types they are converted to a common type, which is also the type of the result. See the section “Coercion Rules for Numbers”.

Examples:
When using Genera, the following functions are synonyms of *:

- `zl:times`
- `zl:*$

For a table of related items, see the section "Arithmetic Functions".

* Variable

While a form is being evaluated by a read-eval-print loop, * is bound to the result printed the last time through the loop. If several values were printed (because of a multiple-value return), * is bound to the first value. If no result was printed, * is not changed. Variable ** is bound to the value returned by the penultimate evaluated form, and *** is bound to the value returned by the form evaluated three before the current form. The star forms always return only a single value.

```
(floor 5 2) => 2, 1
* => 2
```

** Variable

Holds the previous value of *, that is, the result of the form evaluated two interactions ago.

*** Variable

Holds the previous value of **, that is, the result of the form evaluated three interactions ago.

`zl:*$ &rest args` Function

Returns the product of its arguments. If there are no arguments, it returns 1, which is the identity for this operation.

The following functions are synonyms of `zl:*$`:

- `zl:times`
- `*`
(1+ number) is the same as (+ number 1).
Examples:
(1+ 5) => 6  
(1+ 3.0d0) => 4.0d0  
(1+ 3/2) => 5/2  
(1+ #C(4 5)) => #C(5 5)
When using Genera, the following functions are synonyms of 1+:
  zl:add1
  zl:1+$
For a table of related items: See the section "Arithmetic Functions".

zl:1+$ x
Function

(zl:1+$ x) is the same as (+ x 1).
The following functions are synonyms of zl:1+$:
  zl:add1
  1+

1- number
Function

(1- number) is the same as (- number 1). Note that this name might be confusing:
(1- number) does not mean 1 - number; rather, it means number - 1.
Examples:
(1- 9) => 8  
(1- 4.0) => 3.0  
(1- 4.0d0) => 3.0d0  
(1- #C(4 5)) => #C(3 5)
When using Genera, the following functions are synonyms of 1-:
  zl:sub1
  zl:1-$
For a table of related items: See the section "Arithmetic Functions".

zl:1-$ x
Function

(zl:1-$ x) is the same as (- x 1).
The following functions are synonyms of zl:1-$:
  zl:sub1
  1-

sys:%1d-aloc array index
Function
Returns a locative pointer to the array element-cell selected by the index. sys:%1d-aloc is like zl:aloc, except that it ignores the number of dimensions of the array and acts as if it were a one-dimensional array by linearizing the multidimensional elements.

Current style suggests that you should use (locf (sys:%1d-aref | ...|)) instead of sys:%1d-aloc.

When using sys:%1d-aloc it is necessary to understand how arrays are stored in memory: See the section "Row-major Storage of Arrays".

For an example of accessing elements of a multidimensional array as if it were a one-dimensional array: See the function sys:%1d-aref.

For a table of related items: See the section "Accessing Multidimensional Arrays as One-dimensional".

sys:%1d-aref array index

Returns the element of array selected by the index. sys:%1d-aref is the same as aref, except that it ignores the number of dimensions of the array and acts as if it were a one-dimensional array by linearizing the multidimensional elements. copy-array-portion uses this function.

For example:

```
(setq *array* (make-array '((20 30 50))) => #<Art-Q-20-30-50 5023116>
(setf (aref *array* 5 6 7) 'foo) => FOO

;;; The following three forms have the same effect.
(aref *array* 5 6 7) => FOO
(sys:%1d-aref *array* (+ (* (+ (* 5 30) 6) 50) 7)) => FOO
(sys:%1d-aref *array* (array-row-major-index *array*)) => FOO
(sys:%1d-aref *array* (array-row-major-index *array* 5 6 7)) => FOO
```

When using sys:%1d-aref it is necessary to understand how arrays are stored in memory: See the section "Row-major Storage of Arrays".

For a table of related items: See the section "Accessing Multidimensional Arrays as One-dimensional".

sys:%1d-aset value array index

Stores a value into the specified array element, selected by the index. sys:%1d-aset is the same as zl:aset, except that it ignores the number of dimensions of the array and acts as if it were a one-dimensional array.

copy-array-portion uses this function.

Current style suggests that you should use (setf (sys:%1d-aref | ...|)) instead of sys:%1d-aset.
When using `sys:%1d-aset` it is necessary to understand how arrays are stored in memory: See the section "Row-major Storage of Arrays".

For an example of accessing elements of a multidimensional array as if it were a one-dimensional array: See the function `sys:%1d-aref`.

For a table of related items: See the section "Accessing Multidimensional Arrays as One-dimensional".

### 2d-array-blt

```
alu nrows ncolumns from-array from-row from-column to-array to-row to-column
```

**Function**

Copies a rectangular portion of `from-array` into a portion of `to-array`. `2d-array-blt` is similar to `bitblt` but takes (row,column) style arguments on two-dimensional arrays, while `bitblt` takes (x,y) arguments on rasters.

The number of columns in `from-array` times the number of bits per element must be a multiple of 32. The same is true for `to-array`.

This can be used on `sys:art-fixnum` or `sys:art-1b, sys:art-2b, ... sys:art-16b` arrays. It can also be used on `sys:art-q` arrays provided all the elements are fixnums.

For a table of related items: See the section "Copying an Array".

### sys:%32-bit-difference

```
fixnum1 fixnum2
```

**Function**

Returns the difference of `fixnum1` and `fixnum2` in 32-bit two's complement arithmetic. Both arguments must be fixnums. The result is a fixnum.

For a table of related items, see the section "Machine-Dependent Arithmetic Functions".

### sys:%32-bit-plus

```
fixnum1 fixnum2
```

**Function**

Returns the sum of `fixnum1` and `fixnum2` in 32-bit two's complement arithmetic. Both arguments must be fixnums. The result is a fixnum.

For a table of related items, see the section "Machine-Dependent Arithmetic Functions".

### abs

```
number
```

**Function**

Returns `|number|`, the absolute value of `number`. For noncomplex numbers, `abs` could have been defined by:

```lisp
(defun abs (number)
  (cond ((minusp number) (minus number))
        (t number)))
```

Note that if `number` is equal to negative zero in IEEE floating-point format the above algorithm returns -0.0.
For complex numbers, \texttt{abs} could have been defined by:

\begin{verbatim}
(defun abs (number)
  (sqrt (+ (- (realpart number) 2) (- (imagpart number) 2))))
\end{verbatim}

\begin{itemize}
  \item \texttt{(abs 81)} \Rightarrow 81
  \item \texttt{(abs -81.0)} \Rightarrow 81.0
  \item \texttt{(abs #c(3 4))} \Rightarrow 5.0
\end{itemize}

See the function \texttt{phase}.

For a table of related items, see the section "Arithmetic Functions".

\textbf{acons} \texttt{key datum alist} \quad \textit{Function}

Constructs a new association list by adding the pair \texttt{(key . datum)} onto the front of \texttt{alist}. \texttt{acons} returns a new association list which has the new \texttt{key} and \texttt{datum} pair added to it. See the section "Association Lists". This is equivalent to using the \texttt{cons} function on \texttt{key} and \texttt{datum}, and consing it onto the old list as follows:

\begin{verbatim}
(acons key datum alist) \equiv (cons (cons key datum) alist)
\end{verbatim}

Example:

\begin{verbatim}
(setq bird-alist '((wader . heron) (raptor . eagle))) =>
  ((WADER . HERON) (RAPTOR . EAGLE))
\end{verbatim}

\begin{verbatim}
(acons 'diver 'loon bird-alist) =>
  ((DIVER . LOON) (WADER . HERON) (RAPTOR . EAGLE))
\end{verbatim}

\texttt{bird-alist} =>

\begin{verbatim}
  ((WADER . HERON) (RAPTOR . EAGLE))
\end{verbatim}

In the following example, \texttt{acons} updates the association list of tenured professors and their classes.

\begin{verbatim}
(setq professors-with-tenure
  '(("smith" . (CS202 CS231))
  ("parks" . (CS221))("hunter" . (CS216 CS232)))))
\end{verbatim}

\begin{verbatim}
(setq professors-with-tenure
  (acons "Jones" (list CS101 CS242)
    professors-with-tenure))
\end{verbatim}

\begin{verbatim}
(acons-with-tenure
  '(("Jones" . (CS101 CS242))("smith" . (CS202 CS231))
  ("parks" . (CS221))("hunter" . (CS216 CS232)))))
\end{verbatim}

For a table of related items: See the section "Functions that Operate on Association Lists".
**acos number**  
*Function*

Computes and returns the arc cosine of the argument (that is, the angle whose cosine is equal to *number*). The result is in radians.

The argument can be any noncomplex or complex number. Note that if the absolute value of *number* is greater than one, the result is complex, even if the argument is not complex.

The arc cosine being a mathematically multiple-valued function, **acos** returns a principal value whose range is that strip of the complex plane containing numbers with real parts between 0 and π. The range excludes any number with a real part equal to zero and a negative imaginary part, as well as any number with a real part equal to π and a positive imaginary part.

Examples:

```
(acos 1) => 0.0
(acos 0) => 1.5707964 ; π/2 radians
(acos -1) => 3.1415927 ; π
(acos 2) => #C(0.0 1.3169578)
(acos -2) => #C(3.1415927 -1.316958)
(acos (/ (sqrt 2) 2)) => 0.785398
```

For a table of related items, see the section "Trigonometric and Related Functions".

**acosh number**  
*Function*

Computes and returns the hyperbolic arc cosine of the argument (that is, the angle whose *cosh* is equal to *number*). The result is in radians.

The argument can be any noncomplex or complex number, except -1. Note that if the value of *number* is less than one, the result is complex, even if the argument is not complex. The hyperbolic arc cosine being mathematically multiple-valued in the complex domain, **acosh** returns a principal value whose range is that half-strip of the complex plane containing numbers with a non-negative real part and an imaginary part between 0 and π (inclusive). A number with real part zero is in the range if its imaginary part is between zero (inclusive) and π (inclusive).

Example:

```
(acosh 1) => 0.0  ;(cosh 0) => 1.0
(acosh -2) => #c(1.316958 3.1415927)
```

For a table of related items, see the section "Hyperbolic Functions".

**clos:add-method**  
*Generic Function*

Adds *method* to *generic-function* and returns the modified *generic-function*. **clos:add-method** is the underlying mechanism of the **clos:defmethod** macro.

*generic-function*  
A generic function object.
method

A method object.

If the generic function already has a method with the same parameter specializers and qualifiers as method, then the existing method is replaced with method.

An error is signaled if:

• The lambda-list of the method is not congruent with the lambda-list of the generic function.

• The method object is already attached to a different generic function object.

\texttt{zl:add1} x

\texttt{(zl:add1 x)} is the same as \texttt{(+:x 1)}.

The following functions are synonyms of \texttt{zl:add1}:

\begin{verbatim}
 1+
zl:1+$
\end{verbatim}

\textit{adjoin} item list &key (:test #eql) :test-not (:key #identity) (:area sys:default-cons-area) :localize .replace

\textbf{Function}

Adds an element to a set, provided it is not already a member. If item is added, the new cons is returned. Otherwise, list is returned. The keywords are:

\textbf{:test} Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The \texttt{item} matches the specification only if the predicate returns \texttt{t}. If \textbf{:test} is not supplied, the default operation is \texttt{eql}.

\textbf{:test-not} Similar to \textbf{:test}, except that \texttt{item} matches the specification only if there is an element of the list for which the predicate returns \texttt{nil}.

\textbf{:key} If not \texttt{nil}, should be a function of one argument that will extract the part to be tested from the whole element. This function is applied to both \texttt{item} and members of list.

\textbf{:localize} Can be \texttt{nil}, \texttt{t}, or a positive integer when using Genera:

\begin{verbatim}
nil       Does not localize the top level of the list before returning the list.
t       Localizes the top level of list structure, by calling \texttt{sys:localize-list} or \texttt{sys:localize-tree} on the list before returning it.
\end{verbatim}
The `integer` function localizes integer levels of list structure, by calling `sys:localize-list` or `sys:localize-tree` on the list before returning it.

`:replace` destructively modifies the specified element (or elements) and replaces it with the value provided. The value can be `t` or `nil`. Not available in CLOE.

Note that, since `adjoin` adds an element only if it is not already a member, the sense of `:test` and `:test-not` have inverted effect: with `:test`, an item is added to the list only if there is no element of the list for which the predicate returns `t`. With `:test-not`, an item is added if there is no element for which the predicate returns `nil`.

When `:test` is `eql`, the default, then:

\[
\text{(adjoin item list)} = (\text{if (member item list) list (cons item list)})
\]

Here are some examples:

\[
\text{(setq bird-list '((loon . diver) (heron . wader)))} \Rightarrow
\text{((LOON . DIVER) (HERON . WADER))}
\]

\[
\text{(setq bird-list (adjoin '(eagle . raptor) bird-list :key #'car))} \Rightarrow
\text{((EAGLE . RAPTOR) (LOON . DIVER) (HERON . WADER))}
\]

\[
\text{(adjoin '(eagle . oops) bird-list :key #'car) } \Rightarrow
\text{((EAGLE . RAPTOR) (LOON . DIVER) (HERON . WADER))}
\]

\[
\text{(setq add-to-list '((j-jones "John Jones" "acct rep")))}
\]

\[
\text{(setq list (adjoin add-to-list list
:test #'string-equal :key #'cadr))}
\]

For a table of related items: See the section “Functions for Constructing Lists and Conses”.

**Compatibility Note:** The keywords `:area`, `:localize`, and `:replace` are Symbolics extension to Common Lisp, not available in CLOE.

---

The `adjust-array` function changes the dimensions of an array. It returns an array of the same type and rank as `array`, but with the `new-dimensions`. The number of `new-dimensions` must equal the rank of the array. All elements of `array` that are still in the bounds are carried over to the new array.

`:element-type` specifies that elements of the new array are required to be of a certain type. An error is signalled if `array` contains elements that are not of that type. `:element-type` thus provides an error check.
:initial-element allows you to specify an initial element for any elements of the new array that are not in the bounds of array.

The :initial-contents and :displaced-to options have the same effect as they do for make-array. If you use either of these options, none of the elements of array are carried over to the new array.

You can use the :fill-pointer option to reset the fill pointer of array. If array had no fill pointer an error is signalled.

If the size of the array is being increased, adjust-array might have to allocate a new array somewhere. In that case, it alters array so that references to it are made to the new array instead, by means of "invisible pointers" under Genera. See the function structure-forward. adjust-array returns this new array if it creates one, and otherwise it returns array. Be careful to be consistent about using the returned result of adjust-array, because you might end up holding two arrays that are not the same (that is, not eq), but that share the same contents.

Compatibility Note: :displaced-conformally is a Symbolics extension to Common Lisp, and not available in CLOE.

```lisp
(setq *print-array* t)
(setq array-1 (make-array '(2 3 2) :initial-element 'a :adjustable t))
=> #3A(((A A) (A A) (A A)) ((A A) (A A) (A A)))

(adjust-array array-1 '(3 2 2) :initial-element 'b)
=> #3A(((A A) (A A)) ((A A) (A A)) ((B B) (B B)))

(setq an-array (make-array 10 :element-type 'string-char :adjustable t
   :initial-element #\x))
=> "xxxxxxxxxx"

(adjust-array an-array 15 :initial-element #\y)
=> "xxxxxxxxxyyyyy"

(setq *print-array* t)
(setq an-array (make-array '(2 3) :adjustable t
   :initial-contents '(((1 2 3)(4 5 6)))))
#2A((1 2 3)(4 5 6))

(adjust-array an-array '(3 2) :initial-element #\y)
#2A((1 2)(4 5)(#\y #\y))
```

**zl:adjust-array-size array new-size**

Function

If array is a one-dimensional array, its size is changed to be new-size. If array has more than one dimension, its size is changed to new-size by changing only the first dimension.

If array is made smaller, the extra elements are lost. If array is made bigger, the new elements are initialized in the same fashion as make-array would initialize them: either to nil, 0 or (code-char 0), depending on the type of array.
Example:

```lisp
(setq a (make-array 5))
(setf (aref a 4) 'foo)
(aref a 4) => foo
(zl:adjust-array-size a 2)
(aref a 4) => an error occurs
```

See the function `adjust-array`.

The meaning of `zl:adjust-array-size` for conformal indirect arrays is undefined.

**adjustable-array-p array**

Returns `t` if `array` is adjustable, and `nil` if it is not. Lisp dialects supported by Genera make most arrays adjustable even if the `:adjustable` option to `make-array` is not specified; but to guarantee that an array can be adjusted after created, it is necessary to use the `:adjustable` option. Under CLOE, arrays are adjustable only if the `:adjustable` option is specified non-nil.

```lisp
(setq foo (make-array (4 5)))
(adjustable-array-p foo) => nil ;under CLOE
=> T   ;under Genera
(setq bar (make-array (4 5) :adjustable t))
(adjustable-array-p bar) => t  ;CLOE and Genera
```

For a table of related items: See the section "Getting Information About an Array".

**:advance-input-buffer &optional new-pointer**

If `new-pointer` is non-nil, it is the index in the buffer array of the next byte to be read. If `new-pointer` is `nil`, the entire buffer has been used up.

**sys:*all-flavor-names***

This is a list of the names of all the flavors that have ever been created by `defflavor`.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**&allow-other-keys**

In a lambda-list that accepts keyword arguments, specifies that keywords that are not specifically listed after `&key` are allowed. They and their corresponding values are ignored, as far as keywords arguments are concerned, but they do become part of the `&rest` argument, if there is one.

**zl:alloc array &rest subscripts**

Function
Returns a locative pointer to the element of array selected by the subscripts. The subscripts must be integers and their number must match the dimensionality of array. See the section "Cells and Locatives".

Current style suggests using locf with aref instead of zl:alloc. For example:

   (locf (aref array subscripts))

alpha-char-p char

Returns t if char is a letter of the alphabet.

   (alpha-char-p #\A) => T
   (alpha-char-p #\1) => NIL

For a table of related items, see the section "Character Predicates".

alphalessp x y

(alphalessp x y) is equivalent to (string-lessp x y). If the arguments are not strings, alphalessp compares numbers numerically, lists by element, and all other objects by printed representation. alphalessp is a Maclisp all-purpose alphabetic sorting function.

Examples:

   (alphalessp "apple" "orange") => T
   (alphalessp 'tom 'tim) => NIL
   (alphalessp "same" "same") => NIL
   (alphalessp 'symbol "string") => NIL
   (alphalessp '(a b c) '(a b d)) => T

alphanumericp char

Returns t if char is a letter of the alphabet or a base-10 digit.

   (alphanumericp #\7) => T
   (alphanumericp #\%) => NIL

For a table of related items, see the section "Character Predicates".

always keyword for loop

always expr

Causes the loop to return t if expr always evaluates non-null. If expr evaluates to nil, the loop immediately returns nil, without running the epilogue code (if any, as specified with the finally clause); otherwise, t is returned when the loop finishes, after the epilogue code has been run. If the loop terminates before expr is ever evaluated, the epilogue code is run and the loop returns t.
always expr is like (and expr1 expr2 ...), except that if no expr evaluates to nil, always returns t and and returns the value of the last expr. If the loop terminates before expr is ever evaluated, always is like (and).

If you want a similar test, except that you want the epilogue code to run if expr evaluates to nil, use while.

Examples:

(defun loop-always (my-list)
  (loop for x in my-list
    finally (print "what you going to do next ?")
    do
      (princ x) (princ " ")
    do
      and always (equal x 'a))) => LOOP-ALWAYS

(loop-always '(b c a d)) => B NIL

(loop-always '(a a)) => A A
"what you going to do next ?" T

See the section "Aggregated Boolean Tests for loop".

and &rest types

Type Specifier

Allows the definition of data types that are the intersection of other data types specified by types. As a type specifier, and can only be used in list form.

Examples:

(typep 89 '(and integer number)) => T
(subtypep 'bit-vector '(and vector array)) => T and T
(sys:type-arglist 'and) => (&REST TYPES) and T

See the section "Data Types and Type Specifiers".

For a discussion of the function and: See the section "Flow of Control".

and &rest forms

Special Form

Evaluates each form one at a time, from left to right. If any form evaluates to nil, and immediately returns nil without evaluating any other form. If every form evaluates to non-nil values, and returns the value of the last form.

and can be used in two different ways. You can use it as a logical and function, because it returns a true value only if all of its arguments are true. So you can use it as a predicate:

Examples:
(if (and 'this 'that) "reaches this point") => "reaches this point"
(if (and (equal 1 1) (equal nil '())) "equal") => "equal"
(if (and socrates-is-a-person all-people-are-mortal)
  (setq socrates-is-mortal t))

Because the order of evaluation is well-defined, you can do:

(if (and (boundp 'x)
         (eq x 'foo))
  (setq y 'bar)) => NIL

knowing that the \texttt{x} in the \texttt{eq} form is not evaluated if \texttt{x} is found to be unbound.

You can also use \texttt{and} as a simple conditional form:

Examples:

\begin{verbatim}
(\texttt{and}) => T

(\texttt{and t nil}) => NIL

(\texttt{and t 'hi (numberp 3.14)}) => T

(\texttt{when (and (setq temp (assq x y))}
  (\texttt{rplacd temp z)})

(\texttt{when (and bright-day}
  glorious-day
  (\texttt{princ "It is a bright and glorious day."})

In the following example, \texttt{very-expensive-function} is not evaluated because a prior form is false:

(setq foo 12 bar '(3 4 5))

(if (and (eql 12 foo)
  (eql foo bar)
  (very-expensive-function bar))

bar
foo)

Note: (\texttt{and}) => \texttt{t}, which is the identity for the \texttt{and} operation.

For a table of related items: See the section "Conditional Functions".

CLOE Note: This is a macro in CLOE.

\textbf{zl:ap-1 array index} Function

This is an obsolete version of \texttt{zl:alloc} that works only for one-dimensional arrays. There is no reason ever to use it.
**zl:ap-2 array index1 index2**

Function

This is an obsolete version of `zl:aloc` that works only for two-dimensional arrays. There is no reason ever to use it.

**zl:ap-leader array index**

Function

Returns a locative pointer to the indexed element of array’s leader. `array` should be an array with a leader, and `index` should be an integer. See the section "Cells and Locatives".

However, the preferred method is to use `locf` and `array-leader` as shown in the following example:

```lisp
(setq *array*
  (make-array '(2 3) :element-type 'character
              :leader-list '(t nil)))

(locf (array-leader *array* 1))
```

**append &rest lists**

Function

Concatenates `lists`, returning the resulting list. The arguments to `append` are lists. They are not changed (see `nconc`). Example:

```lisp
(append '(a b c) '(d e f) nil '(g)) => (a b c d e f g)
```

`append` makes copies of the top-level list structure of all the arguments it is given, except for the last one. So the new list shares the conses of the last argument to `append`, but all the other conses are newly created. Only the lists are copied, not the elements of the lists. The function `concatenate` can perform a similar operation, but always copies all its arguments. See also `nconc`, which is like `append` but destroys all its arguments except the last.

The last argument does not have to be a list, but can be any Lisp object, which becomes the tail of the constructed list. For example,

```lisp
(append '(a b c) 'd) => (a b c . d)
```

A version of `append` that only accepts two arguments could have been defined by:

```lisp
(defun append2 (x y)
  (cond ((atom x) y)
        ((cons (car x) (append2 (cdr x) y)))))
```

The generalization to any number of arguments could then be made (relying on `car` of `nil` being `nil`):

```lisp
(defun append (&rest args)
  (if (< (length args) 2) (car args)
      (append2 (car args)
                (apply (function append) (cdr args))))))
```
These definitions do not express the full functionality of append; the real definition under Genera minimizes storage utilization by cdr-coding the list it produces. See the section “Cdr-Coding.”

Example:
```scheme
(setq a '(1 2) b '(3 4) c '(5 6) d 7) => 7
(setq x (append a b c)) => (1 2 3 4 5 6)
(setf (car c) 'foo) (setf (car b) 'bar) x =>
(1 2 bar 4 foo 6)
(append a b c d) => (1 2 bar 4 foo 6 . 7)
a => (1 2)
```

To copy a list, use copy-list; the old practice of using `append x ()` to copy lists is unclear and obsolete.

For a table of related items: See the section “Functions for Constructing Lists and Conses”.

**append keyword for loop**

**append expr {into var}**

Causes the values of expr on each iteration to be appended together. When the epilogue of the loop is reached, var has been set to the accumulated result and can be used by the epilogue code.

It is safe to reference the values in var during the loop, but they should not be modified until the epilogue code for the loop is reached.

The forms append and appending are synonymous.

Examples:

```scheme
(defun splice-list (list1 list2)
  (loop for item1 in list1
        for item2 in list2
        append (list item1) into result
        append (list item2) into result
        finally (return (append result )))) => SPLICE-LIST
(splice-list '(Let not the of minds) '(me to marriage true)) =>
(LET ME NOT TO THE MARRIAGE OF TRUE)
```

Is equivalent to
(defun splice-list (list1 list2)
  (loop for item1 in list1
        for item2 in list2
        appending (list item1) into result
        appending (list item2) into result
     finally (return (append result ))))  => SPLICE-LIST
(splice-list '(Let not the of minds) '(me to marriage true)) =>
(LET ME NOT TO THE MARRIAGE OF TRUE)

Not only can there be multiple accumulations in a loop, but a single accumulation can come from multiple places within the same loop form, if the types of the collections are compatible. append, collect, and nconc are compatible.

See the section "Accumulating Return Values for loop".

apply function argument &rest arguments

Function

Applies the function function to arguments. function can be any function, but it cannot be a special form or a macro. The arguments for function consist of the last argument to apply appended to the end of the list of all other arguments to apply except for function itself. It is as if all the arguments to apply except function were given to list* to create the argument list.

Examples:

(setq fred '+)
(apply fred '(1 2)) => 3
(apply fred 1 2 '(3 4) => 10
(apply 'cons '((+ 2 3) 4)) => ((+ 2 3) . 4) not (5 . 4)

Note that if the function takes keyword arguments, you must put the keywords as well as the corresponding values in the argument list.

(apply '#(lambda (&key a b) (list a b)) '(:b 3) => (nil 3)

Compatibility Note: In Symbolics Common Lisp, apply is extended to allow you to call an array as a function.

See the section "Functions for Function Invocation".

zl:apply fn args

Function

Applies the function fn to the list of arguments args. args must be a list; fn can be any function, but it cannot be a special form or a macro. The arguments for fn consist of the elements of the list args.

Examples:
Of course, `args` can be `nil`. Note: Unlike Maclisp, `zl:apply` never takes a third argument; there are no "binding context pointers" in Symbolics Common Lisp.

See the function `funcall`.

See the section "Functions for Function Invocation".

**apropos** `string` &optional `package` `(do-inherited-symbols t) do-packages-used-by`  
**Function**

Searches for all symbols whose print-names contain `string` as a substring. When it finds a symbol, it prints out the symbol's name; if the symbol is defined as a function and/or bound to a value, it tells you so, and prints the names of the arguments (if any) to the function or the dynamic value of the symbol. If `package` is specified, it only searches for symbols containing `string` in that package, otherwise all packages are searched, as if by `do-all-symbols`. Because symbols can be available in more than one package by inheritance, `apropos` might print information about the same symbol more than once.

**Compatibility Note**: Symbolics Common Lisp provides two additional optional arguments, `do-inherited-symbols` and `do-packages-used-by`. If `do-inherited-symbols` is `t`, the set of packages searched includes all packages that `package` uses. If `do-packages-used-by` is `t`, the set also includes all packages that use `package`. You cannot use these two optional arguments in CLOE runtime.

`apropos` prints its information to `*standard-output*`. It returns `nil`.

**zla:apropos** `apropos-substring` &optional `pkg` `(do-packages-used-by t) do-packages-used`  
**Function**

Searches for all symbols whose print-names contain `apropos-substring` as a substring. When it finds a symbol, it prints out the symbol's name; if the symbol is defined as a function and/or bound to a value, it tells you so, and prints the names of the arguments (if any) to the function. It checks all symbols in a certain set of packages. The set always includes `pkg`. If `do-packages-used-by` is `t`, the set also includes all packages that use `pkg`. If `do-packages-used` is `t`, the set also includes all packages that `pkg` uses. `pkg` defaults to the `global` package, so normally all packages are searched. `apropos` returns a list of all the symbols it finds. This is similar to the Find Symbol command, except that Find Symbol only searches the current package unless you specify otherwise.

**apropos-list** `string` &optional `package` `do-packages-used-by`  
**Function**
Searches for all symbols whose print-names contain *string* as a substring. If the Symbolics Common Lisp optional argument *package* is specified, the function only searches for symbols containing *string* in that package, otherwise all packages are searched, as if by **do-all-symbols**. It returns a list of the symbols it finds.

**Compatibility Note:** Symbolics Common Lisp provides the additional optional argument *do-packages-used-by*. If *do-packages-used-by* is **t**, the set also includes all packages that use *package*. *Package* and *do-packages-used-by* may not work in other implementations of Common Lisp and does not work in CLOE Runtime.

For more information, see the function **apropos**.

**zl:ar-1** array index

This is an obsolete version of **aref** that works only for one-dimensional arrays. There is no reason ever to use it.

**zl:ar-2** array index1 index2

This is an obsolete version of **aref** that works only for two-dimensional arrays. There is no reason ever to use it.

**aref** array &rest subscripts

Returns the element of *array* selected by the *subscripts*. The *subscripts* must be integers and their number must match the dimensionality of *array*.

```
(setq this-array (make-array '(2 3) :initial-contents
               '((a b c) (d e f))))

(aref this-array 0 0) => A
(aref this-array 0 1) => B
(aref this-array 0 2) => C
(aref this-array 1 0) => D
```

**setf** can be used with **aref** to set the value of an array element.

```
(setf (aref this-array 1 0) 'x) => X
(aref this-array 1 0) => X
```

The subscripts can refer to an element beyond a fill pointer.

```
(setq this-array
      (make-array '(3 2 2) :element-type 'integer :initial-contents
               '(((5 6) (12 8))
                ((7 8) (5 13))
                ((9 4) (22 6)))))

(aref this-array 1 0 0) => 7
```

For a table of related items: See the section "Basic Array Functions".
(zl:arg \(x\)) Function

(zl:arg nil), when evaluated during the application of a lexpr, gives the number of arguments supplied to that lexpr. This is primarily a debugging aid, since lexprs also receive their number of arguments as the value of their lambda-variable.

(zl:arg \(i\)), when evaluated during the application of a lexpr, gives the value of the \(i\)'th argument to the lexpr. \(i\) must be an integer in this case. It is an error if \(i\) is less than 1 or greater than the number of arguments supplied to the lexpr. Example:

```lisp
(defun foo nargs            ; define a lexpr foo.
  (print (arg 2))         ; print the second argument.
  (+ (arg 1)             ; return the sum of the first
     (arg (- nargs 1)))) ; and next to last arguments.
```

zl:arg exists only for compatibility with Maclisp lexprs. To write functions that can accept variable numbers of arguments, use the &optional and &rest keywords. See the section "Evaluating a Function Form".

arglist function &optional real-flag Function

Given an ordinary function, a generic function, or a function spec, returns a representation of the function's lambda-list. It can also return a second value that is a list of descriptive names for the values returned by the function. The third value is a symbol specifying the type of function:

<table>
<thead>
<tr>
<th>Returned Value</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>ordinary or generic function</td>
</tr>
<tr>
<td>subst</td>
<td>substitutable function</td>
</tr>
<tr>
<td>special</td>
<td>special form</td>
</tr>
<tr>
<td>macro</td>
<td>macro</td>
</tr>
<tr>
<td>si:special-macro</td>
<td>both a special form and a macro</td>
</tr>
<tr>
<td>array</td>
<td>array</td>
</tr>
</tbody>
</table>

If function is a symbol, arglist of its function definition is used.

Some functions' real argument lists are not what would be most descriptive to a user. A function can take an &rest argument for technical reasons even though there are standard meanings for the first element of that argument. For such cases, the definition of the function can specify, with a local declaration, a value to be returned when the user asks about the argument list. Example:

```lisp
(defun foo (&rest rest-arg)
  (declare (arglist x y &rest z))
  
  . . . .
```

Note that since the declared argument list is supplied by the user, it does not necessarily correspond to the function's actual argument list.

real-flag allows the caller of arglist to say that the real argument list should be used even if a declared argument list exists.
If `real-flag` is `t` or a declared argument list does not exist, `arglist` computes its return value using information associated with the function. Normally the computed argument list is the same as that supplied in the source definition, but occasionally some differences occur. However, `arglist` always returns a functionally correct answer in that the number and type of the arguments is correct.

When a function returns multiple values, it is useful to give the values names so that the caller can be reminded which value is which. By means of a `values` declaration in the function's definition, entirely analogous to the `arglist` declaration above, you can specify a list of mnemonic names for the returned values. This list is returned by `arglist` as the second value.

```
(arglist 'arglist)
  => (function &optional real-flag) and (arglist values type)
```

### args-info `fcn`

Returns an integer called the "numeric argument descriptor" of `fcn`, which describes the way the function takes arguments. This descriptor is used internally by the microcode, the evaluator, and the compiler. `fcn` can be a function or a function spec.

The information is stored in various bits and byte fields in the integer, which are referenced by the symbolic names shown below. By the usual Symbolics convention, those starting with a single "$%" are bit-masks (meant to be bit-tested with the number with `logand` or `z:bit-test`), and those starting with "$%%" are byte descriptors (meant to be used with `ldb` or `ldb-test`).

Here are the fields:

**sys:$%arg-desc-min-args**

This is the minimum number of arguments that can be passed to this function, that is, the number of "required" parameters.

**sys:$%arg-desc-max-args**

This is the maximum number of arguments that can be passed to this function, that is, the sum of the number of "required" parameters and the number of "optional" parameters. If there is an `&rest` argument, this is not really the maximum number of arguments that can be passed; an arbitrarily large number of arguments is permitted, subject to limitations on the maximum size of a stack frame (about 200 words).

**sys:$%arg-desc-rest-arg**

If this is nonzero, the function takes an `&rest` argument or `&key` arguments. A greater number of arguments than `sys:$%arg-desc-max-args` can be passed.

**sys:$arg-desc-interpreted**

This function is not a compiled-code object.

**sys:$%arg-desc-interpreted**

This is the byte field corresponding to the `sys:$arg-desc-interpreted` bit.
sys:%%arg-desc-quoted
This is obsolete.

sys:%%args-info function
An internal function; it is like args-info, but does not work for interpreted functions. Also, function must be a function, not a function spec.

zl:argument-typecase arg-name &body clauses
Special Form
A hybrid of zl:typecase and zl:check-arg-type. Its clauses look like clauses to zl:typecase. zl:argument-typecase automatically generates an otherwise clause which signals an error. The proceed types to this error are similar to those from zl:check-arg; that is, you can supply a new value that replaces the argument that caused the error.

For example, this:

(defun foo (x)
  (argument-typecase x
   ([:symbol (print 'symbol)])
   ([:number (print 'number)])))

is the same as this:

(defun foo (x)
  (check-arg x
    (typecase x
     ([:symbol (print 'symbol)] t)
     ([:number (print 'number)] t)
     (otherwise nil))
    "a symbol or a number")

For a table of related items: See the section “Condition-Checking and Signalling Functions and Variables”.

array &optional (element-type '* ) (dimensions '* )
TypeSpecifier
array is the type specifier symbol for the Lisp data structure of that name.
The types array, cons, symbol, number, and character are pairwise disjoint.
The type array is a supertype of the types:
  simple-array
  vector

This type specifier can be used in either symbol or list form. Used in list form, array allows the declaration and creation of specialized arrays whose members are all members of the type element-type and whose dimensions match dimensions.
element-type must be a valid type specifier, or unspecified. For standard Symbolics
Common Lisp type specifiers: See the section "Type Specifiers".

dimensions can be a non-negative integer, which is the number of dimensions, or it
can be a list of non-negative integers representing the length of each dimension
(any of which can be an asterisk). dimensions can also be an asterisk.

Note that (array t) is a proper subset of (array *). This is because (array t) is
the set of arrays that can hold any Symbolics Common Lisp object (the elements
are of type t, which includes all objects). On the other hand, (array *) is the set
of all arrays whatsoever, including for example arrays that can hold only charac-
ters. (array character) is not a subset of (array t); the two sets are in fact dis-
joint because (array character) is not the set of all arrays that can hold charac-
ters, but rather the set of arrays that are specialized to hold precisely characters
and no other objects. To test whether an array foo can hold a character, you
should not use

(typep foo ’(array character))

but rather

(subtypep ’character (array-element-type foo))

Examples:

(setq example-array (make-array '(3) :fill-pointer 2))
=> #<ART-Q-3 43063275>
(typep example-array ’array) => T
(typep example-array ’simple-array) => NIL
; simple arrays do not have fill-pointers.
(zl:typep #*101) => :ARRAY
(subtypep ’array t) => T and T
(array-has-fill-pointer-p example-array) => T
(arrayp example-array) => T
(sys:type-arglist ’array)
=> (OPTIONAL (ELEMENT-TYPE ’*) (DIMENSIONS ’*)) and T

See the section "Data Types and Type Specifiers".

See the section "Arrays".

zl:array x type &rest dimlist

Macro

Creates an sys:art-q type array in sys:default-cons-area with the given dimen-
sions. (That is, dimlist is given to zl:make-array as its first argument.) type is ig-
nored. If x is nil, the array is returned; otherwise, the array is put in the function
cell of symbol, and symbol is returned. This exists for Maclisp compatibility.

Use the Common Lisp function make-array in your new programs.

zl:*array x type &rest dimlist

Function
Creates an \texttt{sys:art-q} type array in \texttt{sys:default-cons-area} with the given dimensions, and evaluates all of the arguments. It exists for Maclisp compatibility.

\texttt{zl:array-#-dims} \texttt{array} \quad \textit{Function}

We recommend that you use the function \texttt{array-rank}, which is the Common Lisp equivalent of \texttt{zl:array-#-dims}.

Returns the dimensionality of \texttt{array}. For example:

\begin{verbatim}
(zl:array-#-dims (make-array '(3 5))) => 2
\end{verbatim}

For a table of related items: See the section "Getting Information About an Array".

\texttt{zl:array-active-length} \texttt{array} \quad \textit{Function}

Returns the number of active elements in \texttt{array}. If \texttt{array} does not have a fill pointer, this returns whatever \texttt{(array-total-size array)} would have. If \texttt{array} does have a fill pointer that is a non-negative fixnum, \texttt{zl:array-active-length} returns it. See the section "Array Leaders".

A general explanation of the use of fill pointers is in that section.

Note that \texttt{length} provides the same functionality for lists and vectors.

\texttt{sys:array-bits-per-element} \quad \textit{Variable}

The value of \texttt{sys:array-bits-per-element} is an association list that associates each array type symbol with the number of bits of unsigned numbers (or fixnums) it can hold, or \texttt{nil} if it can hold Lisp objects. This can be used to tell whether an array can hold Lisp objects or not. See the section "Association Lists".

For a table of related items: See the section "Array Representation Tools".

\texttt{sys:array-bits-per-element} \texttt{index} \quad \textit{Function}

Given the internal array-type code numbers, returns the number of bits per cell for unsigned numeric arrays, or \texttt{nil} for a type of array that can contain Lisp objects.

\texttt{array-dimension} \texttt{array dimension-number} \quad \textit{Function}

Returns the length of the dimension numbered \texttt{dimension-number} of \texttt{array}. \texttt{dimension-number} should be a non-negative integer less than the rank of \texttt{array}.

\begin{verbatim}
(setq foo (make-array '(3 2 4 6)))
(array-dimension foo 0) => 3
(array-dimension foo 3) => 6
\end{verbatim}

For a table of related items: See the section "Getting Information About an Array".
array-dimension-limit

Represents the upper exclusive bound on each individual dimension of an array. The value of this is 134217728 under Genera, and CLOE.

\[
\text{(when (> max-number-in-categories array-dimension-limit)}
\text{(setq *number-of-arrays-needed*}
\text{(ceiling max-number-in-categories array-dimension-limit)))}
\]

For a table of related items: See the section "Basic Array Functions".

zl:array-dimension-n n array

Returns the size for the specified dimension of the array. array can be any kind of array, and n should be an integer. If n is between 1 and the dimensionality of array, this returns the nth dimension of array. If n is 0, this returns the length of the leader of array; if array has no leader it returns nil. If n is any other value, this returns nil. Examples:

\[
\text{(setq a (make-array '(3 5) :leader-length 7))}
\text{(zl:array-dimension-n 1 a) => 3}
\text{(zl:array-dimension-n 2 a) => 5}
\text{(zl:array-dimension-n 3 a) => nil}
\text{(zl:array-dimension-n 0 a) => 7}
\]

We recommend that you use the function array-dimension, which is the Common Lisp equivalent of zl:array-dimension-n.

array-dimensions array

Returns a list whose elements are the dimensions of array. Example:

\[
\text{(setq a (make-array '(3 5)))}
\text{(array-dimensions a) => (3 5)}
\]

For a table of related items: See the section "Getting Information About an Array".

sys:array-displaced-p array

Tests whether the array is a displaced array. array can be any kind of array. This predicate returns t if array is any kind of displaced array (including an indirect array). Otherwise it returns nil.

For a table of related items: See the section "Getting Information About an Array".

sys:array-element-byte-size array

Given an array, returns the number of bits that fit into an element of that array. For arrays that can hold general Lisp objects, the result is 32; this assumes that you are storing bits into the array with sys:%logdpb, rather than storing numbers into the array with dpb.
For a table of related items: See the section "Array Representation Tools".

**sys:array-element-size array**  
*Function*  
Given an array, returns the number of bits that fit into an element of that array. For arrays that can hold general Lisp objects, the result is 31; this assumes that you are storing fixnums in the array and manipulating their bits with `dpb` (rather than `sys:%logdpb`). You can store any number of bits per element in an array that holds general Lisp objects, by letting the elements expand into bignums.

For a table of related items: See the section "Array Representation Tools".

**array-element-type array**  
*Function*  
Returns the type specifier of the elements allowed in the `array`. In some cases this may be different than the element-type specified in the call to `make-array`. Example:

```
(setq a (make-array '(3 5)))
(array-element-type a) => T
(array-element-type "foo") => STRING-CHAR
(setq bar (make-array '(3 2 4) :element-type 'bit))
(array-element-type bar) => (integer 0 (2))
```

For a table of related items: See the section "Getting Information About an Array".

**sys:array-elements-per-q index**  
*Function*  
Given the internal array-type `index`, returns the number of array elements stored in one word, for an array of that type.

For a table of related items: See the section "Array Representation Tools".

**sys:array-elements-per-q index**  
*Variable*  
This is an association list that associates each array type symbol with the number of array elements stored in one word, for an array of that type. See the section "Association Lists".

For a table of related items: See the section "Array Representation Tools".

**zl:array-grow array &rest dimensions**  
*Function*  
Creates a new array of the same type as `array`, with the specified dimensions. Those elements of `array` that are still in bounds are copied into the new array. The elements of the new array that are not in the bounds of `array` are initialized to `nil` or 0 as appropriate. If `array` has a leader, the new array has a copy of it. `zl:array-grow` returns the new array and also forwards `array` to it, like `adjust-array`.

For a table of related items: See the section "Array Representation Tools".
Unlike `adjust-array`, `zl:array-grow` usually creates a new array rather than growing or shrinking the array in place. (If the array is one-dimensional and it is being shrunk, `zl:array-grow` does not create a new array.) `zl:array-grow` of a multi-dimensional array can change all the subscripts and move the elements around in memory to keep each element at the same logical place in the array.

`array-has-fill-pointer-p array`  
Function

Returns `t` if the array has a fill pointer; otherwise it returns `nil`. `array` can be any array.

```lisp
(setq foo (make-array 12 :element-type 'string-char :fill-pointer 0))

(array-has-fill-pointer-p foo) => t
```

`array-has-leader-p array`  
Function

Returns `t` if `array` has a leader; otherwise it returns `nil`. `array` can be any array.

For a table of related items: See the section "Operations on Array Leaders". Also: See the section "Getting Information About an Array".

`array-in-bounds-p array &rest subscripts`  
Function

Checks whether `subscripts` is a valid set of subscripts for `array`, and returns `t` if they are; otherwise it returns `nil`.

In the following example, the second set of indices returns an out-of-bounds result because Common Lisp arrays are zero based. Therefore, 2 is the highest allowable index for a dimension of 3.

```lisp
(setq foo (make-array '(3 2 4 6)))

(array-in-bounds foo 2 1 3 5) => t
(array-in-bounds foo 3 1 3 5) => nil
```

For a table of related items: See the section "Getting Information About an Array".

`sys:array-indexed-p array`  
Function

Returns `t` if `array` is an indirect array with an index-offset. Otherwise it returns `nil`. `array` can be any kind of array. Note, however, that displaced arrays with an offset are not considered indexed.

`sys:array-indirect-p array`  
Function

Returns `t` if `array` is an indirect array. Otherwise it returns `nil`. `array` can be any kind of array.
**array-leader** \( \text{array index} \)  

*Function*

Returns the indexed element of \( \text{array}'s \) leader. \( \text{array} \) should be an array with a leader, and \( \text{index} \) should be an integer.

For a table of related items: See the section "Operations on Array Leaders".

**array-leader-length** \( \text{array} \)  

*Function*

Returns the length of \( \text{array}'s \) leader if it has one, or \text{nil} if it does not. \( \text{array} \) can be any array.

For a table of related items: See the section "Getting Information About an Array".

**array-leader-length-limit**  

*Variable*

This is the exclusive upper bound of the length of an array leader. It is 1024 on Symbolics 3600-family computers, 256 on Ivory-based machines.

```lisp
(condition-case (err)
  (make-array 4 :leader-length array-leader-length-limit)
  (error (princ err)))
=> Leader length specified (1024) is too large.
#<FERROR 60065043>
```

**zl:array-length** \( \text{array} \)  

*Function*

We recommend that you use the function **array-total-size**, which is the Common Lisp equivalent of **zl:array-length**.

Returns the total number of elements in \( \text{array} \). \( \text{array} \) can be any array. The total size of a one-dimensional array is calculated without regard for any fill pointer. For a one-dimensional array, **zl:array-length** returns one greater than the maximum allowable subscript. For example:

```lisp
(zl:array-length (make-array 3)) => 3
(zl:array-length (make-array '(3 5))) => 15
```

Note that if fill pointers are being used and you want to know the active length of the array, you should use **length** or **zl:array-active-length** instead of **zl:array-length**.

**zl:array-length** does not return the same value as the product of the dimensions for conformal arrays.

For a table of related items: See the section "Getting Information About an Array".

**zl:array-pop** \( \text{array} \) 

&optional (default \text{nil})  

*Function*

We recommend that you use the function **vector-pop**, which is the Common Lisp equivalent of the function **zl:array-pop**.
Decreases the fill pointer by one and returns the array element designated by the new value of the fill pointer. array must be a one-dimensional array that has a fill pointer.

The second argument, if supplied, is the value to be returned if the array is empty. If \texttt{zl:array-pop} is called with one argument and the array is empty, it signals an error.

The two operations (decrementing and array referencing) happen uninterruptibly. If the array is of type \texttt{sys:art-q-list}, an operation similar to \texttt{nbutlast} has taken place. The cdr coding is updated to ensure this.

See the function \texttt{vector-pop}.

\begin{verbatim}
\texttt{zl:array-push} \hspace{1em} \textit{array x} \\
\end{verbatim}

Function

Attempts to store \textit{x} in the element of the array designated by the fill pointer and increase the fill pointer by one. \textit{array} must be a one-dimensional array that has a fill pointer, and \textit{x} can be any object allowed to be stored in the array. If the fill pointer does not designate an element of the array (specifically, when it gets too big), it is unaffected and \texttt{zl:array-push} returns \texttt{nil}; otherwise, the two actions (storing and incrementing) happen uninterruptibly, and \texttt{zl:array-push} returns the former value of the fill pointer, that is, the array index in which it stored \textit{x}.

If the array is of type \texttt{sys:art-q-list}, an operation similar to \texttt{nconc} has taken place, in that the element has been added to the list by changing the cdr of the formerly last element. The cdr coding is updated to ensure this.

See the function \texttt{vector-push}.

\begin{verbatim}
\texttt{zl:array-push-extend} \hspace{1em} \textit{array x} \hspace{1em} \text{&optional extension} \\
\end{verbatim}

Function

Similar to \texttt{zl:array-push} except that if the fill pointer gets too large, the array is grown to fit the new element; that is, it never "fails" the way \texttt{zl:array-push} does, and so never returns \texttt{nil}. extension is the number of elements to be added to the array if it needs to be grown. It defaults to something reasonable, based on the size of the array. \texttt{zl:array-push-extend} returns the former value of the fill pointer, that is, the array index in which it stored \textit{x}.

See the function \texttt{vector-push-extend}.

\begin{verbatim}
\texttt{zl:array-push-portion-extend} \hspace{1em} \textit{to-array from-array} \hspace{1em} \text{&optional (from-start 0) from-end} \\
\end{verbatim}

Function

We recommend that you use the function \texttt{vector-push-portion-extend}, which is the Symbolics Common Lisp equivalent of the function \texttt{zl:array-push-portion-extend}.

Copies a portion of one array to the end of another, updating the fill pointer of the other to reflect the new contents. The destination array must have a fill pointer. The source array need not. This is equivalent to numerous \texttt{zl:array-push-extend}
calls, but more efficient. **zl:array-push-portion-extend** returns the *to-array* and the index of the next location to be filled.

Example:

```
(setq to-string
    (zl:array-push-portion-extend to-string
      from-string
        (or from 0)
      to))
```

This is similar to **zl:array-push-extend** except that it copies more than one element and has different return values. The arguments default in the usual way, so that the default is to copy all of *from-array* to the end of *to-array*.

**zl:array-push-portion-extend** adjusts the array size using **adjust-array**. It picks the new array size in the same way that **zl:array-push-extend** does, making it bigger than needed for the information being added. In this way, successive additions do not each end up consing a new array. **zl:array-push-portion-extend** uses **copy-array-portion** internally.

See the function **vector-push-portion-extend**.

---

**array-rank array**

*Function*

Returns the number of dimensions of *array*. For example:

```
(array-rank (make-array '(3 5))) => 2
```

For a table of related items: See the section "Getting Information About an Array".

**array-rank-limit**

*Constant*

Represents the exclusive upper bound on the rank of an array. The value of this is 8 under Genera, and 256 under CLOE.

```
(when (> number-of-categories array-rank-limit)
    (setq *number-of-arrays-needed*
      (ceiling number-of-categories array-rank-limit)))
```

For a table of related items: See the section "Basic Array Functions".

**array-row-major-index array &rest subscripts**

*Function*

Takes an array and valid subscripts for the array and returns a single positive integer, less than the total size of the array, that identifies the accessed element in the row-major ordering of the elements. The number of *subscripts* supplied must equal the rank of the array. Each subscript must be a nonnegative integer less than the corresponding array dimension. Like **aref**, **array-row-major-index** returns the position whether or not that position is within the active part of the array.

For example:
window is a conformal array whose 0,0 coordinate is at 256,256 of big-array. The following code creates a 1/4 size portal into the center of big-array.

;;; -*- Syntax: Zetalisp; Package: USER; Base: 10; Mode: LISP -*-
(setq big-array (make-array '(1024 1024) :type 'art-q
   :initial-value 0))
(setq window (make-array '(512 512) :type 'art-q
   :displaced-to big-array
   :displaced-index-offset
   (array-row-major-index big-array 256 256)
   :displaced-conformally t))

For a one-dimensional array, the result of array-row-major-index equals the supplied subscript.

An error is signalled if some subscript is not valid.

array-row-major-index can be used with the :displaced-index-offset option of make-array to construct the desired value for multidimensional arrays.

(setq foo (make-array '(2 3 3) :initial-contents
   '(((0 1 2) (3 4 5) (6 7 8))
     ((9 10 11) (12 13 14) (15 16 17))))
(array-row-major-index foo 0 2 2) => 8

For a table of related items: See the section "Getting Information About an Array".

sys:array-row-span array

Function

Returns the number of array elements spanned by one of its rows, given a two-dimensional array. Normally, this is just equal to the length of a row (that is, the number of columns), but for conformally displaced arrays, the length and the span are not equal.

(sys:array-row-span (make-array '(4 5))) => 5
(sys:array-row-span (make-array '(4 5)
   :displaced-to (make-array '(8 9))
   :displaced-conformally t))

=> 9

Note: If the array is conceptually a raster, it is better to use decode-raster-array than sys:array-row-span.

For a table of related items: See the section "Getting Information About an Array". See the section "Accessing Multidimensional Arrays as One-dimensional".

array-total-size array

Function

Returns the total number of elements in array. The total size of a one-dimensional array is calculated without regard for any fill pointer.

(array-total-size (make-array '(3 5 2))) => 30
Note that if fill pointers are being used and you want to know the active length of
the array, you should use \texttt{length} or under Genera, \texttt{zl:array-active-length}.
\texttt{array-total-size} does not return the same value as the product of the dimensions
for Genera conformal arrays.

For a table of related items: See the section "Getting Information About an Array".

\texttt{array-total-size-limit} \quad \textit{Constant}

Represents the exclusive upper bound on the number of elements of an array. The
value of this is 134217728 under Genera and CLOE.

\begin{verbatim}
 occas (when (> number-of-data-elements array-total-size-limit)
     (setq *number-of-arrays-needed*
         (ceiling number-of-data-elements array-total-size-limit)))
\end{verbatim}

For a table of related items: See the section "Basic Array Functions".

\texttt{sys:array-type \textit{array}} \quad \textit{Function}

Returns the symbolic type of \textit{array}. Example:

\begin{verbatim}
 (sys:array-type (make-array '(3 5))) => SYS:ART-Q
\end{verbatim}

\texttt{sys:*array-type-codes*} \quad \textit{Variable}

The value of \texttt{sys:*array-type-codes*} is a list of all of the array type symbols such
as \texttt{sys:art-q, sys:art-4b, sys:art-string} and so on. The values of these symbols are
internal array type code numbers for the corresponding type.

For a table of related items: See the section "Array Representation Tools".

\texttt{sys:array-types \textit{index}} \quad \textit{Function}

Returns the symbolic name of the array type. The \textit{index} is the internal numeric
code stored in \texttt{sys:*array-type-codes*}.

For a table of related items: See the section "Array Representation Tools".

\texttt{zl:arraydims \textit{array}} \quad \textit{Function}

Returns a list whose first element is the symbolic name of the type of \textit{array}, and
whose remaining elements are its dimensions. \textit{array} can be any array; it also can
be a symbol whose function cell contains an array (for Maclisp compatibility).

Example:

\begin{verbatim}
 (setq a (make-array '(3 5)))
 (zl:arraydims a) => (sys:art-q 3 5)
\end{verbatim}

Note: the list returned by \texttt{(array-dimensions \textit{x})} is equal to the cdr of the list re-
turned by \texttt{(zl:arraydims \textit{x})}.
See the function array-dimensions.

**arrayp object**  
*Function*

Returns t if its argument is an array, otherwise nil. Note that strings are arrays.

```
(setq screen (make-array (640 350) :element-type 'bit))
(arrayp screen) => t
(arrayp "foo") => t
(arrayp '((a b)(c d))) => nil
```

**zl:as-1 value array index**  
*Function*

This is an obsolete version of zl:aset that works only for one-dimensional arrays. There is no reason ever to use it.

**zl:as-2 value array index1 index2**  
*Function*

This is an obsolete version of zl:aset that works only for two-dimensional arrays. There is no reason ever to use it.

**zl:ascii n**  
*Function*

Returns a symbol whose printname is the character n. n can be an integer (a character code), a character, a string, or a symbol.

Examples:

```
(zl:ascii 2) => α
(zl:ascii '#\y) => ¥
(zl:ascii "γ") => Γ
(zl:ascii 'a) => A
```

The symbol returned is interned in the current package.

This function is provided for Maclisp compatibility only.

**ascii-code spec**  
*Function*

Returns an integer that is the ASCII code named by spec. If spec is a character, char-to-ascii is called. Otherwise, spec can be a string or keyword that names one of the ASCII special characters.

ascii-code returns an integer; for example, (ascii-code #\RETURN) => #\15. ascii-code also recognizes strings and looks up the names of the ASCII "control" characters. Thus (ascii-code "SOH") and (ascii-code #\\↓) return 1. (ascii-code #\c-A) returns 65, not 1; there is no mapping between Symbolics character set control characters and ASCII control characters.
Valid ASCII special character names are listed below. All numbers are in octal.

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>000</td>
</tr>
<tr>
<td>HT</td>
<td>011</td>
</tr>
<tr>
<td>DC1</td>
<td>021</td>
</tr>
<tr>
<td>SUB</td>
<td>032</td>
</tr>
<tr>
<td>SOH</td>
<td>001</td>
</tr>
<tr>
<td>LF</td>
<td>012</td>
</tr>
<tr>
<td>DC2</td>
<td>022</td>
</tr>
<tr>
<td>ESC</td>
<td>033</td>
</tr>
<tr>
<td>STX</td>
<td>002</td>
</tr>
<tr>
<td>NL</td>
<td>012</td>
</tr>
<tr>
<td>DC3</td>
<td>023</td>
</tr>
<tr>
<td>ALT</td>
<td>033</td>
</tr>
<tr>
<td>ETX</td>
<td>003</td>
</tr>
<tr>
<td>VT</td>
<td>013</td>
</tr>
<tr>
<td>DC4</td>
<td>024</td>
</tr>
<tr>
<td>FS</td>
<td>034</td>
</tr>
<tr>
<td>EOT</td>
<td>004</td>
</tr>
<tr>
<td>FF</td>
<td>014</td>
</tr>
<tr>
<td>NAK</td>
<td>025</td>
</tr>
<tr>
<td>GS</td>
<td>035</td>
</tr>
<tr>
<td>ENQ</td>
<td>005</td>
</tr>
<tr>
<td>CR</td>
<td>015</td>
</tr>
<tr>
<td>SYN</td>
<td>026</td>
</tr>
<tr>
<td>RS</td>
<td>036</td>
</tr>
<tr>
<td>ACK</td>
<td>006</td>
</tr>
<tr>
<td>SO</td>
<td>016</td>
</tr>
<tr>
<td>ETB</td>
<td>027</td>
</tr>
<tr>
<td>US</td>
<td>037</td>
</tr>
<tr>
<td>BEL</td>
<td>007</td>
</tr>
<tr>
<td>SI</td>
<td>017</td>
</tr>
<tr>
<td>CAN</td>
<td>030</td>
</tr>
<tr>
<td>SP</td>
<td>040</td>
</tr>
<tr>
<td>BS</td>
<td>010</td>
</tr>
<tr>
<td>DLE</td>
<td>020</td>
</tr>
<tr>
<td>EM</td>
<td>031</td>
</tr>
<tr>
<td>DEL</td>
<td>177</td>
</tr>
<tr>
<td>TAB</td>
<td>011</td>
</tr>
</tbody>
</table>

For a table of related items, see the section "ASCII Characters".

**ascii-to-char code**

Converting *code* (an ASCII code) to the corresponding character. The caller must ignore LF after CR if desired.

*ascii-to-char* performs an inverse mapping of the function *char-to-ascii*, and this mapping embeds the ASCII character character set in the Symbolics character set. There is no attempt to map more obscure ASCII control codes into the also obscure and unrelated Symbolics control codes. For example, Escape, is a character in the Symbolics character set corresponding to the key marked Escape. The ASCII code Escape is not the same as the Symbolics Escape. See the function *char-to-ascii*. See the function *ascii-code*. See the section "ASCII Conversion String Functions".

The functions *char-to-ascii* and *ascii-to-char* provide the primitive conversions needed by ASCII-translating streams. They do not translate the Return character into a CR-LF pair; the caller must handle that. They just translate #\texttt{Return} into CR and #\texttt{Line} into LF. Except for CR-LF, *char-to-ascii* and *ascii-to-char* are wholly compatible with the ASCII-translating streams.

They ignore Symbolics control characters; the translation of #\texttt{c-G} is the ASCII code for G, not the ASCII code to ring the bell, also known as "control G." (ascii-to-char (ascii-code "BEL")(a)) is #\texttt{\pi}, not #\texttt{c-G}. The translation from ASCII to character never produces a Symbolics control character.

For a table of related items, see the section "ASCII Characters".

**ascii-to-string asci-array**

Converts *asci-array*, an *sys:art-8b* array representing ASCII characters, into a Lisp string. Note that the length of the string can vary depending on whether *asci-array* contained a Newline character or Carriage Return Line Feed characters. See the section "ASCII Characters".

Example:
(setq a-string-array
  (zl:make-array 5 :type zl:art-8b :initial-value (ascii-code #\x)))
=> #(120 120 120 120 120)
(ascii-to-string a-string-array) => "xxxxx"

For a table of related items: See the section "ASCII Conversion String Functions".

**zl:aset element array &rest subscripts**

Stores element into the element of array selected by the subscripts. The subscripts must be integers and their number must match the dimensionality of array. The returned value is element.

Current style suggests using setf and aref instead of zl:aset. For example:

(setf (aref array subscripts...) new-value)

**ash number count**

Shifts number arithmetically left count bits if count is positive, or right -count bits if count is negative. Unused positions are filled by zeroes from the right, and by copies of the sign bit from the left. Thus, unlike lsh, the sign of the result is always the same as the sign of number. If number is an integer, this is a shifting operation. If number is a floating-point number in Genera, this does scaling (multiplication by a power of two), rather than actually shifting any bits. If you are using CLOE, it is an error for number to be a float.

Examples:

(ash 1 3) => 8
(ash 10 3) => 80
(ash 10 -3) => 1
(ash 1 -3) => 0
(ash 1.5 3) => 12.0
(ash -1 3) => -8
(ash -1 -3) => -1

See the section "Functions Returning Result of Bit-wise Logical Operations".

For a table of related items: See the section "Functions Returning Result of Bit-wise Logical Operations".

**asin number**

Computes and returns the arc sine of number. The result is in radians.

The argument can be any noncomplex or complex number. Note that if the absolute value of number is greater than one, the result is complex, even if the argument is not complex.

The arc sine being a mathematically multiple-valued function, asin returns a principal value whose range is that strip of the complex plane containing numbers
with real parts between $-\pi/2$ and $\pi/2$. Any number with a real part equal to $-\pi/2$ and a negative imaginary part is excluded from the range. Also excluded from the range is any number with real part equal to $\pi/2$ and a positive imaginary part.

Examples:

\[
\begin{align*}
\text{(asin 1) } &= 1.5707964 \quad ; \pi/2 \text{ radians} \\
\text{(asin 0) } &= 0.0 \\
\text{(asin -1) } &= -1.5707964 \quad ;-\pi/2 \text{ radians} \\
\text{(asin 2) } &= \#c(1.5707964 \ -1.316958) \\
\text{(asin -2) } &= \#c(-1.5707964 \ 1.3169578) \\
\text{(asin (/ (sqrt 2) 2)) } &= 0.785398
\end{align*}
\]

For a table of related items, see the section "Trigonometric and Related Functions".

**asinh number**

Function

Computes and returns the hyperbolic arc sine of number. The result is in radians. The argument can be any noncomplex or complex number.

The hyperbolic arc sine being mathematically multiple-valued in the complex plane, **asinh** returns a principal value whose range is that strip of the complex plane containing numbers with imaginary parts between $-\pi/2$ and $\pi/2$. Any number with an imaginary part equal to $-\pi/2$ is not in the range if its real part is negative; any number with real part equal to $\pi/2$ is excluded from the range if its imaginary part is positive.

Example:

\[
\text{(asinh 0) } => 0.0 \quad ; (\text{sinh 0) } => 0.0}
\]

For a table of related items, see the section "Hyperbolic Functions".

**zl:ass pred item list**

Function

Looks up item in the association list list. Returns the first cons whose car matches item according to pred, or nil if none does. (zl:ass 'eq a b) is the same as (zl:assq a b). As with zl:mem, you can use noncommutative predicates; the first argument to the predicate is item and the second is the indicator of the element of list. See the function zl:mem.

For a table of related items: See the section "Functions that Operate on Association Lists".

**assert test-form &optional references format-string &rest format-args**

Macro

Signals an error if the value of test-form is nil. It is possible to proceed from this error; the function lets you change the values of some variables, and starts over, evaluating test-form again.

assert returns nil.
test-form is any form.

references is a list, each item of which must be a generalized variable reference that is acceptable to the macro setf. These should be variables on which test-form depends, whose values can sensibly be changed by the user in attempting to correct the error. Subforms of each of references are only evaluated if an error is signalled, and can be re-evaluated if the error is re-signalled (after continuing without actually fixing the problem).

format-string is an error message string.

format-args are additional arguments; these are evaluated only if an error is signalled, and reevaluated if the error is signalled again.

The function format is applied in the usual way to format-string and and format-args to produce the actual error message.

If format-string (and therefore also format-args) are omitted, a default error message is used.

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

assoc item a-list &key (test #'eql) test-not (key #'identity) Function

Searches the association list a-list. The value returned is the first pair in a-list whose car satisfies the predicate specified by :test, or nil if no such pair is found. If nil is one of the elements in the association list, assoc passes over it. The keywords are:

:test Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

Example:

(assoc 'loon '((eagle . raptor) (loon . diver))) =>
(LOON . DIVER)

(assoc 'diver '((eagle . raptor) (loon . diver))) => NIL

(assoc '2 '((1 a b c) (2 b c d) (-7 x y z))) => (2 B C D)

It is possible to rplacd the result of assoc (provided that it is non-nil) in order to update a-list.
(setq values '(((x . 100) (y . 200) (z . 50))) =>
((x . 100) (y . 200) (z . 50))

(assoc 'y values) => (y . 200)

(rplacd (assoc 'y values) 201) => (y . 201)

(assoc 'y values) => (y . 201)

The two expressions:

(assoc item alist :test pred)

and

(find item alist :test pred :key #'car)

are almost equivalent in meaning. The difference occurs when nil appears in a-list in place of a pair, and the item being searched for is nil. In these cases, find computes the car of the nil in a-list, finds that it is equal to item, and returns nil, while assoc ignores the nil in a-list and continues to search for an actual cons whose car is nil. See also, find and position.

It is often better to update an association list by adding new pairs to the front, rather than altering old pairs. The following example demonstrates an association list consisting of pairs of keys and association lists.

(setq financial-statement)
'((MONTHLY-CASH-ON-HAND ((11 . 52) (12 . 73)))
 (MONTHLY-EXPENSE ((10 . 20) (11 . 21)))
 (MONTHLY-REVENUE ((10 . 31) (11 . 42))))

In the following example, the first call to assoc extracts the monthly-cash-on-hand association list. The second assoc extracts the monthly-cash-on-hand for the month of November from monthly-cash-on-hand:

(setq monthly-cash-on-hand
  (assoc 'monthly-cash-on-hand financial-statement))
=> (MONTHLY-CASH-ON-HAND ((11 . 52) (12 . 73)))

(assoc '11 (cdr monthly-cash-on-hand))
=>(11 . 52)

In the next example, rplacd alters a value stored in the association list, and assoc delivers the pointer for rplacd.

(assoc 'monthly-revenue financial-statement)
=> (MONTHLY-REVENUE . ((10 . 31) (11 . 42)))

(setf (cdr (assoc '11 (assoc 'monthly-revenue financial-statement)))
  22)

(assoc 'monthly-revenue financial-statement)
=> (MONTHLY-REVENUE . ((10 . 31) (11 . 22)))

Usually, association lists are updated by adding a new pair to the front of the list, as shown in the following example:
(acons '11 '22 (assoc 'monthly-revenue financial-statement))

(assoc 'monthly-revenue financial-statement)
=> (MONTHLY-REVENUE . ((11 . 22) (10 . 31) (11 . 42)))

For a table of related items: See the section "Functions that Operate on Association Lists".

**zl:assoc item in-list Function**

Looks up *item* in the association list *in-list*. Returns the first cons whose car is
*zl:equal* to *item*, or *nil* if none is found. Example:

```
(zl:assoc '(a b) '((x . y) ((a b) . 7) ((c . d) . e)))
=> ((a b) . 7)
```

*zl:assoc* could have been defined by:

```
(defun assoc (item list)
  (cond ((null list) nil)
        ((equal item (caar list)) (car list))
        ((assoc item (cdr list)))))
```

For a table of related items: See the section "Functions that Operate on Association Lists".

**assoc-if predicate a-list &key :key Function**

Searches the association list *a-list*. Returns the first pair in *a-list* whose car satisfies *predicate*, or *nil* if there is no such pair in *a-list*. The keyword is:

**:key**

If not *nil*, should be a function of one argument that will extract the part to be tested from the whole element. **:key** is a Symbolics extension to Common Lisp.

Example:

```
(assoc-if #'integerp '((eagle . raptor) (1 . 2))) =>
(1 . 2)
```

```
(assoc-if #'symbolp '((eagle . raptor) (1 . 2))) =>
(EAGLE . RAPTOR)
```

```
(assoc-if #'floatp '((eagle . raptor) (1 . 2))) =>
NIL
```

In the following example, the function finds the largest numeric key in an association list by repeating **assoc-if** with a test for a key greater than the greatest key found so far.
(defun find-largest-key (a-list &optional (start 0))
 (if (setq pair
    (assoc-if #'(lambda(x) (> x start)) a-list))
       (find-largest-key a-list (car pair))))

For a table of related items: See the section "Functions that Operate on Association Lists".

**Compatibility Note:** :key is a Symbolics extension to Common Lisp, not available in CLOE.

*assoc-if-not* predicate a-list &key :key

Function

Searches the association list *a-list*. The value returned is the first pair in *a-list* whose car does not satisfy *predicate*, or *nil* if there is no such pair in *a-list*. The keyword is:

:key

If not *nil*, should be a function of one argument that will extract the part to be tested from the whole element. :key is a Symbolics extension to Common Lisp.

Example:

(assoc-if-not #'integerp '((eagle . raptor) (1 . 2))) =>
(EAGLE . RAPTOR)

(assoc-if-not #'symbolp '((eagle . raptor) (1 . 2))) =>
(1 . 2)

(assoc-if-not #'symbolp '((eagle . raptor) (loon . diver))) =>
NIL

In the following example, the call to *assoc-if-not* finds the first pair in *a-list* such that its key is not *string-equal* to "salary".

(assoc-if-not #'(lambda(x) (string-equal "salary" x))
  a-list)

For a table of related items: See the section "Functions that Operate on Association Lists".

**Compatibility Note:** :key is a Symbolics extension to Common Lisp, not available in CLOE.

*zl:assq* item in-list

Function

Looks up *item* in the association list *in-list*. The value is the first cons whose car is *eq* to *item*, or *nil* if none is found. Examples:

(zl:assq 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
=> (r . x)
(zl:assq 'foo '(((foo . bar) (zoo . goo)))
=> nil

(zl:assq 'b '(((a b c) (b c d) (x y z)))
=> (b c d)

You can replace the result of zl:assq as long as it is not nil, if you want to update the "table" in-list. Example:

(setq values '(((x . 100) (y . 200) (z . 50)))
(zl:assq 'y values) => (y . 200)
(rplacd (zl:assq 'y values) 201)
(zl:assq 'y values) => (y . 201) now

A typical trick is to use (cdr (zl:assq x y)). Since the cdr of nil is guaranteed to be nil, this yields nil if no pair is found (or if a pair is found whose cdr is nil.)

zl:assq could have been defined by:

(defun zl:assq (item list)
  (cond ((null list) nil)
        ((eq item (caar list)) (car list))
        ((zl:assq item (cdr list)))))

For a table of related items: See the section "Functions that Operate on Association Lists".

atan y &optional x

Function

With two arguments, y and x, computes and returns the arc tangent of the quantity y/x. If either argument is a double-float, the result is also a double-float. In the two argument case neither argument can be complex. The returned value is in radians and is always between -π (exclusive) and π (inclusive). The signs of y and x determine the quadrant of the result angle.

Note that either y or x (but not both simultaneously) can be zero. The examples illustrate a few special cases.

With only one argument y, atan computes and returns the arc tangent of y. The argument can be any noncomplex or complex number. The result is in radians and its range is as follows: for a noncomplex y the result is noncomplex and lies between -π/2 and π/2 (both exclusive); for a complex y the range is that strip of the complex plane containing numbers with a real part between -π/2 and π/2. A number with real part equal to -π/2 is not in the range if it has a non-positive imaginary part. Similarly, a number with real part equal to π/2 is not in the range if its imaginary part is non-negative.

Examples:
(atan 0) => 0.0
(atan 0 673) => 0.0 ; (atan (/ y x))
(atan 1 1) => 0.7853982 ; first quadrant
(atan 1 -1) => 2.3561945 ; second quadrant
(atan -1 -1) => -2.3561945 ; third quadrant
(atan -1 1) => -0.7853982 ; fourth quadrant
(atan 1 0) => 1.5707964

(setq theta (/ pi 4)) → 0.785398

(atan (cos theta) (sin theta)) = theta => 0.785398

When given a single argument, atan accepts a complex argument.
(atan (/ (cos theta) (sin theta))) = theta => 0.785398

(atan y) is the same as
(* -1 (log (* (+ 1 (* i y))
  (sqrt (/ 1 (+ 1 (expt y 2)))))))

For a table of related items, see the section "Trigonometric and Related Functions".

zl:atan y x

Returns the angle, in radians, whose tangent is y/x. zl:atan always returns a number between zero and 2π.

Examples:
(zl:atan 1 1) => 0.7853982
(zl:atan -1 -1) => 3.926991

For a table of related items: See the section "Trigonometric and Related Functions".

zl:atan2 y x

Returns the angle, in radians, whose tangent is y/x. zl:atan2 always returns a number between -π and π.

Similar to zl:atan, except that it accepts only noncomplex arguments.

For a table of related items: See the section "Trigonometric and Related Functions".

atanh number

Computes and returns the hyperbolic arc tangent of number. The result is in radians. The argument can be any noncomplex or complex number. Note that if the absolute value of the argument is greater than one, the result is complex even if the argument is not complex.
The hyperbolic arc tangent being mathematically multiple-valued in the complex plane, \texttt{atanh} returns a principal value whose range is that strip of the complex plane containing numbers with imaginary parts between $-\pi/2$ and $\pi/2$. Any number with an imaginary part equal to $-\pi/2$ is not in the range if its real part is non-negative; any number with imaginary part equal to $\pi/2$ is excluded from the range if its real part is non-positive.

Example:

\[(\texttt{atanh 0}) \Rightarrow 0.0/j141\]

For a table of related items, see the section "Hyperbolic Functions".

\textbf{atom object} \hspace{1cm} \textit{Function}

Returns \texttt{t} if \texttt{object} is not a \texttt{cons}, otherwise \texttt{nil}.

Note that \texttt{(atom '())} is true because () is equivalent to \texttt{nil}.

\[(\texttt{atom x})\]

is equivalent to

\[(\texttt{type x 'atom})\]

is equivalent to

\[(\texttt{not (typep x 'cons)})\]

Note that arrays, strings, structures, vectors, numbers, and symbols are all atoms.

\[(\texttt{atom '())} \Rightarrow \texttt{t}\]

\[(\texttt{setq foo (make-array '(4 2)) bar "24" baz '(a foo bar)})\]

\[(\texttt{atom foo}) \Rightarrow \texttt{t}\]

\[(\texttt{atom bar}) \Rightarrow \texttt{t}\]

\[(\texttt{atom baz}) \Rightarrow \texttt{nil}\]

For a table of related items, see the section "Predicates that Operate on Lists".
(atom '()) => t
(setq foo (make-array '(4 2)) bar "24" baz '(a foo bar))
(atom foo) => t
(atom bar) => t
(atom baz) => nil

For a table of related items, see the section "Predicates that Operate on Lists".

atom

atom is the type specifier symbol for the predefined Lisp object of that name.
atom ≡ (not cons).

Examples:
(typep 'a 'atom) => T
(zl:typep 'a) => :SYMBOL
(subtypep 'atom 'common) => NIL and NIL
(atom 'a) => T
(sys:type-arglist 'atom) => NIL and T

See the section "Data Types and Type Specifiers".
See the section "Symbols, Keywords, and Variables".

&aux

Lambda List Keyword

Separates the arguments of a function from the auxiliary variables. If it is present, all specifiers after it are entries of the form:

(variable initial-value-form)

zl:base

Variable

The value of zl:base is a number that is the radix in which integers and ratios are printed in, or a symbol with a si:princ-function property. The initial value of zl:base is 10. zl:base should not be greater than 36 or less than 2.

The printing of trailing decimal points for integers in base 10 is controlled by the value of variable *print-radix*. See the section "Printed Representation of Rational Numbers".

In your new programs use the Common Lisp variable *print-base*.

beep &optional beep-type (stream zl:terminal-io)

Function

Tries to attract the user’s attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the :beep operation, this function sends it a :beep message, passing type along as an argument. Otherwise it
just causes an audible beep on the terminal. type is a keyword selecting among several different beeping noises. The allowed types have not yet been defined; type is currently ignored and should always be nil. See the message :beep.

:beep &optional type

This is supported by interactive streams. It attracts the attention of the user by making an audible beep and/or flashing the screen. type is a keyword selecting among several different beeping noises. The allowed types have not yet been defined; type is currently ignored and should always be nil.

bignum

bignum is the type specifier symbol for the predefined primitive Lisp object of that name.

The types bignum and fixnum are an exhaustive partition of the type integer, since integer = (or bignum fixnum). These two types are internal representations of integers used by the system for efficiency depending on integer size; in general, bignums and fixnums are transparent to the programmer.

Examples:

(typep 1000000000000000000000000000000000 'bignum) => T
(typep '1 'bignum) => NIL
(zl:typep '1000000000000000000000000000000000 'bignum) => :BIGNUM
(subtypep 'bignum 'integer) => T and T ; subtype and certain
(typep 565682366398848748463539404874 'common) => T
(zl:bigp 4444444444455555555555555566666666666666) => T
(sys:type-arglist 'bignum) => NIL and T
(type-of 098893748973837368948449494373639484099876) => BIGNUM

See the section "Data Types and Type Specifiers".
See the section "Numbers".

zl:bigp object

Returns t if object is a bignum, otherwise nil.
For a table of related items, see the section "Numeric Type-checking Predicates".

bit array &rest subscripts

Returns the element of array selected by the subscripts. The subscripts must be integers and their number must match the dimensionality of array. The array must be an array of bits.
(setq foo (make-array (2 3)
   :adjustable t
   :element-type 'bit
   :initial-contents '((1 1 1)
                        (1 0 1))))

(bit foo 1 1) => 0

Note that the bit-array in the previous example is adjustable, and therefore not simple. Therefore, we can not use sbit for foo. We could have used aref, but bit is generally more efficient for bit-arrays.

For a table of related items: See the section "Arrays of Bits".

**bit**

Type Specifier

bit is equivalent to the type (integer 0 1) and (unsigned-byte 1).

**bit-and**  

first second &optional third

Function

Performs logical and operations on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bit-andc1**  

first second &optional third

Function

Performs logical and operations on the complement of first with second on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bit-andc2**  

first second &optional third

Function

Performs logical and operations on first with the complement of second on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bit-eqv**  

first second &optional third

Function

Performs logical exclusive nor operations on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the re-
result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bit-ior** first second &optional third

Performs logical inclusive or operations on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bit-nand** first second &optional third

Performs logical not and operations on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bit-nor** first second &optional third

Performs logical not or operations on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bit-not** source &optional destination

Returns a bit-array of the same rank and dimensions that contains a copy of the argument with all the bits inverted. source must be a bit-array. If destination is nil or omitted, a new array is created to contain the result. If destination is t, the result is destructively placed in the source array.

(bit-not #*1001) => #*0110

For a table of related items:
See the section "Arrays of Bits".

**bit-orc1** first second &optional third

Performs logical or operations on the complement of first with second on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.
For a table of related items: See the section "Arrays of Bits".

**bit-orc2** first second &optional third

*Function*

Performs logical or operations on first with the complement of second on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is **nil** or omitted. If the third argument is **t**, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**zl:bit-test** x y

*Function*

In your new programs, we recommend that you use the function **logtest**, which is the Common Lisp equivalent of the function **zl:bit-test**.

**zl:bit-test** is a predicate that returns **t** if any of the bits designated by the 1's in x are 1's in y.

For a table of related items: See the section "Predicates for Testing Bits in Integers".

**bit-vector** &optional (size '*)

*Type Specifier*

**bit-vector** is the type specifier symbol for the Lisp data structure of that name. The type **bit-vector** is a subtype of the type **vector**; (**bit-vector**) means (**vector bit**).

The type **bit-vector** is a supertype of the type **simple-bit-vector**.

The types (**vector t**), **string**, and **bit-vector** are disjoint.

This type specifier can be used in either symbol or list form. Used in list form, **bit-vector** allows the declaration and creation of specialized types of bit vectors whose size is restricted to the specified size. (**bit-vector size**) means the same as (**array bit (size)**); the set of bit-vectors of the indicated size.

Examples:

```lisp
(setq array-bit-vector
      (make-array '(3) :element-type 'bit :fill-pointer 2))
=> #<ART-1B-3 43015121>
(typep #*10110 'bit-vector) => T
(typep #*1011 'bit-vector) => T
(typep array-bit-vector 'bit-vector) => T
(subtypep 'bit-vector 'vector) => T and T
(bit-vector-p #*) => T ;empty bit vector
(sys:type-arglist 'bit-vector) => (&OPTIONAL (SIZE '*)) and T
```

See the section "Data Types and Type Specifiers".
See the section "Arrays".

**bit-vector-cardinality** bit-vector &key (:start 0) :end

Counts how many of the bits in the range are one's and returns the number found.

*bit-vector* is a one-dimensional array whose elements are required to be bits. See the type specifier *bit-vector*.

*start* and *end* must be non-negative integer indices into the bit-vector. *start* must be less than or equal to *end*, or else an error is signalled. *start* defaults to zero (the start of the bit vector).

*start* indicates the start position for the operation within the bit-vector. *end* is the position of the first element in the bit-vector beyond the end of the operation.

For example:

```lisp
(bit-vector-cardinality #*111111) => 5

(bit-vector-cardinality #*111000) => 3

(bit-vector-cardinality #*1110011 :start 0 :end 5) => 3
```

For a table of related items: See the section "Operations on Vectors".

**bit-vector-disjoint-p** bit-vector-1 bit-vector-2 &key (:start1 0) :end1 (:start2 0) :end2

Tests two bit vectors to see if they are disjoint (have no common positions containing 1's) in a range specified by *start1*, *end1*, *start2*, and *end2*.

*bit-vector-1* and *bit-vector-2* are one-dimensional arrays whose elements are required to be bits. See the type specifier *bit-vector*.

*start1*, *end1*, *start2*, and *end2* must be non-negative integer indices into *bit-vector-1* and *bit-vector-2*. *start1* and *start2* must be less than or equal to *end1* and *end2*, or else an error is signalled. *start1* and *start2* default to zero (the start of the bit vector). If *end* is unspecified or nil, the length *bit-vector* is used.

*start1* and *start2* indicate the start positions for the operation within the bit-vector. *end1* and *end2* are the position of the first element in the bit-vector beyond the end of the operation.

For example:
(bit-vector-disjoint-p #*001000001 #*001000001) => NIL

(bit-vector-disjoint-p #*1110010000 #*1110010011) => NIL

(bit-vector-disjoint-p #*1110010000 #*1110010011 :start1 1 :end1 6 :start2 6 :end2 8) => T

For a table of related items: See the section "Operations on Vectors".

**bit-vector-p object**

Tests whether the given *object* is a bit vector. A bit vector is a one-dimensional array whose elements are required to be bits. See the type specifier **bit-vector**.

(bit-vector-p (make-array 3 :element-type 'bit :fill-pointer 2)) => T

(bit-vector-p (make-array 5 :element-type 'string-char)) => NIL

For a table of related items: See the section "Operations on Vectors".

**bit-vector-position bit bit-vector &key (:start 0) :end**

If *bit-vector* contains an element matching *bit*, returns the index within the bit vector of the leftmost such element as a non-negative integer; otherwise nil is returned.

*bit* is either 0 or 1.

*bit-vector* is a one-dimensional array whose elements are required to be bits. See the type specifier **bit-vector**.

:start and :end must be non-negative integer indices into the bit-vector. :start must be less than or equal to :end, or else an error is signalled. :start defaults to zero (the start of the bit vector). If :end is unspecified or nil, the length *bit-vector* is used.

:start indicates the start position for the operation within the bit vector. :end is the position of the first element in the bit-vector beyond the end of the operation.

For example:

(bit-vector-position 1 #*11111) => 0

(bit-vector-position 1 #*0111111) => 2
(bit-vector-position 1 #*0011111 :start 3 :end 5)
=> 3

(bit-vector-position 0 #*111)
=> NIL

For a table of related items: See the section "Operations on Vectors".

**bit-vector-subset-p**  
*bit-vector-1* *bit-vector-2* &key (:start1 0) (:end1 (:start2 0) :end2)

**Function**

Tests if one bit vector is a subset of another bit vector (subset means that for each position of bit-vector-2 that contains a one, the same position in bit-vector-1 also contains a 1) in a range specified by :start1, :end1, :start2, and :end2.

*bit-vector-1* and *bit-vector-2* are one-dimensional arrays whose elements are required to be bits. See the type specifier **bit-vector**.

:star1, :end1, :star2, and :end2 must be non-negative integer indices into *bit-vector-1* and *bit-vector-2*. :star1 and :start2 must be less than or equal to :end1 and :end2, else an error is signalled. :start1 and :start2 default to zero (the start of the bit vector). If :end is unspecified or nil, the length *bit-vector* is used.

:star1 and :start2 indicate the start position for the operation within the bit vector. :end1 and :end2 are the positions of the first element in the bit vector beyond the end of the operation.

For example:

(bit-vector-subset-p #*00100100111 #*0010010011)
=> T

(bit-vector-subset-p #*1110010011 #*0010010011)
=> NIL

(bit-vector-subset-p #*11100000 #*11100011 :start1 0 :end1 6 :start2 0 :end2 6)
=> T

(bit-vector-subset-p #*11100000 #*11100011 :start1 0 :end1 8 :start2 0 :end2 8)
=> NIL

For a table of related items: See the section "Operations on Vectors".

**bit-vector-zero-p**  
*bit-vector* &key (:start 0) :end

**Function**

Tests if *bit-vector* is a bit vector of zeros in the range specified by :start and :end.

*bit-vector* is a one-dimensional array whose elements are required to be bits.

:star and :end must be non-negative integer indices into the bit-vector. :star must be less than or equal to :end, or else an error is signalled. :start defaults to zero (the start of the bit vector).
:start indicates the start position for the operation within the bit vector. :end is the position of the first element in the bit-vector beyond the end of the operation. See the type specifier bit-vector.

For example:

```lisp
(bit-vector-zero-p #*00000 :start 0 :end 5) => T

(bit-vector-zero-p #*00011) => NIL

(bit-vector-zero-p #*00011 :start 0 :end 3) => T
```

For a table of related items: See the section "Operations on Vectors".

**bit-xor first second &optional third**

Performs logical exclusive or operations on bit arrays. The arguments must be bit arrays of the same rank and dimensions. A new array is created to contain the result if the third argument is nil or omitted. If the third argument is t, the first array is used to hold the result.

For a table of related items: See the section "Arrays of Bits".

**bitblt alu width height from-raster from-x from-y to-raster to-x to-y**

Copies a rectangular portion of from-raster into a rectangular portion of to-raster. from-raster and to-raster must be two-dimensional arrays of bits or bytes (sys:art-1b, sys:art-2b, sys:art-4b, sys:art-8b, sys:art-16b, or sys:art-fixnum). The value stored can be a Boolean function of the new value and the value already there, under the control of alu. This function is most commonly used in connection with raster images for TV displays.

The top-left corner of the source rectangle is:

```
(raster-aref from-raster from-x from-y)
```

The top-left corner of the destination rectangle is:

```
(raster-aref to-raster to-x to-y)
```

width and height are the dimensions of both rectangles. If width or height is zero, bitblt does nothing.

from-raster and to-raster are allowed to be the same array. bitblt normally traverses the arrays in increasing order of x and y subscripts. If width is negative, (abs width) is used as the width, but the processing of the x direction is done backwards, starting with the highest value of x and working down. If height is negative it is treated analogously. When bitblting an array to itself, when the two rectangles overlap, it might be necessary to work backwards to achieve the desired
effect, such as shifting the entire array upwards by a certain number of rows.
Note that negativity of width or height does not affect the \((x,y)\) coordinates specified by the arguments, which are still the top-left corner even if \textbf{bitblt} starts at some other corner.

If the two arrays are of different types, \textbf{bitblt} works bit-wise and not element-wise. That is, if you \textbf{bitblt} from an \texttt{sys:art-2b} raster into an \texttt{sys:art-4b} raster, then two elements of the \texttt{from-raster} correspond to one element of the \texttt{to-raster}. width is in units of elements of the \texttt{to-raster}. Note that the \texttt{width} and \texttt{height} arguments are relative to the \texttt{to-raster} array, not the \texttt{from-raster} array.

If \textbf{bitblt} goes outside the bounds of the source array, it wraps around. This allows such operations as the replication of a small stipple pattern through a large array. If \textbf{bitblt} goes outside the bounds of the destination array, it signals an error.

If \texttt{src} is an element of the source rectangle, and \texttt{dst} is the corresponding element of the destination rectangle, then \textbf{bitblt} changes the value of \texttt{dst} to \texttt{(boole alu src dst)}. The following are the symbolic names for some of the most useful \texttt{alu} functions:

\begin{verbatim}
tv:alu-seta plain copy
tv:alu-setz set destination to 0
tv:alu-ior inclusive or
tv:alu-xor exclusive or
tv:alu-andca and with complement of source
\end{verbatim}

For a chart of more \texttt{alu} possibilities: See the function \texttt{boole}.

\textbf{bitblt} is written in highly optimized microcode and goes very much faster than the same thing written with ordinary raster operations would. Unfortunately this causes \textbf{bitblt} to have a couple of strange restrictions. Wraparound does not work correctly if \texttt{from-raster} is an indirect array with an index offset. On black-and-white screens, \textbf{bitblt} signals an error if the \texttt{widths} of \texttt{from-raster} and \texttt{to-raster} are not both integral multiples of the machine word length. On color screens, the product of the number of bits per raster element and the width must be an integral multiple of 32. You can determine the number of bits per raster element by the number of bits which correspond to a single pixel on the screen. For \texttt{sys:art-1b} arrays, \texttt{width} must be a multiple of 32, for \texttt{sys:art-2b} arrays it must be a multiple of 16, and so on. Use \texttt{:draw-1-bit-raster} rather than \textbf{bitblt} in programs that run without modification on color screens.

For a table of related items: See the section "Operations on Rasters". Also: See the section "Copying an Array".

\textbf{block name} \&\texttt{body body}

\begin{verbatim}
Special Form
\end{verbatim}

Provides an exit context for the evaluation of its body argument. Evaluates each \texttt{form} in sequence and normally returns the (possibly multiple) values of the last \texttt{form}. However, \texttt{(return-from name value)} or \texttt{(return or (return (values-list list))}
form) might be evaluated during the evaluation of some form. In that case, the
(possibly multiple) values that result from evaluating value are immediately re-
turned from the innermost block that has the same name and that lexically con-
tains the return-from form. Any remaining forms in that block are not evaluated.
name is not evaluated. It must be a symbol.
The scope of name is lexical. That is, the return-from form must be inside the
block itself (or inside a block that that block lexically contains), not inside a func-
tion called from the block.
do, prog, and their variants establish implicit blocks around their bodies; you can
use return-from to exit from them. These blocks are named nil unless you specify
a name explicitly.
Examples:
   (block nil
      (print "clear")
      (return)
      (print "open")( => "clear" NIL

   (let ((x 2400))
      (block time-x
        (when (= x 2400)
          (return-from time-x "time to go")
          ("time time time")( => "time to go"

   (defun bar ()
      (princ "zero ")
      (block a
        (princ "one ") (return-from a "two ")
        (princ "three ")
        (princ "four ")
      t) => BAR
      (bar) => zero one four T

   (block negative
      (mapcar (function (lambda (x)
        (cond ((minusp x)
          (return-from negative x))
          (t (f x))))
      y))

   (block foo
      (let ((num *a-number*)
        (result 0))
        (dotimes (i num result)
          (if (= i 20) (return-from foo result))
          (setq result (+ result (expt i 2)))))

   defun establishes an implicit block whose name is the same as that of the defined
(defun matrix-find (elt matrix)
  (dotimes (i (array-dimension matrix 0))
    (dotimes (j (array-dimension matrix 1))
      (if (eql elt (aref matrix i j))
        (return-from matrix-find (values i j)))))

The following two forms are equivalent:

  (cond ((predicate x)
   (do-one-thing))
    (t
      (format t "The value of X is ~S~%" x)
      (do-the-other-thing)
      (do-something-else-too)))

  (block deal-with-x
    (when (predicate x)
      (return-from deal-with-x (do-one-thing)))
    (format t "The value of X is ~S~%" x)
    (do-the-other-thing)
    (do-something-else-too))

The interpreter and compiler generate implicit blocks for functions whose name is a list (such as methods) just as they do for functions whose name is a symbol. You can use return-from for methods. The name of a method’s implicit block is the name of the generic function it implements. If the name of the generic function is a list, the block name is the second symbol in that list.

For a table of related items: See the section "Blocks and Exits Functions and Variables".

&body

Lambda List Keyword

This keyword is used with macros only. It is identical in function to &rest, but it informs output-formatting and editing functions that the remainder of the form is treated as a body, and should be indented accordingly.

Note that either &body or &rest, but not both, should be used in any definition.

boole op integer1 &rest more-integers

Function

This function is the generalization of logical functions such as zl:logand, zl:logior and zl:logxor. It performs bit-wise logical operations on integer arguments returning an integer which is the result of the operation.

The argument op specifies the logical operation to be performed; sixteen operations are possible. These are listed and described in the table below which also shows the truth tables for each value of op.
op can be specified by writing the name of one of the constants listed below which
represents the desired operation, or by using an integer between 0 and 15 inclusive
which controls the function that is computed. If the binary representation of op is
abcd (a is the most significant bit, d the least) then the truth table for the
Boolean operation is as follows:

<table>
<thead>
<tr>
<th>integer2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>integer1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Examples:

(boole 6 0 0) => 0 ; a=0
(boole 11 1 0) => -2 ; a=1 and b=0
(boole 2 6 9) => 9 ; a=b=d=0 c=1 therefore 1's appear only
; when integer1 is 0 and integer2 is 1

With two arguments, the result of boole is simply its second argument. At least
two arguments are required.

If boole has more than three arguments, it is associated left to right; thus,

(boole op x y z) = (boole op (boole op x y) z)
(boole boole-and 0 1 1) => 0

For the basic case of three arguments, the results of boole are shown in the table
below. This table also shows the value of bits abcd in the binary representation of
op for each of the sixteen operations. (For example, boole-clr corresponds to
#b0000, boole-and to #b0001, and so on.) As the table shows,

<table>
<thead>
<tr>
<th>op</th>
<th>Integer1</th>
<th>Integer2</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>Operation Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boole-clr</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>clear, always 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-and</td>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-andc1</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
<td>and complement of integer1 with integer2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-2</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>last of more-integers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-andc2</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>and integer1 with complement of integer2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-1</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
<td>integer1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-xor</td>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
<td>exclusive or</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-ior</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>inclusive or</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-nor</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>nor (complement of inclusive or)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boole-eqv</td>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>equivalence (exclusive nor)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**boole-c1**
1 0 1 0 complement of integer1

**boole-orc1**
1 0 1 1 or complement of integer1
with integer2

**boole-c2**
1 1 0 0 complement of integer2

**boole-orc2**
1 1 0 1 or integer1 with complement
of integer2

**boole-nand**
1 1 1 0 nand (complement of and)

**boole-set**
1 1 1 1 set, always 1

Examples:

```
(boole boole-clr 3) => 3 ;with two arguments always returns
 ;integer1
(boole boole-set 7) => 7
(boole boole-1 1 0) => 1
(boole boole-2 1 0) => 0
(boole boole-orc2 1 4) => -5
```

(Boole (if flag then boole-xor boole-ior) int1 int2)

As a matter of style the explicit logical functions such as **logand**, **logior**, and
**logxor** are usually preferred over the equivalent forms of **boole**. **boole** is useful,
however, when you want to generalize a procedure so that it can use one of several
logical operations.

For a table of related items: See the section "Functions Returning Result of Bit-
wise Logical Operations".

**boole-1**
Constant
Can be used as the first argument to the function **boole**; it specifies a bit-wise log-
ical operation that returns the first integer argument of **boole**.

**boole-2**
Constant
Can be used as the first argument to the function **boole**; it specifies a bit-wise log-
ical operation that returns the last integer argument of **boole**.

**boole-and**
Constant
Can be used as the first argument to the function **boole**; it specifies a bit-wise log-
ical *and* operation to be performed on the integer arguments of **boole**.

**boole-andc1**
Constant
Can be used as the first argument to the function `boole`; it specifies a logical operation to be performed on the integer arguments of `boole`, namely, a bit-wise logical and of the complement of the first integer argument with the next integer argument.

**boole-andc2**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a logical operation to be performed on the integer arguments of `boole`, namely, a bit-wise logical and of the first integer argument with the complement of the next integer argument.

**boole-c1**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a bit-wise logical operation that returns the complement of the first integer argument of `boole`.

**boole-c2**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a bit-wise logical operation that returns the complement of the last integer argument of `boole`.

**boole-clr**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a bit-wise logical clear operation to be performed on the integer arguments of `boole`.

**boole-eqv**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a bit-wise logical equivalence operation to be performed on the integer arguments of `boole`.

**boole-ior**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a bit-wise logical inclusive or operation to be performed on the integer arguments of `boole`.

**boole-nand**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a bit-wise logical not-and operation to be performed on the integer arguments of `boole`.

**boole-nor**

*Constant*

Can be used as the first argument to the function `boole`; it specifies a bit-wise logical not-or operation to be performed on the integer arguments of `boole`. 
**boole-ore1**

*Constant*

Can be used as the first argument to the function **boole**; it specifies a bit-wise logical operation to be performed on the integer arguments of **boole**, namely, the logical or of the complement of the first integer argument with the next integer argument.

**boole-ore2**

*Constant*

Can be used as the first argument to the function **boole**; it specifies a bit-wise logical operation to be performed on the integer arguments of **boole**, namely, the logical or of the first integer argument with the complement of the next integer argument.

**boole-set**

*Constant*

Can be used as the first argument to the function **boole**; it specifies a bit-wise logical set operation to be performed on the integer arguments of **boole**.

**boole-xor**

*Constant*

Can be used as the first argument to the function **boole**; it specifies a bit-wise logical exclusive or operation to be performed on the integer arguments of **boole**.

**both-case-p char**

*Function*

Returns t if **char** is a letter that exists in another case.

(both-case-p #\M) => T

(both-case-p #\m) => T

Returns t if **char** is an uppercase character and a lowercase character analog can be obtained by using **char-downcase**, or if **char** is a lowercase character and an uppercase character analog can be obtained by using **char-upcase**.

(both-case-p #\$) => nil

(both-case-p #\a) => t

For a table of related items, see the section "Character Predicates".

**boundp symbol**

*Function*

Returns t if the dynamic (special) variable **symbol** is bound; otherwise, it returns nil.

(defvar *alarms*)

(boundp #'*alarms*) => nil
(setq *alarms* 20)
(boundp '*alarms*) => t

See the section "Functions Relating to the Value of a Symbol".

**boundp-in-closure** closure symbol

*Function*

Returns t if symbol is bound in the environment of closure; that is, it does what `boundp` would do if you restored the value cells known about by closure. If symbol is not closed over by closure, this is just like `boundp`. See the section "Dynamic Closure-Manipulating Functions".

**boundp-in-instance** instance symbol

*Function*

Returns t if the instance variable symbol is bound in the given instance.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**break** &optional format-string &rest format-args

*Function*

Like `zl:dbg`, when evaluated, causes entry to the Debugger (a Debugger Break). However, `break` takes a format-string and format-args instead of a process.

The format-string is a user-written error message that is printed in the Debugger's Break message whenever `break` is encountered and you enter the Debugger. format-args are the `zl:format`-style arguments to `zl:format` directives in format-string.

`break` is a temporary way to insert Debugger breakpoints into your program while you are debugging it. It is not designed for permanent use in your program as a way of signalling errors. Therefore, you would use this function only for the duration of your debugging session. Continuing from `break` will not trigger any unusual recovery action.

**zl:break** &optional tag (conditional t)

*Special Form*

Enters a breakpoint loop, which is similar to a Lisp top-level loop. (zl:break tag) always enters the loop; (zl:break tag conditional) evaluates conditional and only enter the break loop if it returns non-nil. If the break loop is entered, zl:break prints out:

;Breakpoint tag; Resume to continue, Abort to quit.

The standard values for any variables are checked. If `zl:break` rebinds any of these standard variables, it warns you that it has done so. `zl:break` then enters a loop reading, evaluating, and printing forms. A difference between a break loop and the top-level loop is that when reading a form, zl:break checks for the following special cases: If the ABORT key is pressed, control is returned to the previous
break or Debugger, or to top level if there is none. If the **Resume** key is pressed, 
**zl:break** returns **nil**. If the list (**return** form) is typed, **zl:break** evaluates **form** 
and returns the result.

Inside the **zl:break** loop, the streams **zl:standard-output**, **zl:standard-input**, and 
**zl:query-io** are bound to be synonymous to **zl:terminal-io**; **zl:terminal-io** itself is 
not rebound. Several other internal system variables are bound, and you can add 
your own symbols to be bound by pushing elements onto the value of the variable 
**sys:*break-bindings***. (See the variable **sys:*break-bindings***.)

If tag is omitted, it defaults to **nil**.

There are two easy ways to write a breakpoint into your program: (**zl:break**) gets 
a read-eval-print loop, and (**zl:dbg**) gets the Debugger. (These are the programmat- 
ic equivalents of the SUSPEND and ą-SUSPEND keys on the keyboard.)

**sys:*break-bindings***

When **zl:break** is called, it binds some special variables under control of the list 
that is the value of **sys:*break-bindings***. Each element of the list is a list of two 
elements: a variable and a form that is evaluated to produce the value to bind it 
to. The bindings happen sequentially. You can push things on this list (adding to 
the front of it), but should not replace the list wholesale since several of the vari- 
able bindings on this list are essential to the operation of **zl:break**.

***break-on-warnings***

This variable controls the action of the function **warn**. If ***break-on-warnings*** is 
**nil**, **warn** prints a warning message without signalling.

If ***break-on-warnings*** is not **nil**, **warn** enters the Debugger and prints the warn- 
ing message. The default value is **nil**.

This flag is intended primarily for use when you are debugging programs that is- 
sue warnings.

Note that this flag is still supported but is considered obsolete.

For a table of related items: See the section "Condition-Checking and Signalling 
Functions and Variables".

**breakon** &optional **function** (**condition** **t**)

**Function**

With no arguments, returns a list of all functions with breakpoints set by 
**breakon**.

**breakon** sets a trace-style breakpoint for the **function**. Whenever the function 
named by **function** is called, the condition **dbg:breakon-trap** is signalled, and the 
Debugger assumes control. At this point, you can inspect the state of the Lisp en- 
vironment and the stack. Proceeding from the condition then causes the program 
to continue to run.
The first argument can be any function, so that you can trace methods and other functions not named by symbols. See the section "Function Specs".

`condition` can be used for making a conditional breakpoint. `condition` should be a Lisp form. It is evaluated when the function is called. If it returns `nil`, the function call proceeds without signalling anything. `condition` arguments from multiple calls to `breakon` accumulate and are treated as an `or` condition. Thus, when any of the forms becomes true, the breakpoint "goes off". `condition` is evaluated in the dynamic environment of the function call. You can inspect the arguments of function by looking at the variable `arglist`.

For a table of related items: See the section "Breakpoint Functions".

`dbg:bug-report-description` *condition* *stream* *nframes*  
**Generic Function**

Called by the :Mail Bug Report (c-M) command in the Debugger to print out the text that is the initial contents of the mail-sending buffer. The handler should simply print whatever information it considers appropriate onto `stream`. *nframes* is the numeric argument given to c-M. The Debugger interprets *nframes* as the number of frames from the backtrace to include in the initial mail buffer. A *nframes* of `nil` means all frames.

The compatible message for `dbg:bug-report-description` is:

`:bug-report-description`

For a table of related items: See the section "Debugger Bug Report Functions".

`dbg:bug-report-recipient-system` *condition*  
**Generic Function**

Called by the :Mail Bug Report (c-M) command in the Debugger to find the mailing list to which to send the bug report mail. The mailing list is returned as a string.

The default method (the one in the `condition` flavor) returns "lispm", and this is passed as the first argument to the `zl:bug` function.

The compatible message for `dbg:bug-report-recipient-system` is:

`:bug-report-recipient-system`

For a table of related items: See the section "Debugger Bug Report Functions".

`clos:built-in-class`  
**Class**

The class of many of the predefined classes corresponding to Common Lisp types, such as `list` and `t`.

These classes (objects whose class is `clos:built-in-class`) are provided so users can define methods that specialize on them. They do not support the full behavior of user-defined classes (whose class is `clos:standard-class`). For example, you cannot use `clos:make-instance` to create instances of these classes.
**butlast** \( x \) &optional \((n 1)\)

*Function*

Creates and returns a list with the same elements as \( x \), excepting the last element.

Examples:

\[
\begin{align*}
\text{(butlast '(a b c d))} & \Rightarrow (a b c) \\
\text{(butlast '((a b) (c d)))} & \Rightarrow ((a b)) \\
\text{(butlast '(a))} & \Rightarrow \text{nil} \\
\text{(butlast nil)} & \Rightarrow \text{nil} \\
\text{(setq a '(1 2 3 4 5 6 7))} \\
\text{(butlast a)} & \Rightarrow (1 2 3 4 5 6) \\
\text{(butlast a 4)} & \Rightarrow (1 2 3) \\
\text{a} & \Rightarrow (1 2 3 4 5 6 7)
\end{align*}
\]

The name is from the phrase "all elements but the last".

For a table of related items: See the section "Functions for Modifying Lists".

**byte** \( size \) \( position \)

*Function*

Creates a byte specifier for a byte \( size \) bits wide, \( position \) bits from the right-hand (least-significant) end of the word. The arguments \( size \) and \( position \) must be integers greater than or equal to zero.

The byte specifier so created serves as an argument to various byte manipulation functions.

Examples:

\[
\begin{align*}
\text{(ldb (byte 2 1) 9)} & \Rightarrow 0 \\
\text{(ldb (byte 3 4) #o12345)} & \Rightarrow 6 \\
\text{(setq byte-spec (byte 5 2))} \\
\text{(byte-size byte-spec)} & \Rightarrow 5 \\
\text{(byte-position byte-spec)} & \Rightarrow 2
\end{align*}
\]

For a table of related items: See the section "Summary of Byte Manipulation Functions".

**byte-position** \( bytespec \)

*Function*

Extracts the position field of \( bytespec \).

\( bytespec \) is built using function \( byte \) with bit \( size \) and \( position \) arguments.

Example:

\[
\text{(byte-position (byte 3 4))} \Rightarrow 4
\]

For a table of related items: See the section "Summary of Byte Manipulation Functions".

**byte-size** \( bytespec \)

*Function*

Extracts the size field of \( bytespec \).
bytespec is built using function byte with bit size and position arguments.

Example:

```
(byte-size (byte 3 4)) => 3
```

For a table of related items: See the section "Summary of Byte Manipulation Functions".

**Function**

```
caaaar x

(caaaar x) is the same as (car (car (car (car x))))
```

**Function**

```
caaadr x

(caaadr x) is the same as (car (car (car (cdr x))))
```

**Function**

```
caaar x

(caaar x) is the same as (car (car (car x)))
```

**Function**

```
caadar x

(caadar x) is the same as (car (car (cdr (car x))))
```

**Function**

```
caaddr x

(caaddr x) is the same as (car (car (cdr (cdr x))))
```

**Function**

```
caadr x

(caadr x) is the same as (car (car (cdr x)))
```

**Function**

```
caar x

(caar x) is the same as (car (car x))
```

**Function**

```
cadaar x

(cadaar x) is the same as (car (cdr (car (car x))))
```

**Function**

```
cadadr x

(cadadr x) is the same as (car (cdr (car (cdr x))))
```
(cadadr x) is the same as (car (cdr (car (cdr x))))

**cadar x**

(cadar x) is the same as (car (cdr (car x)))

**caddar x**

(caddar x) is the same as (car (cdr (cdr (car x))))

**cadddr x**

(cadddr x) is the same as (car (cdr (cdr (cdr x))))

**caddr x**

(caddr x) is the same as (car (cdr (cdr x)))

**cadr x**

(cadr x) is the same as (car (cdr x))

**call-arguments-limit**

A positive integer that is the upper exclusive bound on the number of arguments that can be passed to a function. The current value is 128 for 3600-series machines, 50 for Ivory-based machines, and 256 for CLOE.

For example, let’s assume that we have two functions, **process-elements-pairwise** and **process-elements-atonce**. The first takes the elements of an array and operates on them by repeatedly calling a subordinate function of two variables. The second function atonce calls a subordinate function that takes each element of the array as arguments. Then we might use the following code to call the appropriate function:

```
(if (> (array-total-size array) call-arguments-limit)
  (process-elements-pairwise array)
  (process-elements-atonce array))
```

**flavor:call-component-method function-spec &key apply arglist**

Function

Produces a form that calls function-spec, which must be the function-spec for a component method. If no keyword arguments are given to flavor:call-component-
method, the method receives the same arguments that the generic function received. That is, the first argument to the generic function is bound to self inside the method, and succeeding arguments are bound to the argument list specified with defmethod. Additional internal arguments are passed to the method, but the user never needs to be concerned about these.

arglist is a list of forms to be evaluated to supply the arguments to the method, instead of simply passing through the arguments to the generic function.

When arglist and apply are both supplied, :apply should be followed by t or nil. If :apply t is supplied, the method is called with apply instead of funcall. :apply nil causes the method to be called with funcall.

When arglist is not supplied, the value following :apply is the argument that should be given to apply when the method is called. (Certain internal arguments are also included in the apply form.) For example:

(flavor:call-component-method function-spec :apply list)

Results in:

(apply #'function-spec internal arguments list)

In other words, the following two forms have the same effect:

(flavor:call-component-method function-spec :apply list)
(flavor:call-component-method function-spec :arglist (list list)
 :apply t)

If function-spec is nil, flavor:call-component-method produces a form that returns nil when evaluated.

For examples, see the section "Examples of define-method-combination".

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

flavor:call-component-methods function-spec-list &key (operator 'progn) Function

Produces a form that invokes the function or special form named operator. Each argument or subform is a call to one of the methods in function-spec-list. operator defaults to progn.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

clos:call-method method &optional next-method-list Macro

Used within effective method forms (forms returned by the body of clos:define-method-combination) to call a method. The macro clos:call-method calls the method and supplies it with the arguments that were supplied to generic function.

The next-method-list argument to clos:call-method defines the "next method" for clos:call-next-method and clos:next-method-p. That is, if clos:call-next-method is
called within the method, the first method in next-method-list will be called; if
clos:call-next-method is called within that method, the second method in next-
method-list will be called, and so on.

method A method object, or a list such as (clos:make-method form).
Such a list specifies a method object whose method function
has a body that is the given Lisp form.

next-method-list A list of method objects. Each element is either a method ob-
ject or a list such as (clos:make-method form), as described
above.

clos:call-method returns the value or values returned by the method.

When clos:call-method is called and the next-method-list argument is unsupplied,
it means that semantically there is no such thing as a "next method"; for example,
this is true for before-methods and after-methods in clos:standard method combi-
nation. Thus, when the next-method-list is unsupplied, clos:call-next-method is not
allowed inside the method, and the behavior of clos:next-method-p is undefined. If
the next-method-list argument is supplied as nil, and the method uses clos:call-
next-method, then clos:no-next-method is called.

clos:call-next-method &rest args

Function

Used within a method body to call the "next method". clos:call-next-method re-
turns the value or values returned by the method it calls.

args Arguments to be passed to the next method. If any args are
provided, the following condition must hold: the ordered set of
methods applicable for args must be the same as the ordered
set of methods applicable for the arguments that were passed
to the generic function. If this requirement is not satisfied, an
error is signaled.

If no args are provided, clos:call-next-method passes the
method's original arguments on to the next method.

The method-combination type in use determines which kinds of methods can use
clos:call-next-method, and defines the meaning of "next method". The
clos:standard method-combination type supports clos:call-next-method in around-
methods and primary methods, but not in before-methods or after-methods. It de-
defines the next method as follows:

• If clos:call-next-method is called in an around-method, the next method is the
  next most specific around-method, if one is applicable.

• If clos:call-next-method is called in the least specific applicable around-method,
  the next method consists of the following:

  ° All the before-methods in most-specific-first order.
° The most specific primary method. If clos:call-next-method is called in the primary method, then the next method is the next most specific primary method.

° All the after-methods in most-specific-last order.

If clos:call-next-method is called and there is no next method, then clos:no-next-method is called. The default method for clos:no-next-method signals an error.

If clos:call-next-method is called with arguments but omits optional arguments, the next method called defaults those arguments.

Clos:call-next-method has lexical scope and indefinite extent.

You can use clos:next-method-p to test whether the next method exists.

If clos:call-next-method is called in a method that does not support it, an error is signaled. The method-combination type in use controls which kinds of methods support clos:call-next-method.

car x  

Function

Returns the head (car) of list or cons x. Example:

(car '(a b c)) => a
(setq a '(first second third)) =>
(first second third) =>
(car a) =>
first
(car (cdr a)) =>
second

Officially car is applicable only to conses and locatives. However, as a matter of convenience, car of nil returns nil.

For a table of related items: See the section "Functions for Extracting from Lists".

zl:car-location cons  

Function

Returns a locative pointer to the cell containing the car of cons.

Note: there is no cdr-location function; since the cons itself can be used as a locative to its cdr.

For a table of related items: See the section "Functions for Finding Information About Lists and Conses".

case test-object &body clauses  

Special Form

This is a conditional that chooses one of its clauses to execute by comparing a value to various constants. The constants can be any object.
Its form is as follows:

(case test-object
  (keylist consequent consequent ...)
  (keylist consequent consequent ...)
  (keylist consequent consequent ...)
  ...
)

Structurally case is much like cond, and it behaves like cond in selecting one clause and then executing all consequents of that clause. However, case differs in the mechanism of clause selection.

The first thing case does is to evaluate test-object, to produce an object called the key object. Then case considers each of the clauses in turn. If key is eql to any item in the test list of a clause, case evaluates the consequents of that clause as an implicit progn.

If no clause is satisfied, case returns nil.

case returns the value of the last consequent of the clause evaluated, or nil if there are no consequents to that clause.

The keys in the clauses are not evaluated; they must be literal key values. It is an error for the same key to appear in more than one clause. The order of the clauses does not affect the behavior of the case construct.

Instead of a test, one can write one of the symbols t and otherwise. A clause with such a symbol always succeeds and must be the last clause; this is an exception to the order-independence of clauses.

If there is only one key value for a clause, that key value can be written in place of a list of that key, provided that no ambiguity results. Such a "singleton key" can not be nil (which is confusable with (), a list of no keys), t, otherwise, or a cons.

Examples:

(let ((num 69))
  (case num
    ((1 2) "math...ack")
    ((3 4) "great now we can count"))) => NIL

(let ((num 3))
  (case num
    ((1 2) "one two")
    ((3 4 5 6) (princ "numbers") (princ " three") (fresh-line))
    (t "not today"))) => numbers three

T

(let ((object-one 'candy))
  (case object-one
    (apple (setq class 'health) "weekdays")
    (candy (setq class 'junk) "weekends")
    (otherwise (setq class 'unknown) "all week long"))) => "weekends"

class => JUNK
For a table of related items: See the section "Conditional Functions".

```
(defun print-field (object)
  (when (consp object)
    (case (list-length object)
      (1 (print (car object)))
      ((2 3 4 5) (print (cadr object)))
      (otherwise (print "too large to print")))))
```

**zl:caseq** *test-object* &body *clauses*  

*Special Form*

Provided for Maclisp compatibility; it is exactly the same as **zl:selectq**. This is not perfectly compatible with Maclisp, because **zl:selectq** accepts **otherwise** as well as t where **zl:caseq** would not accept **otherwise**, and because Maclisp accepts a more limited set of keys then **zl:selectq** does. Maclisp programs that use **zl:caseq** work correctly as long as they do not use the symbol **otherwise** as the key.

Examples:

```
(let (( a 'big-bang))
  (caseq a
    (light "day")
    (dark "night")) => NIL

(setq a 3) => 3
(caseq a
  (1 "one")
  (2 "two")
  (t "not one or two")) => "not one or two"

(let (( a 'big-bang))
  (caseq a
    (light "day")
    (dark "night")
    (otherwise "night and day"))) => "night and day"
```

For a table of related items: See the section "Conditional Functions".

**catch** *tag* &body *body*  

*Special Form*

Provides an environment for evaluating its argument *forms* as an implicit **progn** with dynamic exit capability **throw**. Although **throw** need not be in the lexical scope of **catch**, it must be in the dynamic scope.

Used with **throw** for nonlocal exits. **catch** first evaluates *tag* to obtain an object that is the "tag" of the catch. Then the *body* forms are evaluated in sequence, and **catch** returns the (possibly multiple) values of the last form in the body.

However, a **throw** (or in Genera, a ***throw** ) form might be evaluated during the evaluation of one of the forms in *body*. In that case, if the throw "tag" is **eq** to the catch "tag" and if this **catch** is the innermost **catch** with that tag, the evaluation
of the body is immediately aborted, and catch returns values specified by the throw or zl:*throw form.

If the catch exits abnormally because of a throw form, it returns the (possibly multiple) values that result from evaluating throw’s second subform. If the catch exits abnormally because of a zl:*throw form, it returns two values: the first is the result of evaluating zl:*throw’s second subform, and the second is the result of evaluating zl:*throw’s first subform (the tag thrown to).

(catch 'foo form) catches a (throw 'foo form) but not a (throw 'bar form). It is an error if throw is done when no suitable catch exists.

The scope of the tags is dynamic. That is, the throw does not have to be lexically within the catch form; it is possible to throw out of a function that is called from inside a catch form.

For example:

(catch 'done
  (ask-database <pattern>
    #'(lambda (x) (when (nice-p x)
       (throw 'done x)))))

The throw to 'done returns x, the pattern searched for in the database. The second example that follows acts as a somewhat extended example of a tiny parser.

(catch 'foo (list 'a (catch 'bar (throw 'foo 'b)))) → B

(defvar *input-buffer* nil)

(defun parse (*input-buffer*)
  (catch 'parse-error
    (list 's (parse-np) (parse-vp))))

(defun parse-np (&aux (item (pop *input-buffer*)))
  (if (member item '(a an the))
    '(np (det item) (n , (pop *input-buffer*)))
    (throw 'parse-error
      (format t "Problem with ~A in noun phrase.~" item))))

(defun parse-vp (&aux (item (pop *input-buffer*)))
  (if (member item '(eats sleeps runs))
    '(vp (v item))
    (throw 'parse-error
      (format t "Problem with ~A in verb phrase.~" item))))

(parse '(a man eats)) => (S (NP (DET A) (N MAN)) (VP (V EATS)))

(parse '(a man walks)) => NIL
  prints: Problem with WALKS in verb phrase.

For a table of related items, see the section "Nonlocal Exit Functions".
zl:*catch tag &body body

Special Form

An obsolete version of catch that is supported for compatibility with Maclisp. It is equivalent to catch except that if zl:*catch exits normally, it returns only two values: the first is the result of evaluating the last form in the body, and the second is nil. If zl:*catch exits abnormally, it returns the same values as catch when catch exits abnormally: that is, the returned values depend on whether the exit results from a throw or a zl:*throw. See the special form catch.

For a table of related items, see the section "Nonlocal Exit Functions".

catch-error form &optional (printflag t)

Macro

Evaluates form, trapping all errors.

form can be any Lisp expression.

printflag controls the printing or suppression of an error message by catch-error.

If an error occurs during the evaluation of form, catch-error prints an error message if the value of printflag is not nil. The default value of printflag is t.

catch-error returns two values: if form evaluated without error, the value of form and nil are returned. If an error did occur during the evaluation of form, t is returned.

Only the first value of form is returned if it was successfully evaluated.

catch-error-restart (flavors description &rest args) &body body

Special Form

Establishes a restart handler for flavors and then evaluates the body. If the handler is not invoked, catch-error-restart returns the values produced by the last form in the body, and the restart handler disappears. If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment of the catch-error-restart form. In this case, catch-error-restart also returns nil as its first value and something other than nil as its second value. Its format is:

(catch-error-restart (flavors description)
  form-1
  form-2
  ...)

flavors is either a condition or a list of conditions that can be handled. description is a list of arguments to be passed to format to construct a meaningful description of what would happen if the user were to invoke the handler. The Debugger uses these values to create a message explaining the intent of the restart handler.

The conditional variant of catch-error-restart is the form:
catch-error-restart-if

For a table of related items: See the section "Restart Functions".

catch-error-restart-if cond (flavors description &rest args) &body body

Special Form

Establishes its restart handler conditionally. In all other respects, it is the same as catch-error-restart. Its format is:

(catch-error-restart-if cond
  (flavors description)
  form-1
  form-2
  ...
)

catch-error-restart-if first evaluates cond. If the result is nil, it evaluates body as if it were a progn but does not establish any handlers. If the result is not nil, it continues just like catch-error-restart, establishing the handlers and executing body.

For a table of related items: See the section "Restart Functions".

ccase object &body body

Special Form

The name of this function stands for "continuable exhaustive case".

Structurally ccase is much like case, and it behaves like case in selecting one clause and then executing all consequents of that clause. However, ccase does not permit an explicit otherwise or t clause. The form of ccase is as follows:

(ccase key-form
  (test consequent consequent ...
   )
  (test consequent consequent ...
   )
  (test consequent consequent ...
   )
  ...
)

object (which serves as the key-form) must be a generalized variable reference acceptable to setf.

The first thing ccase does is to evaluate key-form, to produce an object called the key object.

Then ccase considers each of the clauses in turn. If key is eql to any item in the test list of a clause, ccase evaluates the consequents of that clause as an implicit progn.

ccase returns the value of the last consequent of the clause evaluated, or nil if there are no consequents to that clause.

The test lists in the clauses are not evaluated; literal key values must appear in test. It is an error for the same key value to appear in more than one clause. The order of the clauses does not affect the behavior of the ccase construct.
If there is only one key value for a clause, that key value can be written in place of a list of that key, provided that no ambiguity results. Such a "singleton key" can not be \texttt{nil} (which is confusable with \texttt{()}, a list of no keys), \texttt{t}, \texttt{otherwise}, or a cons.

If no clause is satisfied, \texttt{ccase} uses an implicit \texttt{otherwise} clause to signal an error with a message constructed from the clauses. To continue from this error supply a new value for \texttt{object} argument, causing \texttt{ccase} to store that value and restart the clause tests. Subforms of \texttt{object} can be evaluated multiple times.

Examples:

\begin{verbatim}
(let ((num 24))
  (ccase num
    ((1 2 3) "integer less then 4")
    ((4 5 6) "integer greater than 3")
  ) =>
Error: The value of NUM is SI:*EVAL, 24, was of the wrong type.
  The function expected one of 1, 2, 3, 4, 5, or 6.

SI:*EVAL:
  Arg 0 (SYS:FORM): (DBG:CHECK-TYPE-1 'NUM NUM '#)
  Arg 1 (SI:ENV): ((# #) NIL (#) (#) ...)
  --defaulted args:--
  Arg 2 (SI:HOOK): NIL
s-A, <RESUME>: Supply a replacement value to be stored into NUM
s-B, <ABORT>: Return to Lisp Top Level in dynamic Lisp Listener 1
→ Supply a replacement value to be stored into NUM:
  4
  "integer greater than 3"

(let ((num 3))
  (ccase num
    ((1 2) "one two")
    ((3 4 5 6) (princ "numbers") (princ " three") (terpri) )
    (t "not today")
  ) => numbers three
T
(let ((Dwarf 'Sleepy))
  (ccase Dwarf
    ((Grumpy Dopey) (setq class "confused"))
    ((Bilbo Frodo) (setq class "Hobbits not Dwarfs"))
    (otherwise (setq class 'unknown) "talk to Snow White")
  ) => "talk to Snow White"
  class => UNKNOWN
)\end{verbatim}

For a table of related items: See the section "Conditional Functions".

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".
(cdaaar \textit{x}) \textit{is the same as} (cdr (car (car (car \textit{x}))))

\textbf{Function}

cdaadr \textit{x}

(cdaadr \textit{x}) \textit{is the same as} (cdr (car (car (cdr \textit{x}))))

\textbf{Function}

cdaar \textit{x}

(cdaar \textit{x}) \textit{is the same as} (cdr (car (car \textit{x})))

\textbf{Function}

cdadar \textit{x}

(cdadar \textit{x}) \textit{is the same as} (cdr (car (cdr (car \textit{x}))))

\textbf{Function}

cdaddr \textit{x}

(cdaddr \textit{x}) \textit{is the same as} (cdr (car (cdr (cdr \textit{x}))))

\textbf{Function}

cdadr \textit{x}

(cdadr \textit{x}) \textit{is the same as} (cdr (car (cdr \textit{x})))

\textbf{Function}

cdar \textit{x}

(cdar \textit{x}) \textit{is the same as} (cdr (car \textit{x}))

\textbf{Function}

cddaar \textit{x}

(cddaar \textit{x}) \textit{is the same as} (cdr (cdr (car (car \textit{x}))))

\textbf{Function}

cddadr \textit{x}

(cddadr \textit{x}) \textit{is the same as} (cdr (cdr (car (cdr \textit{x}))))

\textbf{Function}

cddar \textit{x}

(cddar \textit{x}) \textit{is the same as} (cdr (cdr (car \textit{x})))

\textbf{Function}

cdddar \textit{x}

(cdddar \textit{x}) \textit{is the same as} (cdr (cdr (cdr (car \textit{x}))))
(cdddar x) is the same as (cdr (cdr (cdr (car x))))

Function

(cddddd r x)

(cddddd r x) is the same as (cdr (cdr (cdr (cdr x))))

Function

(cdddr x)

(cdddr x) is the same as (cdr (cdr (cdr x)))

Function

(cddr x)

(cddr x) is the same as (cdr (cdr x))

Function

(cdr x)

Returns the tail (cdr) of list or cons x. Example:

(cdr '(a b c)) => (b c)
(setq a '(1 (first second third) c d) =>
=> (1 (FIRST SECOND THIRD) C D))
(setq b (cdr a))
=> ((FIRST SECOND THIRD) C D)
(cdr (car b))
=> (SECOND THIRD)

Officially cdr is applicable only to conses and locatives. However, as a matter of convenience, cdr of nil returns nil.

For a table of related items: See the section "Functions for Extracting from Lists".

ceiling number &optional (divisor 1)

Divides number by divisor, and truncates the result toward positive infinity. The truncated result and the remainder are the returned values.

number and divisor must each be a noncomplex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are Q and R, (+ (* q divisor) R) equals number. If divisor is 1, then Q and R add up to number. If divisor is 1 and number is an integer, then the returned values are number and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, then the second returned value is always a number of the same type as the argument.
Examples:

(ceiling 5) => 5 and 0
(ceiling -5) => -5 and 0
(ceiling 5.2) => 6 and -0.8000002
(ceiling -5.2) => -5 and -0.19999981
(ceiling 5.8) => 6 and -0.19999981
(ceiling -5.8) => -5 and -0.8000002
(ceiling 5 3) => 2 and -1
(ceiling -5 3) => -1 and -2
(ceiling 5 4) => 2 and -3
(ceiling -5 4) => -1 and -1
(ceiling 5.2 3) => 2 and -0.8000002
(ceiling -5.2 3) => -1 and -2.1999998
(ceiling 5.2 4) => 2 and -2.8000002
(ceiling -5.2 4) => -1 and -1.1999998
(ceiling 5.8 3) => 2 and -0.19999981
(ceiling -5.8 3) => -1 and -2.8000002
(ceiling 5.8 4) => 2 and -2.1999998
(ceiling -5.8 4) => -1 and -1.8000002

For a table of related items: See the section "Functions that Divide and Convert Quotient to Integer".

\texttt{cerror\ optional-condition-name\ continue-format-string\ error-format-string\ &rest\ \textit{args}}

\textbf{Function}

Signals proceedable (continuable) errors. Like \texttt{error}, it signals an error and enters the Debugger. However, \texttt{cerror} allows the user to continue program execution from the debugger after resolving the error.

If the program is continued after encountering the error, \texttt{cerror} returns \texttt{nil}. The code following the call to \texttt{cerror} is then executed. This code should correct the problem, perhaps by accepting a new value from the user if a variable was invalid.

If the code that corrects the problem interacts with the program’s use and might possibly be misleading, it should make sure the error has really been corrected before continuing. One way to do this is to put the call to \texttt{cerror} and the correction code in a loop, checking each time to see if the error has been corrected before terminating the loop.

\textbf{Compatibility Note:} \textit{Optional-condition-name} is a Symbolics Common Lisp extension, which allows you to specify a particular flavor error.

The \textit{continue-format-string} argument, like the \textit{error-format-string} argument, is given as a control string to \texttt{format} along with \textit{args} to construct a message string. The error message string is used in the same way that \texttt{error} uses it. The continue message string should describe the effect of continuing. The message is displayed as an aid to the user in deciding whether and how to continue. For example, it might be used by an interactive debugger as part of the documentation of its "continue" command.
The content of the continue message should adhere to the rules of style for error messages.

In complex cases where the error-format-string uses some of the args and the continue-format-string uses others, it may be necessary to use the format directives ~* and ~/j141 to skip over unwanted arguments in one or both of the format control strings.

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

**clos:change-class** instance new-class  

Changes the class of the existing instance to new-class, and returns the modified instance. The modified instance is eq to the original instance.

- **instance** The instance whose class is to be changed.
- **new-class** The desired class of the instance. This can be the name of a class or a class object.

**clos:change-class** modifies the structure of the instance to be correct for the new class. It does the following:

- Adds local slots: For any local slot defined by the new class that is not defined by the previous class, that slot is added to the instance.

- Deletes local slots: For any local slot defined by the previous class that is not defined by the new class, that slot is deleted from the instance.

- Retains the values of local slots: For any local slot defined by both the previous and the new class, the instance retains the value of that slot. If the slot was unbound, it remains unbound.

- Retains the values of slots defined as shared in the previous class and local in the new class.

- Replaces the values of slots defined as local in the previous class and shared in the new class; the instance now "sees" the value of the shared slot.

Next, **clos:change-class** initializes newly added slots according to their initforms by calling **clos:update-instance-for-different-class** with two arguments: a copy of the instance before its class was changed (which enables methods to access the slot values), and the modified instance. **clos:change-class** does not provide any initialization arguments in its call to **clos:update-instance-for-different-class**.

You can customize the behavior of this step by defining an after-method for **clos:update-instance-for-different-class**.

See the section "Changing the Class of a CLOS Instance".
change-instance-flavor instance new-flavor Function

Changes the flavor of an instance to another flavor. The result is a modified instance, which is eq to the original.

For those instance variables in common (contained in the definition of the old flavor and the new flavor), the values of the instance variables remain the same when the instance is changed to the new format. New instance variables (defined by the new flavor but not the old flavor) are initialized according to any defaults contained in the definition of the new flavor.

Instance variables contained by the old flavor but not the new flavor are no longer part of the instance, and cannot be accessed once the instance is changed to the new format.

Instance variables are compared with eq of their names; if they have the same name and are defined by both the old flavor (or any of its component flavors) and the new flavor (or any of its component flavors), they are considered to be “in common”.

If you need to specify a different treatment of instance variables when the instance is changed to the new flavor, you can write code to be executed at the time that the instance is changed. See the generic function flavor:transform-instance.

Note: There are two possible problems that might occur if you use change-instance-flavor while a process (either the current process or some other process) is executing inside of a method. The first problem is that the method continues to execute until completion even if it is now the “wrong” method. That is, the new flavor of the instance might require a different method to be executed to handle the generic function. The Flavors system cannot undo the effects of executing the wrong method and cause the right method to be executed instead.

The second problem is due to the fact that change-instance-flavor might change the order of storage of the instance variables. A method usually commits itself to a particular order at the time the generic function is called. If the order is changed after the generic function is called, the method might access the wrong memory location when trying to access an instance variable. The usual symptom is an access to a different instance variable of the same instance or an error “Trap: The word #<DTP-HEADER-I nnnn> was read from location nnnn”. If the garbage collector has moved objects around in memory, it is possible to access an arbitrary location outside of the instance.

When a flavor is redefined, the implicit change-instance-flavor that happens never causes accesses to the wrong instance variable or to arbitrary locations outside the instance. But redefining a flavor while methods are executing might leave those methods as no longer valid for the flavor.

We recommend that you do not use change-instance-flavor of self inside a method. If you cannot avoid it, then make sure that the old and new flavors have the same instance variables and inherit them from the same components. You can do this by using mixins that do not define any instance variables of their own, and using change-instance-flavor only to change which of these mixins are included. This prevents the problem of accessing the wrong location for an instance variable,
but it cannot prevent a running method from continuing to execute even if it is
now the wrong method.

A more complex solution is to make sure that all instance variables accessed after
the change-instance-flavor by methods that were called before the change-
instance-flavor are ordered (by using the :ordered-instance-variables option to
defflavor), or are inherited from common components by both the old and new
flavors. The old and new flavors should differ only in components more specific
than the flavors providing the variables.

For a summary of all functions, macros, special forms, and variables related to
Flavors, see the section “Summary of Flavor Functions and Variables”.

:change-properties error-p &rest properties

Changes the file properties of the file open on this stream. You should not use

If the error-p argument is t, a Lisp error is signalled. If error-p is nil and an error
occurs, the error object is returned.

char string index

Returns the character at position index of string. The count is from zero. The
character is returned as a character object; it will necessarily satisfy the predicate
string-char-p.

string must be a string.

index must be a non-negative integer less than the length of string.

Note that the array-specific function aref, and the general sequence function elt
also work on strings.

To destructively replace a character within a string, use char in conjunction with
the function setf.

Examples:

(char "a string" 1) => \Space
(string-char-p (char "a string" 3)) => T

(char (make-array 4 :element-type 'character
       :initial-element \y) 3) => \y
(string-char-p (char (make-array 4 :element-type 'character
                        :initial-element \.) 2)) => T

(char (make-array 4 :element-type 'character
       :initial-element \.
       :fill-pointer 2) 1) => \.

(defvar a-string
  (make-array 10
    :element-type 'string-char
    :fill-pointer t
    :initial-element #\a))
=> "aaaaaaaaaa"

(char a-string 0) => #\a

(setf (char a-string 1) #\b) => #\b

a-string => "abaaaaaaaa"

(char a-string 1) => #\b

Because a-string is not a simple string, char rather than schar is used to access
elements of the string.

For a table of related items: See the section "String Access and Information".

char# char &rest chars

This comparison predicate compares characters exactly, depending on all fields in-
cluding code, bits, character style, and alphabetic case. If all of the arguments are
equal, nil is returned, otherwise t.

(char/= #\A #\A #\A) => NIL
(char/= #\A #\B #\C) => T

char# can be used in place of user::char//=.

For a table of related items, see the section "Character Comparisons Affected by
Case and Style".

char<= char &rest chars

This predicate compares characters exactly, depending on all fields including code,
bits, character style, and alphabetic case. If each of the arguments is equal to or
less than the next, t is returned, otherwise nil.

(char<= #\A #\B #\C) => T
(char<= #\C #\B #\A) => NIL
(char<= #\A #\A #\A) => T

char<= can be used instead of char<=.

char>= char &rest chars

This comparison predicate compares characters exactly, depending on all fields in-
cluding code, bits, character style, and alphabetic case. If each of the arguments is
equal to or greater than the next, t is returned, otherwise nil.
(char>= #\C #\B #\A) => T
(char>= #\A #\A) => T
(char>= #\A #\B #\C) => NIL

char≥ can be used instead of char>=.

For a table of related items, see the section "Character Comparisons Affected by Case and Style".

For a table of related items, see the section "Character Comparisons Affected by Case and Style".

char/= char &rest chars

Function

This comparison predicate compares characters exactly, depending on all fields including code, bits, character style, and alphabetic case. If all of the arguments are equal, nil is returned, otherwise t.

(char/= #\A #\A #\A) => NIL
(char/= #\A #\B #\C) => T

char# can be used in place of user::char///=.

For a table of related items, see the section "Character Comparisons Affected by Case and Style".

char< char &rest chars

Function

This comparison predicate compares characters exactly, depending on all fields including code, bits, character style, and alphabetic case. If all of the arguments are ordered from smallest to largest, t is returned, otherwise nil.

(char< #\A #\B #\C) => T
(char< #\A #\A) => NIL
(char< #\A #\C #\B) => NIL

For a table of related items, see the section "Character Comparisons Affected by Case and Style".

char<= char &rest chars

Function

This predicate compares characters exactly, depending on all fields including code, bits, character style, and alphabetic case. If each of the arguments is equal to or less than the next, t is returned, otherwise nil.

(char<= #\A #\B #\C) => T
(char<= #\C #\B #\A) => NIL
(char<= #\A #\A) => T

char≤ can be used instead of char<=.

char= char &rest chars

Function
This comparison predicate compares characters exactly, depending on all fields including code, bits, character style, and alphabetic case. If all of the arguments are equal, \texttt{t} is returned, otherwise \texttt{nil}.

\[
\begin{align*}
\text{(char=} & \text{\#A \#A \#A)} \Rightarrow \text{t} \\
\text{(char=} & \text{\#A \#B \#C)} \Rightarrow \text{nil}
\end{align*}
\]

For a table of related items, see the section "Character Comparisons Affected by Case and Style".

\textbf{char> char &rest chars \hspace{1cm} Function}

This comparison predicate compares characters exactly, depending on all fields including code, bits, character style, and alphabetic case. If all of the arguments are ordered from largest to smallest, \texttt{t} is returned, otherwise \texttt{nil}.

\[
\begin{align*}
\text{(char>} & \text{\#C \#B \#A)} \Rightarrow \text{t} \\
\text{(char>} & \text{\#A \#A)} \Rightarrow \text{nil} \\
\text{(char>} & \text{\#A \#B \#C)} \Rightarrow \text{nil}
\end{align*}
\]

For a table of related items, see the section "Character Comparisons Affected by Case and Style".

\textbf{char>= char &rest chars \hspace{1cm} Function}

This comparison predicate compares characters exactly, depending on all fields including code, bits, character style, and alphabetic case. If each of the arguments is equal to or greater than the next, \texttt{t} is returned, otherwise \texttt{nil}.

\[
\begin{align*}
\text{(char>=} & \text{\#C \#B \#A)} \Rightarrow \text{t} \\
\text{(char>=} & \text{\#A \#A)} \Rightarrow \text{t} \\
\text{(char>=} & \text{\#A \#B \#C)} \Rightarrow \text{nil}
\end{align*}
\]

\texttt{char\geq} can be used instead of \texttt{char>=}.

For a table of related items, see the section "Character Comparisons Affected by Case and Style".

\textbf{character \hspace{1cm} Type Specifier}

\texttt{character} is the type specifier symbol for the predefined Lisp character data type.

The types \texttt{character, cons, symbol, and array} are pairwise disjoint.

The type \texttt{character} is a supertype of the type \texttt{string-char}.

Examples:
(typep #\@ 'character) => T
(zl:typep #\") => :CHARACTER
(characterp #\A) => T
(characterp (character "l")) => T
(sys:type-arglist 'character) => NIL and T

See the section “Data Types and Type Specifiers”. See the section “Characters”.

character x

Function

Coerces x to a single character. If x is a character, it is returned. If x is a string, an array, or a symbol, an error is returned. If x is a number, the number is converted to a character using int-char. See the section “The Character Set”.

For a table of related items, see the section “Character Conversions”.

characterp object

Function

Returns t if object is a character object. See the section “Type Specifiers and Type Hierarchy for Characters”.

(setq foo '(#\c 44 "h"))
(characterp foo) => nil
(characterp (car foo)) => t
(characterp (cadr foo)) => nil
(characterp (caddr foo)) => nil

Note in the previous example that "h" is not a character, but a string.

(characterp (aref "h" 0)) => t

For a table of related items: See the section “Character Predicates”.

:characters

Message

Returns t if the stream is a character stream, nil if it is a binary stream.

dbg:*character-style-for-bug-mail-prologue*

Variable

Creates the bug-report banner inserted into the text of bug messages, enabling you to choose the font. The default is NIL.NIL.TINY, specifying a small font for the bug-report banner.

To display a bug-report banner in a small font you can type the following:

(setq dbg:*character-style-for-bug-mail-prologue*
   (si:character-style-for-device-font 'fonts:quantum si:*b&w-screen*))

To display a bug-report banner in a large font you can type the following:

(setq dbg:*character-style-for-bug-mail-prologue*
   (si:parse-character-style ‘(nil nil :huge)))
You can also type the following to specify a particular font:

```
(setq dbg:*character-style-for-bug-mail-prologue* '(nil nil :huge))
```

**char-bit** char name  
*Function*  
Returns t if the bit specified by name is set in char; otherwise it returns nil. name can be :control, :meta, :super, or :hyper. You can use setf on char-bit access-form name.

```
(char-bit #\c-A :control) => T
(char-bit #\h-c-A :hyper) => T
(char-bit #\h-c-A :meta) => NIL
(setq char #\D)
(char-bit (set-char-bit char :control t) :control) => t
(char-bit char :control) => nil
```

For a table of related items, see the section "Character Fields".

**char-bits** char  
*Function*  
Returns the bits field of char. You can use setf on (char-bits access-form).

```
(char-bits #\c-A) => 1
(char-bits #\h-c-A) => 9
(char-bits #\m-c-A) => 3
(char-bits #\Control-D) => 1
(char-bits #\D) => 0
```

For a table of related items, see the section "Character Fields".

**char-bits-limit**  
*Constant*  
The value of char-bits-limit is a non-negative integer that is the upper limit for the value in the bits field. Its value is 16.

```
(if (= char-bits-limit 1)
  (setq *no-bits* t)
  (setq *no-bits* nil))
```

For a table of related items: See the section "Character Attribute Constants".

**char-code** char  
*Function*  
Returns the code field of char.

```
(char-code #\A) => 65
(char-code #\&) => 38
```

For a table of related items, see the section "Character Fields".
char-code-limit

The value of **char-code-limit** is a non-negative integer that is the upper limit for the number of character codes that can be used. Its value is 65536.

(let ((intnum (read stream))
      (bits (read stream)))
  (if (> intnum char-code-limit)
      (error "Cannot make ~A a character code" intnum)
      (code-char intnum bits)))

For a table of related items: See the section "Character Attribute Constants".

char-control-bit

The value of **char-control-bit** is the weight of the control bit, which is 1.

For a table of related items: See the section "Character Bit Constants".

char-downcase char

If **char** is an uppercase alphabetic character in the standard character set, **char-downcase** returns its lowercase form; otherwise, it returns **char**. If character style information is present it is preserved. In no case will the font or bits attribute values differ from those of **char**.

(char-downcase #\A) => #\a
(char-downcase #\A) => #\a
(char-downcase #\3) => #\3
(char-downcase #\a) => #\a

For a table of related items, see the section "Character Conversions".

char-equal char &rest chars

This is the primitive for comparing characters for equality; many of the string functions call it. **char** and **chars** must be characters; they cannot be integers. **char-equal** compares code and bits, ignores case and character style, and returns **t** if the characters are equal. Otherwise it returns **nil**.

(char-equal #\A #\A) => T
(char-equal #\A #\Control-A) => NIL
(char-equal #\A #\B #\A) => NIL

Compatibility Note: Common Lisp specifies that **char-equal** should ignore bits. This difference is incompatible. Under CLOE, **lisp:char-equal** ignores the bits attribute of the character arguments.

For a table of related items, see the section "Character Comparisons Ignoring Case and Style".
char-fat-p char

Function

Returns t if char is a fat character, otherwise nil. char must be a character object. A character that contains non-zero bits or style information is called a fat character. See the section "Type Specifiers and Type Hierarchy for Characters".

(char-fat-p #\A) => NIL
(char-fat-p #\c-A) => T
(char-fat-p (make-character #\A :style '(nil :bold nil))) => T

For a table of related items: See the section "Character Predicates".

char-flipcase char

Function

If char is a lowercase alphabetic character in the standard character set, char-flipcase returns its uppercase form. If char is an uppercase alphabetic character in the standard character set, char-flipcase returns its lowercase form. Otherwise, it returns char. If character style information is present it is preserved.

(char-flipcase #\X) => #\x
(char-flipcase #\b) => #\B

For a table of related items, see the section "Character Conversions".

char-font char

Function

Returns the font field of the character object specified by char. Genera characters do not have a font field so char-font always returns zero for character objects.

Genera does not support the Common Lisp concept of fonts, but supports the character style system instead. See the section "Character Styles". To find out the character style of a character, use si:char-style: See the function si:char-style.

The only reason to use char-font would be when writing a program intended to be portable to other Common Lisp systems.

(char-font #\A) => 0

For a table of related items: See the section "Character Fields".

char-font-limit

Constant

The value of char-font-limit is the upper exclusive limit for the value of values of the font bit. Genera characters do not have a font field so the value of char-font-limit is 1. Genera does not support the Common Lisp concept of fonts, but supports the y character style system instead. See the section "Character Styles".

(if (= char-font-limit 1)
 (setq *no-fonts* t)
 (setq *no-fonts* nil))

For a table of related items: See the section "Character Attribute Constants".
char-greaterp char &rest chars

Function

Compares characters for order; many of the string functions call it. char and chars must be characters; they cannot be integers. The result is t if char comes after chars ignoring case and style, otherwise nil. See the section "The Character Set". Details of the ordering of characters are in that section.

This function compares the code and bits fields and ignores character style and distinctions of alphabetic case.

(char-greaterp #\A #\B #\C) => NIL
(char-greaterp #\A #\B #\B) => T

Compatibility Note: Common Lisp specifies that char-greaterp should ignore bits. This difference is incompatible.

For a table of related items, see the section "Character Comparisons Ignoring Case and Style".

char-hyper-bit

Constant

The name for the hyper bit attribute. The value of char-hyper-bit is 8.

For a table of related items: See the section "Character Bit Constants".

char-int char

Function

Returns the character as an integer, including the fields that contain the character's code (which itself contains the character's set and subindex into that character set), bits, and style.

(char-int #\a) => 97
(char-int #\B) => 56
(char-int #\C-m-A) => 50331713 ;under Genera
(char-int
(make-character #\a :style '(nil :bold nil))) => 65633 ;under Genera
(char-int #\A) => 65

(eq (< (char-int char1) (char-int char2)))
(char< char1 char2))

=> T

(defvar char-arr (make-array 512))
(setf (elt char-arr (char-int #\a)) 'first)

For a table of related items, see the section "Character Conversions".

char-lessp char &rest chars

Function
This primitive compares characters for order; many of the string functions call it.  
`char` and `chars` must be characters; they cannot be integers. The result is `t` if `char`  
comes before `chars` ignoring case and style, otherwise `nil`. See the section "The  
Character Set". Details of the ordering of characters are in that section.

This comparison predicate compares the code and bits fields and ignores character  
style and distinctions of alphabetic case.

```
(char-lessp #'\A #'\B #'\C) => t
(char-lessp #'\A #'\B #'\B) => nil
```

**Compatibility Note:** Common Lisp specifies that `char-lessp` should ignore bits.  
This difference is incompatible.

For a table of related items, see the section "Character Comparisons Ignoring Case  
and Style".

---

**char-meta-bit**

The name for the meta bit attribute. The value of `char-meta-bit` is 2.

For a table of related items: See the section "Character Bit Constants".

---

**char-mouse-button char**

Returns the number corresponding to the mouse button that would have to be  
pushed to generate `char`. 0, 1, and 2 correspond to the Left, Middle, and Right  
mouse buttons, respectively.

Example:

```
(char-mouse-button #'\m-mouse-m) =>
1
```

The complementary function is `make-mouse-char`.

---

**char-mouse-equal char1 char2**

Returns `t` if the mouse characters `char1` and `char2` are equal, `nil` otherwise. `char-

-mouse-equal` checks that its arguments are really mouse characters and signals an  
error otherwise. You can also use `eql`, which is slightly faster, to compare mouse  
characters, when you do not require the argument checking.

---

**char-name char**

`char` must be a character object. `char-name` returns the name of the object (a  
string) if it has one. If the character has no name, or if it has non-zero bits or a  
character style other than NIL.NIL.NIL, `nil` is returned.

```
(char-name #'\Tab) => "Tab"
(char-name #'\Space) => "Space"
(char-name #'\A) => NIL
```
For a table of related items, see the section "Character Names".

**char-not-equal**  \texttt{char \&rest chars}  \ \ \ Function

This primitive compares characters for non-equality; many of the string functions call it. \texttt{char} and \texttt{chars} must be characters; they cannot be integers. **char-equal** compares code and bits, ignores case and character style, and returns \texttt{T} if the characters are not equal. Otherwise it returns \texttt{NIL}.  

\begin{align*}
\text{(char-not-equal \#'A \#'B)} & \Rightarrow \text{T} \\
\text{(char-not-equal \#'A \#'c-A)} & \Rightarrow \text{T} \\
\text{(char-not-equal \#'A \#'\text{\textbackslash A})} & \Rightarrow \text{NIL} \\
\text{(char-not-equal \#'a \#'\text{\textbackslash A})} & \Rightarrow \text{NIL}
\end{align*}

**Compatibility Note:** Common Lisp specifies that **char-not-equal** should ignore bits. This difference is incompatible.

For a table of related items, see the section "Character Comparisons Ignoring Case and Style".

**char-not-greaterp**  \texttt{char \&rest chars}  \ \ \ Function

This primitive compares characters for order; many of the string functions call it. \texttt{char} and \texttt{chars} must be characters; they cannot be integers. The result is \texttt{T} if \texttt{char} does not come after \texttt{chars} ignoring case and style, otherwise \texttt{NIL}. See the section "The Character Set". Details of the ordering of characters are in that section.

This comparison predicate compares the code and bits fields and ignores character style and distinctions of alphabetic case.

\begin{align*}
\text{(char-not-greaterp \#'A \#'B)} & \Rightarrow \text{T} \\
\text{(char-not-greaterp \#'a \#'\text{\textbackslash A})} & \Rightarrow \text{T} \\
\text{(char-not-greaterp \#'A \#'\text{\textbackslash A})} & \Rightarrow \text{T} \\
\text{(char-not-greaterp \#'A \#'\text{\textbackslash A})} & \Rightarrow \text{T}
\end{align*}

For a table of related items, see the section "Character Comparisons Ignoring Case and Style".

**char-not-lessp**  \texttt{char \&rest chars}  \ \ \ Function

This primitive compares characters for order; many of the string functions call it. \texttt{char} and \texttt{chars} must be characters; they cannot be integers. The result is \texttt{T} if \texttt{char} does not come before \texttt{chars} ignoring case and style, otherwise \texttt{NIL}. See the section "The Character Set". Details of the ordering of characters are in that section.

This comparison predicate compares the code and bits fields and ignores character style and distinctions of alphabetic case.

\begin{align*}
\text{(char-not-lessp \#'A \#'B)} & \Rightarrow \text{NIL} \\
\text{(char-not-lessp \#'B \#'b)} & \Rightarrow \text{T} \\
\text{(char-not-lessp \#'A \#'\text{\textbackslash A})} & \Rightarrow \text{T}
\end{align*}

For a table of related items, see the section "Character Comparisons Ignoring Case and Style".
For a table of related items, see the section "Character Comparisons Ignoring Case and Style".

si:char-style char  Function

Returns the character style of the character object specified by char. The returned value is a character style object.

(si:char-style #\a)
=> #<character-style nil nil nil 204004146>

(si:char-style (make-character #\a :style '(:swiss :bold nil)))
=> #<character-style swiss bold nil 116035602>

For a table of related items: See the section "Character Fields".

sys:char-subindex char  Function

Returns the subindex field of char as an integer.

For a table of related items, see the section "Character Fields".

char-super-bit  Constant

The name for the super bit attribute. The value of char-super-bit is 4.

For a table of related items: See the section "Character Bit Constants".

char-to-ascii ch  Function

Converts the character object ch to the corresponding ASCII code. This function works only for characters with neither bits nor style.

char-to-ascii performs an inverse mapping of the function ascii-to-char, and this mapping embeds the ASCII character character set in the Symbolics character set in an invertible way. There is no attempt to map more obscure ASCII control codes into the also obscure and unrelated Symbolics control codes. For example, Escape, is a character in the Symbolics character set corresponding to the key marked Escape. The ASCII code Escape is not the same as the Symbolsics Escape. See the function ascii-to-char. See the function ascii-code. See the section "ASCII Conversion String Functions".

It is an error to give char-to-ascii anything other than one of the 95 standard ASCII printing characters. To get the ASCII code of one of the other characters, use ascii-code, and give it the correct ASCII name.

The functions char-to-ascii and ascii-to-char provide the primitive conversions needed by ASCII-translating streams. They do not translate the Return character into a CR-LF pair; the caller must handle that. They just translate #\Return into CR and #\Line into LF. Except for CR-LF, char-to-ascii and ascii-to-char are wholly compatible with the ASCII-translating streams.
They ignore Symbolics control characters; the translation of \#\c-G is the ASCII code for G, not the ASCII code to ring the bell, also known as "control G." (ascii-to-char (ascii-code "BEL")) is \#\n, not \#\c-G. The translation from ASCII to character never produces a Symbolics control character.

For a table of related items, see the section "ASCII Characters".

**char-upcase char**

If `char`, which must be a character, is a lowercase alphabetic character in the standard character set, `char-upcase` returns its uppercase form; otherwise, it returns `char`. In Genera, if character style information is present, it is preserved. In no case will the font or bits attribute values differ from those of `char`.

```
(char-upcase #\a) => #\A
(char-upcase #\b) => #\B
(char-upcase #\3) => #\3
(char-upcase #\A) => #\A
```

For a table of related items, see the section "Character Conversions".

**zl:check-arg arg-name predicate-or-form type-string**

Macro

Checks arguments to make sure that they are valid. A simple example is:

```
(zl:check-arg foo stringp "a string")
```

`foo` is the name of an argument whose value should be a string. `stringp` is a predicate of one argument, which returns `t` if the argument is a string. "A string" is an English description of the correct type for the variable.

The general form of `zl:check-arg` is

```
(zl:check-arg var-name
  predicate
  description)
```

`var-name` is the name of the variable whose value is of the wrong type. If the error is proceeded this variable is `setq`'ed to a replacement value. `predicate` is a test for whether the variable is of the correct type. It can be either a symbol whose function definition takes one argument and returns non-`nil` if the type is correct, or it can be a nonatomic form which is evaluated to check the type, and presumably contains a reference to the variable `var-name`. `description` is a string which expresses `predicate` in English, to be used in error messages.

The `predicate` is usually a symbol such as `zl:fixp`, `stringp`, `zl:listp`, or `zl:closurep`, but when there isn't any convenient predefined predicate, or when the condition is complex, it can be a form. For example:
(defun test1 (a)
  (zl:check-arg a
    (and (numberp a) (≤ a 10.) (> a 0.))
    "a number from one to ten")
  ...)

If test1 is called with an argument of 17, the following message is printed:

The argument A to TEST1, 17, was of the wrong type.
The function expected a number from one to ten.

In general, what constitutes a valid argument is specified in two ways in a
zl:check-arg. description is human-understandable and predicate is executable. It is
up to the user to ensure that these two specifications agree.

zl:check-arg uses predicate to determine whether the value of the variable is of
the correct type. If it is not, zl:check-arg signals the sys:wrong-type-argument
condition. See the flavor sys:wrong-type-argument.

For a table of related items: See the section "Condition-Checking and Signalling
Functions and Variables".

zl:check-arg-type arg-name type &optional type-string Macro

A useful variant of the zl:check-arg form. A simple example is:

  (zl:check-arg-type foo :number)

foo is the name of an argument whose value should be a number. :number is a
value which is passed as a second argument to zl:typep; that is, it is a symbol
that specifies a data type. The English form of the type name, which gets put into
the error message, is found automatically.

The general form of zl:check-arg-type is:

  (zl:check-arg-type var-name
    type-name
    description)

var-name is the name of the variable whose value is of the wrong type. If the er-
ror is proceeded this variable is setq'ed to a replacement value. type-name de-
scribes the type which the variable's value ought to have. It can be exactly those
things acceptable as the second argument to zl:typep. description is a string which
expresses predicate in English, to be used in error messages. It is optional. If it is
omitted, and type-name is one of the keywords accepted by zl:typep, which de-
scribes a basic Lisp data type, then the right description is provided correctly. If it
is omitted and type-name describes some other data type, then the description is
the word "a" followed by the printed representation of type-name in lowercase.

The Common Lisp equivalent of zl:check-arg-type is the macro:

    check-type

For a table of related items: See the section "Condition-Checking and Signalling
Functions and Variables".
check-type place type &optional (type-string 'nil)  

Macro

Signals an error if the contents of place are not of the desired type. If you continue from this error, you will be asked for a new value; check-type stores the new value in place and starts over, checking the type of the new value and signalling another error if it is still not of the desired type. Subforms of place can be evaluated multiple times because of the implicit loop generated. check-type returns nil.

place must be a generalized variable reference acceptable to the macro setf.

type must be a type specifier; it is not evaluated. For standard Symbolics Common Lisp type specifiers, see the section "Type Specifiers".

type-string should be an English description of the type, starting with an indefinite article ("a" or "an"); it is evaluated. If type-string is not supplied, it is computed automatically from type. This optional argument is allowed because some applications of check-type may require a more specific description of what is wanted than can be generated automatically from the type specifier.

The error message mentions place, its contents, and the desired type.

Examples:

```
(setq bees '(bumble wasp jacket)) => (BUMBLE WASP JACKET)
(check-type bees (vector integer ))
=> Error : The value of BEES in SI:*EVAL, (BUMBLE WASP JACKET),
      was of the wrong type.
      The function expected a vector whose typical element
      is an integer.

(setq naards 'foo) => FOO
(check-type naards (integer 0 *) "a positive integer")
=> Error : The value of NAARDS in SI:*EVAL, FOO, was of the wrong
      type.
      The function expected a positive integer.
```

In CLOE, if a condition is signalled, handlers of this condition can use the functions type-error-object and type-error-expected-type to access the contents of place and the typespec, respectively.

Compatibility Note: In Zetalisp, the equivalent facility is called user::check-arg-type.

See the section "Data Types and Type Specifiers".

Using check-type in CLOE

In CLOE, if store-value is called, check-type will store the new value which is the argument to store-value (or which is prompted for interactively by the debugger) in place and start over, checking the type of the new value and signalling another error if it is still not the desired type. Subforms of place may be evaluated multiple times because of the implicit loop generated. check-type returns nil.

Here's an example of using check-type in CLOE:
Lisp> (SETQ AARDVARKS '(SAM HARRY FRED))
→ (SAM HARRY FRED)
Lisp> (CHECK-TYPE AARDVARKS (ARRAY * (3)))
Error: The value of AARDVARKS, (SAM HARRY FRED),
   is not a 3-long array.
1: Specify a value to use instead.
2: Return to Lisp Toplevel.
Debug> :1
Use Value: #(SAM FRED HARRY)
→ NIL
Lisp> AARDVARKS
→ #<ARRAY-T-3 13571>
Lisp> (MAP 'LIST #'IDENTITY AARDVARKS)
→ (SAM FRED HARRY)
Lisp> (SETQ AARDVARK-COUNT 'FOO)
→ FOO
Lisp> (CHECK-TYPE AARDVARK-COUNT (INTEGER 0 *) "a positive integer")
Error: The value of AARDVARK-COUNT, FOO, is not a positive integer.
1: Specify a value to use instead.
2: Return to Lisp Toplevel.
Debug> :2
Lisp>

circular-list &rest args 

Constructs a circular list whose elements are args, repeated infinitely. circular-list 
is the same as list except that the list itself is used as the last cdr, instead of nil. circular-list 
returns a circular list, repeating its elements infinitely. circular-list 
is especially useful with mapcar, as in the expression:

(mapcar (function +) foo (circular-list 5))

which adds each element of foo to 5. circular-list could have been defined by:

(defun circular-list (&rest elements)
  (setq elements (copylist* elements))
  (rplacd (last elements) elements)
  elements)

circular-list is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

cis radians 

This function can be defined by:

(defun cis (radians)
  (complex (cos radians) (sin radians)))
radians must be a noncomplex number.

\[(\text{signum } \#c(x \ y)) \rightarrow (\text{cis } (\text{phase } \#c(x \ y)))\]

Mathematically, this is equivalent to \[e^{i \text{ radians}}.\]

For a table of related items: See the section "Trigonometric and Related Functions".

\textbf{clos:class-name} \textit{class-object} \hspace{1cm} \textit{Generic Function}

Returns the name of the class object. You can use \textbf{setf} with \textbf{clos:class-name} to set the name of the class object.

\textit{class-object} \hspace{1cm} A class object.

If the class object has no name, \textbf{nil} is returned.

\textbf{clos:class-of} \textit{object} \hspace{1cm} \textit{Function}

Returns the class of the given object. The returned value is a class object.

\textit{object} \hspace{1cm} Any Lisp object.

\textbf{(flavor:method :clear si:heap)} \hspace{1cm} \textit{Method}

Remove all of the entries from the heap.

For a table of related items: See the section "Heap Functions and Methods".

\textbf{:clear-hash} \hspace{1cm} \textit{Message}

Removes all of the entries from the hash table. This message is obsolete; use \textbf{clrhash} instead.

\textbf{clear-input} &optional \textit{input-stream} \hspace{1cm} \textit{Function}

Clears any buffered input associated with \textit{input-stream}. It is primarily useful for removing type-ahead from keyboards when some kind of asynchronous error has occurred. If this operation doesn’t make sense for the stream involved, then \textbf{clear-input} does nothing. \textbf{clear-input} returns \textbf{nil}.

\begin{verbatim}
(let ((c (read-char)))
  (list (peek-char)
        (progn (clear-input) (read-char-no-hang))))xy
=> (#\x NIL)
\end{verbatim}

\textbf{:clear-input} \hspace{1cm} \textit{Message}
The stream clears any buffered input. If the stream does not handle this, the default handler ignores it.

**clear-output** &optional output-stream

Function

Some streams are implemented in an asynchronous, or buffered, manner. **clear-output** attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination. This is useful, for example, to abort a lengthy output to the terminal when an asynchronous error occurs. **clear-output** returns **nil**.

*output-stream*, if unspecified or **nil**, defaults to **standard-output**, and if **t**, is **terminal-io**.

**:clear-output**

Message

The stream clears any buffered output. If the stream does not handle this, the default handler ignores it.

**:clear-rest-of-line**

Message

Erases from the current position to the end of the current line. This message is supported by all terminal streams and windows.

**:clear-rest-of-line** replaces the obsolete message **:clear-eol**.

**:clear-rest-of-window**

Message

Erases from the current position to the end of the current window. This message is supported by all windows. Non-window streams do not support this operation.

**:clear-window**

Message

Erases the window on which this stream displays. Non-window streams do not support this operation.

**:clear-window** replaces the obsolete message **:clear-screen**.

**:close** &optional mode

Message

The stream is "closed", and no further operations should be performed on it; you can, however, **:close** a closed stream. If the stream does not handle **:close**, the default handler ignores it.

The **mode** argument is normally not supplied. If it is **:abort**, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly created file is deleted, as if it were never opened in the first place. Any previously existing file with the same name remains, undisturbed.
Use the Symbolics Common Lisp function `make-dynamic-closure`, which is equivalent to the function `zl:closure`.

`zl:closure` creates and returns a dynamic closure of `function` over the variables in `symbol-list`. Note that all variables on `symbol-list` must be declared special.

To test whether an object is a dynamic closure, use the `zl:closurep` predicate. See the section “Predicates”. The `typep` function returns the symbol `zl:closure` if given a dynamic closure. `(typep x :closure)` is equivalent to `(zl:closurep x)`.

See the section “Dynamic Closure-Manipulating Functions”.

Use the Symbolics Common Lisp function `dynamic-closure-alist`, which is equivalent to the function `zl:closure-alist`.

Returns an alist of `(symbol . value)` pairs describing the bindings which the dynamic closure performs when it is called. This list is not the same one that is actually stored in the closure; that one contains pointers to value cells rather than symbols, and `zl:closure-alist` translates them back to symbols so you can understand them. As a result, clobbering part of this list does not change the closure.

If any variable in the closure is unbound, this function signals an error.

See the section “Dynamic Closure-Manipulating Functions”.

Returns the closed function from the dynamic closure `closure`. This is the function that was the second argument to `zl:closure` when the dynamic closure was created.

See the section “Dynamic Closure-Manipulating Functions”.

Use the Symbolics Common Lisp function `dynamic-closure-variables`, which is equivalent to the function `zl:closure-variables`.

Creates and returns a list of all of the variables in the dynamic closure `closure`. It returns a copy of the list that was passed as the first argument to `zl:closure` when `closure` was created.

See the section “Dynamic Closure-Manipulating Functions”.

Returns `t` if its argument is a closure, otherwise `nil`.

Use the `clrhash` function.
Removes all of the entries from table and returns the hash table itself.

\[(\text{hash-table-count (clrhash old-hash-table)}) \Rightarrow 0\]

For a table of related items: See the section "Table Functions".

\textbf{zl:clrhash-equal hash-table} \hspace{1cm} \textit{Function}

Removes all of the entries from hash-table. This function is obsolete; use \texttt{clrhash} instead.

\textbf{sys:cl-structure-printer structure-name object stream depth} \hspace{1cm} \textit{Macro}

Expands into an efficient function that prints a given structure object of type structure-name to the specified stream in \#S format. It depends on the information calculated by \texttt{defstruct}, and so is only useful after the \texttt{defstruct} form has been compiled. This macro enables a structure print function to respect the variable \texttt{*print-escape*}.

\begin{verbatim}
(defun foo-printer (object stream depth)
  (if *print-escape*
      (sys:cl-structure-printer foo object stream depth)
      other-printing-strategy))
\end{verbatim}

For a table of related items: See the section "Functions Related to \texttt{defstruct} Structures".

\textbf{code-char code \&optional (bits 0) (font 0)} \hspace{1cm} \textit{Function}

Constructs a character given its code field. code, bits, and font must be non-negative integers. If \texttt{code-char} cannot construct a character given its arguments, it returns \texttt{nil}.

To set the bits of a character, supply one of the character bits constants as the \texttt{bits} argument. See the section "Character Bit Constants".

For example:

\begin{verbatim}
(code-char 65 char-control-bit) => #\c-A
(char-code 65) => #\A
(char-code 65 4) => #\Super-A
\end{verbatim}

Since the value of \texttt{char-font-limit} is 1, the only valid value of \texttt{font} is 0. The only reason to use the \texttt{font} option would be when writing a program intended to be portable to other Common Lisp systems.

In Genera, to construct a new character that has character style other than NIL.NIL.NIL, use \texttt{make-character}. See the function \texttt{make-character}. 
For a table of related items, see the section "Making a Character".

**coerce object result-type**

Converting an object to another object of another type.

Object is a Lisp object.

Result-type must be a type-specifier; object is converted to an equivalent object of the specified type. If object is already of the specified type, as determined by typep, it is returned.

If the coercion cannot be performed, an error is signalled. In particular, (coerce x nil) always signals an error.

Example:

(coerce 'x nil)

=> Error: I don't know how to coerce an object to nothing

It is not generally possible to convert any object to be of any type whatsoever; only certain conversions are allowed:

Any sequence type can be converted to any other sequence type, provided the new sequence can contain all actual elements of the old sequence (it is an error if it cannot). If the result-type is specified as simply array, for example, then array t is assumed. A specialized type such as string or (vector (complex short-float)) can be specified;

Examples:

(coerce '(a b c) 'vector) => #(A B C)
(coerce '(a b c) 'array) => #(A B C)
(coerce #*101 '(vector (complex short-float))) => #(1 0 1)
(coerce #(4 4) 'number)

=> Error: I don't know how to coerce an object to a number

Elements of the new sequence will be eql to corresponding elements of the old sequence. Note that elements are not coerced recursively. If you specify sequence as the result-type, the argument can simply be returned without copying it, if it already is a sequence.

Examples:

(coerce #(8 9) 'sequence) => #(8 9)
(eq (coerce #(1 2) 'sequence) #(1 2)) => NIL
(equalp (coerce #(1 2) 'sequence) #(1 2)) => T

In this respect, (coerce sequence type) differs from (concatenate type sequence), since the latter is required to copy the argument sequence.

Some strings, symbols, and integers can be converted to characters. If object is a string of length 1, the sole element of the string is returned. If object is a symbol whose print name is of length 1, the sole element of the print name is returned. If object is an integer n, (int-char n) is returned.
Examples:

(coerce "b" 'character) => #\b
(coerce "ab" 'character)
=> Error: "AB" is not one character long.
(coerce 'a 'character) => #\A
(coerce 'ab 'character)
=> Error: "AB" is not one character long.
(coerce 65 'character) => #\A
(coerce 150 'character) => #\Circle

Any non-complex number can be converted to a short-float, single-float double-
float, or long-float. If simply float is specified as the result-type and if object is
not already a floating-point number of some kind, object is converted to a single-
float.

Examples:

(coerce 0 'short-float) => 0.0
(coerce 3.5L0 'float) => 3.5d0
(coerce 7/2 'float) => 3.5

Any number can be converted to a complex number. If the number is not already
complex, a zero imaginary part is provided by coercing the integer zero to the type
of the given real part. If the given real part is rational, however, the rule of
canonicalization for complex rational numbers results in the immediate reconver-
sion of the the result type from type complex back to type rational.

Examples:

(coerce 4.5s0 'complex) => #C(4.5 0.0)
(coerce 7/2 'complex) => 7/2
(coerce #C(7/2 0) '(complex double-float))
=> #C(3.5d0 0.0d0)

Any object can be coerced to type t.

Example:

(coerce 'house 't) => HOUSE
is equivalent to

(identity 'house) => HOUSE

Coercions from floating-point numbers to rational numbers, and of ratios to inte-
gers are not supported because of rounding problems. Use one of the specialized
functions such as rational, rationalize, floor, and ceiling instead. See the section
"Numeric Type Conversions".

Similarly, coerce does not convert characters to integers; use the specialized func-
tions char-code or char-int instead.

See the section "Data Types and Type Specifiers".
**collect expr [into var]**

Causes the values of `expr` on each iteration to be collected into a list. When the epilogue of the `loop` is reached, `var` has been set to the accumulated result and can be used by the epilogue code.

It is safe to reference the values in `var` during the loop, but they should not be modified until the epilogue code for the loop is reached.

The forms `collect` and `collecting` are synonymous.

Examples:

```lisp
(defun loop1 (start end)
  (loop for x from start to end
        collect x)) => LOOP1
(loop1 0 4) => (0 1 2 3 4)

(defun loop2 (small-list)
  (loop for x from 0
        for item in small-list
        collect (list x item))) => LOOP2
(loop2 '("one" "two" "three" "four")
  => ((0 "one") (1 "two") (2 "three") (3 "four"))
```

The following examples are equivalent.

```lisp
(defun loop3 (small-list)
  (loop for x from 0
        for item in small-list
        collect x into result-1
        collect item into result-2
        finally (print (list result-1 result-2))) => LOOP3
(loop3 '(a b c d e f)) =>
  ((0 1 2 3 4 5) (A B C D E F)) NIL

(defun loop3 (small-list)
  (loop for x from 0
        for item in small-list
        collecting x into result-1
        collecting item into result-2
        finally (print (list result-1 result-2))) => LOOP3
(loop3 '(a b c d e f)) =>
  ((0 1 2 3 4 5) (A B C D E F)) NIL
```

Not only can there be multiple accumulations in a `loop`, but a single accumulation can come from multiple places within the same `loop` form, if the types of the collections are compatible. `collect`, `nconc`, and `append` are compatible.

See the section "Accumulating Return Values for `loop"."
**zl:comment** *Special Form*

Ignores its form and returns the symbol `zl:comment`. Example:

```lisp
(defun foo (x)
  (cond ((null x) 0)
        (t (comment x has something in it)
           (+ (foo (cdr x))))))
```

Usually it is preferable to comment code using the semicolon-macro feature of the standard input syntax. This allows you to add comments to your code that are ignored by the Lisp reader. Example:

```lisp
(defun foo (x)
  (cond ((null x) 0)
        (t (+ (foo (cdr x))) ; x has something in it))
)
```

A problem with such comments is that they are discarded when the form is read into Lisp. If the function is read into Lisp, modified, and printed out again, the comment is lost. However, this style of operation is hardly ever used; usually the source of a function is kept in an editor buffer and any changes are made to the buffer, rather than the actual list structure of the function. Thus, this is not a real problem.

See the section "Functions and Special Forms for Constant Values".

**common** *Type Specifier*

This is the type specifier symbol denoting an *exhaustive union* of the following Common Lisp data types:

- `cons`, `symbol`
- `(array x)`, where `x` is either `t` or a subtype of `common`
- `string`, `fixnum`, `bignum`, `ratio`, `short-float`, `single-float`, `double-float`, `long-float`
- `(complex x)` where `x` is a subtype of `common`
- `standard-char`, `hash-table`, `readtable`, `package`, `pathname`, `stream`, `random-state`
- and all types created by the user with `defstruct`, or `defflavor`.

The type `common` is a *subtype* of type `t`.

Examples:

```lisp
(typep '#c(3 4) 'common) => T
(subtypep 'common t) => T and T
(commonp 'cons) => T
(sys:type-arglist 'common) => NIL and T
```
(setq four
  (let ((x 4))
    (closure '(x) 'zerop))) => #<DTP-CLOSURE 1510647>

(typep four 'sys:dynamic-closure) => T

(subtypep 'sys:dynamic-closure 'common) => NIL and NIL

See the section "Data Types and Type Specifiers".

**commonp object**

*Function*

Returns true if *object* is a standard Common Lisp data object; otherwise, returns false. However, some standard Common Lisp data objects (such as characters with one or more bits attributes set) and function objects are not included in type common. All structure objects are of type common, even though their types are defined by the user with defstruct.

(commonp x) ≡ (typep x 'common)

Examples:

(commonp 1.5d9) => T
(commonp 1.0) => T
(commonp -12.) => T
(commonp '3kd) => T
(commonp 'symbol) => T
(commonp #c(3.4)) => T
(commonp 4) => T is equivalent to (typep 4 'common) => T

See the section "Data Types and Type Specifiers".
See the section "Predicates".

**commonp object**

*Function*

Returns true if *object* is a standard Common Lisp data object; otherwise, returns false. However, some standard Common Lisp data objects (such as characters with one or more bits attributes set) and function objects are not included in type common. All structure objects are of type common, even though their types are defined by the user with defstruct.

(commonp x) ≡ (typep x 'common)

Examples:
(commonp 1.5d9) => T  
(commonp 1.0) => T  
(commonp -12.) => T  
(commonp '3kd) => T  
(commonp 'symbol) => T  
(commonp #c(3 4)) => T  
(common 4) => T  is equivalent to (typep 4 'common) => T

See the section "Data Types and Type Specifiers".
See the section "Predicates".

**compile-flavor-methods** flavor1 flavor2...  

*Macro*

Causes the combined methods of a program to be compiled at compile-time, and the data structures to be generated at load-time, rather than both happening at run-time. **compile-flavor-methods** is thus a very good thing to use, since the need to invoke the compiler at run-time slows down a program using flavors the first time it is run. (The compiler is still called if incompatible changes have been made, such as addition or deletion of methods that must be called by a combined method.)

It is necessary to use **compile-flavor-methods** when you use the :constructor option for **defflavor**, to ensure that the constructor function is defined.

Generally, you use **compile-flavor-methods** by including the forms in a file to be compiled. (The **compile-flavor-methods** forms can also be interpreted.) This causes the compiler to include the automatically generated combined methods for the named flavors in the resulting binary file, provided that all of the necessary flavor definitions have been made. Furthermore, when the binary file is loaded, internal data structures (such as the list of all methods of a flavor) are generated.

You should use **compile-flavor-methods** only for flavors that will be instantiated. For a flavor that will never be instantiated (that is, one that only serves to be a component of other flavors that actually do get instantiated), it is almost always useless. The one exception is the unusual case where the other flavors can all inherit the combined methods of this flavor instead of each having its own copy of a combined method that happens to be identical to the others.

The **compile-flavor-methods** forms should be compiled after all of the information needed to create the combined methods is available. You should put these forms after all of the definitions of all relevant flavors, wrappers, and methods of all components of the flavors mentioned.

In general, Flavors cannot guarantee that **defmethod** macro-expands correctly unless the flavor (and all of its component flavors) have been compiled. Therefore, the compiler gives a warning when you try to compile a method before the flavor and its components have been compiled.

If you see this warning and no other warnings, it is usually the case that the flavor system did compile the method correctly.
In complicated cases, such as a regular function and an internal flavor function (defined by `defun-in-flavor` or the related functions) having the same name, the flavor system cannot compile the method correctly. In those cases it is advisable to compile all the flavors first, and then compile the method.

See the function `flavor:print-flavor-compile-trace`.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

**compiled-function**

*Type Specifier*

This is the type specifier symbol for the predefined Lisp data type `compiled-function`.

Examples:

```lisp
(typep (compile nil '(lambda (a b) (+ a b))) 'compiled-function)
=> T
(zl:typep (compile nil '(lambda (a b) (+ a b))))
=> :COMPILED-FUNCTION
(sys:type-arglist 'compiled-function) => NIL and T
(compiled-function-p (compile nil '(lambda (a) (+ a a)))) => T
```

See the section "Data Types and Type Specifiers".

See the section "Functions".

**compiled-function-p x**

*Function*

Returns t if its argument is any compiled code object.

**compiler-let bindlist &body body**

*Special Form*

When interpreted, a `compiler-let` form is equivalent to `let` with all variable bindings declared special. When the compiler encounters a `compiler-let`, however, it performs the bindings specified by the form (no compiled code is generated for the bindings) and then compiles the body of the `compiler-let` with all those bindings in effect. In particular, macros within the body of the `compiler-let` form are expanded in an environment with the indicated bindings. See the section "Nesting Macros".

`compiler-let` allows compiler switches to be bound locally at compile time, during the processing of the `body` forms. Value forms are evaluated at compile time. See the section "Compiler Switches". In the following example the use of `compiler-let` prevents the compiler from open-coding the `map`.

```lisp
(compiler-let (((compiler:open-code-map-switch nil)))
  (map (function (lambda (x) ...)) foo))
```
The body of the **compiler-let** form is an implicit **progn**; thus, the forms are evaluated sequentially, and the value of the last evaluated form is returned. The difference between **compiler-let** and **let** is that the former use the bindings at the time of semantic analysis, rather than use the bindings at execution time. For example, causing the compiler to use the bindings while generating code for the body, rather than generate code for the bindings. Of course, another difference is the implicit special declaration of the bindings. In general, only embedded **macrolet** and **compiler-let** forms can reliably recognize the bindings (though in some dialects these bindings may coincidentally be visible in interpreted code).

In the following example, **compiler-let** enables two macros which are used together for effective communication. First, the macro **with-end-push** establishes a context that points to the end of a list. Second macro **push-onto-end** uses the pointer to add items to the end of the list, much as **push** adds to the beginning of a list. The special variable ***end-ptr*** is bound to the pointer. Therefore, when **push-onto-end** is expanded in the context of that binding, the appropriate pointer is employed.

```lisp
(defun *end-ptr* nil)

(defmacro with-end-push (list &body body)
  (let ((lastptr (gensym)))
    `(let ((lastptr (last ,list)))
        (compiler-let ((*end-ptr* ',lastptr)
                       ,body))))

(defmacro push-onto-end (val)
  `(setf ,*end-ptr* (setq ,*end-ptr* (cons ,val nil))))

(let ((mylist (list 1 2 3))
      (a-list (list 'a 'b 'c 'd))
      (with-end-push mylist
          (dolist (l a-list mylist)
              (push-onto-end l)))))

=> (1 2 3 A B C D)
```

The difference between **compiler-let** and **let** is only relevant when the actual code that contains the macro with **compiler-let** is compiled.

See the section "Special Forms for Binding Variables".

**:complete-connection** &key (timeout (* 60. 6.))  
This message is sent to a new stream created by **:start-open-auxiliary-stream**, in order to wait for the connection to be fully established. **:complete-connection** is used whether or not this side is active. **Timeout** is interpreted as the number of sixtieths of a second to wait before timing out.
When :complete-connection returns, the stream is fully connected to an active network connection. At this point, :connected-p to that stream returns t.

:complete-connection signals an error if the connection times out or does not complete for another reason.

complex &optional ( type "*" )  

Type Specifier

complex is the type specifier symbol for the predefined Lisp complex number type.

The types complex, rational, and float are pairwise disjoint subtypes of the type number.

This type specifier can be used in either symbol or list form. Used in list form, complex allows the declaration and creation of complex numbers, whose real part and imaginary part are each of type type.

Examples:

(typep #c(3 4) 'complex) => T  
(subtypep 'complex 'number) => T and T ; subtype and certain  
(typep '((complex 3 4) 'common) => T

The expression

(complexp #c(4/5 7.0)) => T

Is equivalent to

(typep #c(4/5 7.0) 'complex) => T

Here is an example of using the type argument for complex:

(typep #c(3.0 4.0) 'complex) => T

(typep #c(3.0 4.0) '(complex integer)) => NIL

(typep #c(3.0 4.0) '(complex float)) => T

(typep #c(3 4) '(complex integer)) => T

See the section "Data Types and Type Specifiers".

See the section "Numbers".

complex realpart &optional imagpart  

Function

Constructs a complex number from real and imaginary noncomplex parts, applying complex canonicalization.

If the types of the real and imaginary parts are different, the coercion rules are applied to make them the same. If imagpart is not specified, a zero of the same type as realpart is used. If realpart is an integer or a ratio, and imagpart is 0, the result is realpart.

Examples:
(complex 7) => 7
(complex 4.3 0) => #C(4.3 0.0)
(complex 2 0) => 2
(complex 3 4) => #C(3 4)
(complex 3 4.0) => #C(3.0 4.0)
(complex 3.0d0 4) => #C(3.0d0 4.0d0)
(complex 5/2 4.0d0) => #C(2.5d0 4.0d0)

Related Functions:
  realpart
  imagpart

For a table of related items: See the section "Functions that Decompose and Construct Complex Numbers".

### complexp object

Function

Returns `t` if `object` is a complex number, otherwise `nil`. The following code tests whether `a` and `b` are numbers. If numbers, they are added. Otherwise, we attempt to extract complex numbers that are then tested by `complexp`.

```
(if (and (numberp a) (numberp b))
  (+ a b)
  (if (and (consp a)
             (complexp (cadr a))
             (consp b)
             (complexp (cadr b)))
    (+ (cadr a) (cadr b))
    (error "couldn’t extract complexs from ~a and ~a" a b))
```

For a table of related items, see the section "Numeric Type-checking Predicates".

### complexp object

Function

Returns `t` if `object` is a complex number, otherwise `nil`. The following code tests whether `a` and `b` are numbers. If numbers, they are added. Otherwise, we attempt to extract complex numbers that are then tested by `complexp`.

```
(if (and (numberp a) (numberp b))
  (+ a b)
  (if (and (consp a)
             (complexp (cadr a))
             (consp b)
             (complexp (cadr b)))
    (+ (cadr a) (cadr b))
    (error "couldn’t extract complexs from ~a and ~a" a b))
```

For a table of related items, see the section "Numeric Type-checking Predicates".

### flavor:compose-handler

Function

`flavor:compose-handler generic flavor-name &key env`
Finds the methods that handle the specified generic operation on instances of the specified flavor. Four values are returned:

**handler-function-spec**
- The name of the handler, which can be a combined method, a single method, or an instance-variable accessor.

**combined-method-list**
- A list of function specs of all the methods called, in order of execution; the order is approximate because of wrappers.

**methodcombination**
- A list of the method combination type and parameters to it.

**error**
- nil normally, otherwise a string describing an error that occurred.

For example, to use `flavor:compose-handler` on the generic function `change-status` for the flavor `box-with-cell`:

```
(flavor:compose-handler 'change-status 'box-with-cell)
```

```
-->(FLAVOR:COMBINED CHANGE-STATUS BOX-WITH-CELL)
  ((FLAVOR:METHOD CHANGE-STATUS CELL)
   (FLAVOR:METHOD CHANGE-STATUS BOX-WITH-CELL))
  (:AND :MOST-SPECIFIC-LAST)
  NIL
```

The generic function `change-status` and the methods for the flavors `box-with-cell` and `cell` are defined elsewhere. See the section "Example of Programming with Flavors: Life".

In the second return value of sample output here, we put each method on one line, for readability. This is not done by `flavor:compose-handler`.

For documentation of the `env` parameter, see the function `flavor:compose-handler-source`.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

---

**flavor:compose-handler-source**

Finds the methods that handle the specified `generic` operation on instances of the flavor specified by `flavor-name`, and finds the source code of the combined method (if any). Seven values are returned:

**form**
- A Lisp form which is the body of the combined method. If there isn't actually a combined method, this is nil.

**handler-function-spec**
- The name of the handler, which can be a combined method, a single method, or an instance-variable accessor.
combined-method-list
A list of function specs of all the methods called, in order of execution; the order is approximate because of wrappers.

wrapper-sources
Information that the combined method requires so that Flavors knows when it needs to be recompiled.

lambda-list
A list describing what the arguments of the combined method should be (not including the three internal arguments automatically given to all methods).

method-combination
A list of the method combination type and parameters to it.

error
nil normally, otherwise a string describing an error that occurred.

flavor:compose-handler-source is generally slower than flavor:compose-handler, since the latter function can usually take advantage of pre-computed information present in virtual memory.

The env parameter to flavor:compose-handler and flavor:compose-handler-source can be used to insert hypotheses into their computations. If env is nil, the generics, flavors, and methods in the running world are used. env can be an alist of modifications to the running world; each element takes the form:

(name flavor-structure generic-structure (method definition) ...)

Everything except name can be nil. name is the name of a generic, or a flavor, or both. flavor-structure is nil or the internal structure that describes the flavor. generic-structure is nil or the internal structure that describes the generic function. The remaining elements of an alist element refer to methods of the flavor named name; method is a function spec and definition is nil if that method is to be ignored, t if the method is to be assumed to exist, or the actual definition (expander function) in the case of a wrapper.

env can also be the symbol compile, which is used internally to access the compile-time environment.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

si:compress-who-calls-database
Function

Makes the who-calls database more compact and efficient. Call this function after si:enable-who-calls. With the function (si:enable-who-calls ':all), the function si:compress-who-calls-database takes a long time to complete its job. However, it is faster than using si:full-gc, and you can perform an Incremental Disk Save (IDS) afterwards. See the section "Using the Incremental Disk Save (IDS) Facility".

clos:compute-applicable-methods
generic-function function-arguments
Function
Returns the set of methods that are applicable for function-arguments; the methods are sorted according to precedence order.

**generic-function**  A generic function object.

**function-arguments**  A list of the arguments to the generic function.

---

**concatenate result-type &rest sequences**  Function

Combines the elements of the sequences in the order the sequences were given as arguments. Returns the new, combined sequence.

The result does not share any structure with any of the argument sequences. The type of the result is specified by result-type, which must be a subtype of type sequence. It must be possible for every element of the argument sequences to be an element of a sequence of type result-type.

**sequence** can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:

```lisp
(concatenate 'vector "abc" #(ab) "gh") => #(\a \b \c AB \g \h)
```

```lisp
(setq vector (vector 'a 'b '1 '2)) => #(A B 1 2)
```

```lisp
(setq list (make-list 3 :initial-element 'blah))
  => (BLAH BLAH BLAH)
```

```lisp
(concatenate 'list vector list)
  => (A B 1 2 BLAH BLAH BLAH)
```

```lisp
(concatenate 'vector list vector) => #(BLAH BLAH BLAH A B 1 2)
```

```lisp
(concatenate 'string Threshold => "abc"
```

If only one sequence argument is provided and it has the type specified by result-type, concatenate is required to copy the argument rather than simply returning it. If a copy is not required, but only possible type-conversion, then the function coerce can be appropriate.

For a table of related items: See the section "Sequence Construction and Access".

---

**cond &rest clauses**  Special Form

Consists of the symbol cond followed by several clauses. Each clause consists of a predicate form, called the antecedent, followed by zero or more consequent forms.
Each clause represents a case that is selected if its antecedent is satisfied and the antecedents of all preceding clauses were not satisfied. When a clause is selected, its consequent forms are evaluated.

`cond` processes its clauses in order from left to right. First, the antecedent of the current clause is evaluated. If the result is `nil`, `cond` advances to the next clause. Otherwise, the `cdr` of the clause is treated as a list of consequent forms that are evaluated in order from left to right. After evaluating the consequents, `cond` returns without inspecting any remaining clauses. The value of the `cond` special form is the value of the last consequent evaluated, or the value of the antecedent if there were no consequents in the clause. If `cond` runs out of clauses, that is, if every antecedent evaluates to `nil`, and thus no case is selected, the value of the `cond` is `nil`.

Examples:

```
(cond) => NIL

(cond ( (> 2 3) (print "2 equals 3, new math"))
     ( (< 3 3) (print "3 < 3, not yet !"))) => NIL

(cond ((equal 'Becky 'Becky) "Girl")
     ((equal 'Tom 'Tom) "Boy")) => "Girl"

(cond ((equal 'Rover 'Red) "dog")
     ((equal 'Pumpkin 'Pickles) "cat")
     (t "rat")) => "rat"

(cond ((zerop x) ;First clause:
     (+ y 3)) ;(zerop x) is the antecedent.
          ;(+ y 3) is the consequent.
     ((null y) ;A clause with 2 consequents:
          (setq y 4) ;this
          (cons x z)) ;and this.
     (z) ;A clause with no consequents: the antecedent
          ;is just z. If z is non-nil, it is returned.
     (t ;An antecedent of t
     105) ;is always satisfied.
     ) ;This is the end of the cond.
```

For a table of related items: See the section "Conditional Functions". The following is an approximate possible implementation of `zl-user:constantp` using `cond:`
(defun constantp (object)
  (cond ((consp object) (eq (car object) (quote quote)))
        ((not (symbolp object)) t)
        ((defined-constant-p object) t)
        ((or (null object) (eq object t)) t)
        ((keywordp object) t)
        (t nil)))

cond-every &body clauses

Has the same syntax as cond, but executes every clause whose predicate is satisfied, not just the first. If a predicate is the symbol otherwise, it is satisfied if and only if no preceding predicate is satisfied. The value returned is the value of the last consequent form in the last clause whose predicate is satisfied. Multiple values are not returned.

Examples:

(cond-every) => NIL

(cond-every ((> 2 3) (print "sister"))
            ((= 2 3) (print "brother"))) => NIL

(cond-every ((equal 'mom 'mom) (princ "mother "))
            ((equal 'dog 'cat) (princ "pet dog")
            ((equal 'dad 'dad) (princ "father"))
            => mother father "father"

(cond-every ((= 1 1) t) ((= 2 2) "yes!")
            (otherwise "no") => "yes!"

For a table of related items: See the section "Conditional Functions".

condition-bind list &body body

Binds handlers for conditions and then evaluates its body with those handlers bound. One of the handlers might be invoked if a condition is signalled while the body is being evaluated. The handlers bound have dynamic scope.

The following simple example sets up application-specific handlers for two standard error conditions, fs:file-not-found and fs:delete-failure.

(condition-bind (((fs:file-not-found 'my-fnf-handler)
            (fs:delete-failure 'my-delete-handler))
            (deletef pathname))

The format for condition-bind is:
(condition-bind ((condition-flavor-1 handler-1)
                 (condition-flavor-2 handler-2)
                ...
                 (condition-flavor-m handler-m))
form-1
form-2
...
form-n)

condition-flavor  The name of a condition flavor or a list of names of condition flavors. condition-flavor need not be unique or mutually exclusive. (See the section "Finding a Handler". Search order is explained in that section.)

handler  A form that is evaluated to produce a handler function. One handler is bound for each condition flavor clause in the list. The forms for binding handlers are evaluated in order from handler-1 to handler. All the handler-j forms are evaluated and then all handlers are bound.

When handler is a lambda-expression, it is compiled. The handler function is a lexical closure, capable of referring to the lexical variables of the containing block.

Note: handler must have one argument, which is the condition object. Otherwise, an error is signalled.

form  A body, constituting an implicit progn. The forms are evaluated sequentially. The condition-bind form returns whatever values form returns (nil when the body contains no forms). The handlers that are bound disappear when the condition-bind form is exited.

If a condition signal occurs for one of the condition-flavors during evaluation of the body, the signalling mechanism examines the bound handlers in the order in which they appear in the condition-bind form, invoking the first appropriate handler. You can think of the mechanism as being analogous to typecase or case. It invokes the handler function with one argument, the condition object. The handler runs in the dynamic environment in which the error occurred; no throw is performed.

Any handler function can take one of three actions:

- It can return nil to indicate that it does not want to handle the condition after all. The handler is free to decide not to handle the condition, even though the condition-flavors matched. (In this case the signalling mechanism continues to search for a condition handler.)

- It can throw to some outer catch-form, using throw.
• If the condition has any proceed types, it can proceed from the condition by calling the `sys:proceed` generic function on the condition object and returning the resulting values. In this case, `signal` returns all of the values returned by the handler function. (Proceed types are not available for conditions signalled with `error`. See the section "Proceeding".)

The conditional variant of `condition-bind` is the form:

```lisp
condition-bind-if
```

For a table of related items, see the section "Basic Forms for Bound Handlers".

```lisp
condition-bind-default list &body body
```

Binds its handlers on the default handler list instead of the bound handler list. See the section "Finding a Handler". In other respects `condition-bind-default` is just like `condition-bind`. The default handlers are examined by the signalling mechanism only after all of the bound handlers have been examined. Thus, a `condition-bind-default` can be overridden by a `condition-bind` outside of it. This advanced feature is described in more detail in another section. See the section "Default Handlers and Complex Modularity".

The conditional variant of `condition-bind-default` is the form:

```lisp
condition-bind-default-if
```

For a table of related items, see the section "Basic Forms for Default Handlers".

```lisp
condition-bind-default-if cond list &body body
```

Binds its handlers on the default handler list instead of the bound handler list. See the section "Finding a Handler". In other respects `condition-bind-default-if` is just like `condition-bind-if`. The default handlers are examined by the signalling mechanism only after all of the bound handlers have been examined. Thus, a `condition-bind-default-if` can be overridden by a `condition-bind` outside of it. This advanced feature is described in more detail in another section. See the section "Default Handlers and Complex Modularity".

For a table of related items, see the section "Basic Forms for Default Handlers".

```lisp
condition-bind-if cond list &body body
```

Binds its handlers conditionally. In all other respects, it is just like `condition-bind`. It has an extra subform called `cond`, for the conditional. Its format is:
\[
(\text{condition-bind-if } \text{cond}
  \begin{array}{l}
  ((\text{condition-flavor-1 handler-1})
  \\
  (\text{condition-flavor-2 handler-2})
  \\
  \ldots
  \\
  (\text{condition-flavor-m handler-m}))
  \end{array}
  \\
  \begin{array}{l}
  \text{form-1}
  \\
  \text{form-2}
  \\
  \ldots
  \\
  \text{form-n}
  \end{array}
)\]

\textbf{condition-bind-if} first evaluates \textit{cond}. If the result is \textit{nil}, it evaluates the handler forms but does not bind any handlers. It then executes the body as if it were a \textit{progn}. If the result is not \textit{nil}, it continues just like \textbf{condition-bind} binding the handlers and executing the body.

For a table of related items: See the section "Basic Forms for Bound Handlers".

\[
\textbf{condition-call} \ (\text{&rest varlist}) \ \text{form} \ \&\text{body clauses} \quad \text{Special Form}
\]

Binds handlers for conditions, expressing the handlers as clauses of a case-like construct instead of as functions. These handlers have dynamic scope.

\textbf{condition-call} and \textbf{condition-case} have similar applications. The major distinction is that \textbf{condition-call} provides the mechanism for using a complex conditional criterion to determine whether or not to use a handler. \textbf{condition-call} clauses have the ability to decline to handle a condition because the clause is selected on the basis of the predicate, rather than on the basis of the type of a condition.

The format is:

\[
(\text{condition-call} \ (\text{var})
  \begin{array}{l}
  \text{form}
  \\
  (\text{predicate-1 form-1-1 form-1-2 ... form-1-n})
  \\
  (\text{predicate-2 form-2-1 form-2-2 ... form-2-n})
  \\
  \ldots
  \\
  (\text{predicate-m form-m-1 form-m-2 ... form-m-n})
  \end{array}
)\]

Each \textit{predicate} must be a function of one argument. The predicates are called, rather than evaluated. The \textit{form-m-n} are a body, a list of forms constituting an implicit \textit{progn}. The handler clauses are bound simultaneously.

When a condition is signalled, each predicate in turn (in the order in which they appear in the definition) is applied to the condition object. The corresponding handler clause is executed for the first predicate that returns a value other than \textit{nil}. The predicates are called in the dynamic environment of the signaller.

\textbf{condition-call} takes the following actions when it finds the right predicate:

1. It automatically performs a \textit{throw} to unwind the dynamic environment back to the point of the \textbf{condition-call}. This discards the handlers bound by the \textbf{condition-call}. 
2. It executes the body of the corresponding clause.

3. It makes \texttt{condition-call} return the values produced by the last form in the clause.

During the execution of the clause, the variable \texttt{var} is bound to the condition object that was signalled. If none of the clauses needs to examine the condition object, you can omit \texttt{var}:

\begin{verbatim}
(condition-call () ...)
\end{verbatim}

\textbf{condition-call and :no-error}

As a special case, \texttt{predicate-m} (the last one) can be the special symbol \texttt{:no-error}. If \texttt{form} is evaluated and no error is signalled during the evaluation, \texttt{condition-case} executes the \texttt{:no-error} clause instead of returning the values returned by \texttt{form}. The variables \texttt{vars} are bound to the values produced by \texttt{form}, in the style of \texttt{multiple-value-bind}, so that they can be accessed by the body of the \texttt{:no-error} case. Any extra variables are bound to \texttt{nil}.

Some limitations on predicates:

- Predicates must not have side effects. The number of times that the signalling mechanism chooses to invoke the predicates and the order in which it invokes them are not defined. For side effects in the dynamic environment of the signal, use \texttt{condition-bind}.

- The predicates are not lexical closures and therefore cannot access variables of the lexically containing form, unless those variables are declared \texttt{special}.

- Lambda-expression predicates are not compiled.

The conditional variant of \texttt{condition-call} is the form:

\begin{verbatim}
condition-call-if
\end{verbatim}

For a table of related items: See the section "Basic Forms for Bound Handlers".

\begin{verbatim}
condition-call-if cond (&rest varlist) form &body clauses
\end{verbatim}

Binds its handlers conditionally. In all other respects, it is just like \texttt{condition-call}. Its format includes \texttt{cond}, the subform that controls binding handlers:

\begin{verbatim}
(condition-call-if cond (var)
 form
 (predicate-1 form-1-1 form-1-2 ... form-1-n)
 (predicate-2 form-2-1 form-2-2 ... form-2-n)
 ... 
 (predicate-m form-m-1 form-m-2 ... form-m-n))
\end{verbatim}
**condition-call-if** first evaluates *cond*. If the result is **nil**, it does not set up any handlers; it just evaluates the form. If the result is not **nil**, it continues just like **condition-call**, binding the handlers and evaluating the form.

The **:no-error** clause applies whether or not *cond* is **nil**.

For a table of related items: See the section "Basic Forms for Bound Handlers".

---

**condition-case** (&rest varlist) form &rest clauses

*Special Form*

Binds handlers for conditions, expressing the handlers as clauses of a **case**-like construct instead of as functions. The handlers bound have dynamic scope.

Examples:

```lisp
(condition-case ()
  (time:parse string)
  (time:parse-error *default-time*))

(condition-case (e)
  (time:parse string)
  (time:parse-error
   (format *error-output* ""A, using default time instead." e)
   *default-time*))

(do () (nil)
  (condition-case (e)
    (return (time:parse string))
    (time:parse-error
     (setq string
       (prompt-and-read
        :string
        ""Use what time instead? " e)))))
```

The format is:

```
(condition-case (var1 var2 ...)
  form
  (condition-flavor-1 form-1-1 form-1-2 ... form-1-n)
  (condition-flavor-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (condition-flavor-m form-m-1 form-m-2 ... form-m-n))
```

Each **condition-flavor-j** is either a condition flavor, a list of condition flavors, or **:no-error**. If **:no-error** is used, it must be the last of the handler clauses. The remainder of each clause is a body, a list of forms constituting an implicit **progn**.

**condition-case** binds one handler for each clause. The handlers are bound simultaneously.
If a condition is signalled during the evaluation of form, the signalling mechanism examines the bound handlers in the order in which they appear in the definition, invoking the first appropriate handler.

**condition-case** normally returns the values returned by form. If a condition is signalled during the evaluation of form, the signalling mechanism determines whether the condition is one of the `condition-flavor-j`. If so, the following actions occur:

1. It automatically performs a `throw` to unwind the dynamic environment back to the point of the **condition-case**. This discards the handlers bound by the **condition-case**.

2. It executes the body of the corresponding clause.

3. It makes **condition-case** return the values produced by the last form in the handler clause.

While the clause is executing, `var1` is bound to the condition object that was signalled and the rest of the variables (`var2`, ...) are bound to `nil`. If none of the clauses needs to examine the condition object, you can omit `var1`.

```
(condition-case () ...)
```

As a special case, `condition-flavor-m` (the last one) can be the special symbol :no-error. If form is evaluated and no error is signalled during the evaluation, **condition-case** executes the :no-error clause instead of returning the values returned by form. The variables `var1`, `var2`, and so on are bound to the values produced by form, in the style of multiple-value-bind, so that they can be accessed by the body of the :no-error case. Any extra variables are bound to `nil`.

When an event occurs that none of the cases handles, the signalling mechanism continues to search the dynamic environment for a handler. You can provide a case that handles any error condition by using error as one `condition-flavor-j`.

The conditional variant of **condition-case** is the form:

**condition-case-if**

For a table of related items: See the section "Basic Forms for Bound Handlers".

```condition-case-if cond (&rest varlist) form &rest clauses```

Binds its handlers conditionally. In all other respects, it is just like **condition-case**. Its syntax includes `cond`, a subform that controls binding handlers:

```
(condition-case-if cond (var)
  form
  (condition-flavor-1 form-1-1 form-1-2 ... form-1-n)
  (condition-flavor-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (condition-flavor-m form-m-1 form-m-2 ... form-m-n))
```
condition-case-if first evaluates cond. If the result is nil, it does not set up any handlers; it just evaluates the form. If the result is not nil, it continues just like condition-case, binding the handlers and evaluating the form.

The :no-error clause applies whether or not cond is nil.

For a table of related items: See the section "Basic Forms for Bound Handlers".

dbg:condition-handled-p condition

Function

Searches the bound handler list and the default handler list to see whether a handler exists for the condition object, condition. This function should be called only from a condition-bind handler function. It starts looking from the point in the lists from which the current handler was invoked and proceeds to look outwards through the bound handler list and the default handler list. It returns a value to indicate what it found:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:maybe</td>
<td>condition-bind handlers for the flavor exist. These handlers are permitted to decline to handle the condition. You cannot determine what would happen without actually running the handler.</td>
</tr>
<tr>
<td>nil</td>
<td>No handler exists.</td>
</tr>
<tr>
<td>t</td>
<td>A handler exists.</td>
</tr>
</tbody>
</table>

conjugate number

Function

Returns the complex conjugate of number. If number is complex, then conjugate returns a complex with the same real part as number, and with imaginary part the negation of number's imaginary part. A non-complex argument number is returned. The conjugate of a noncomplex number is itself. conjugate could have been defined by:

```
(defun conjugate (number)
  (complex (realpart number) (- (imagpart number))))
```

For a table of related items, see the section "Arithmetic Functions".

:connected-p

Message

Returns t if the stream is fully connected to an active network connection, nil otherwise. If the stream is in a transitory state that is not completely connected, :connected-p returns nil.

:connected-p must be callable in a scheduler context. That is, it cannot call :process-wait.

cons x y

Function
Creates a new cons whose car is $x$ and whose cdr is $y$.

**cons** can be thought of as creating a cons or a list, or as adding a new element to the front of a list.

Examples:

- $(\text{cons } 'a 'b)$ => $(a . b)$
- $(\text{cons } 'a (\text{cons } 'b (\text{cons } 'c \text{ nil})))$ => $(a b c)$
- $(\text{cons } 'a('(b c d)))$ => $(a b c d)$

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**cons**

Type Specifier

This is the type specifier symbol for the predefined Lisp object **cons**.

The types **cons** and **null** form an exhaustive partition of the type **list**.

The types **cons**, **symbol**, **array**, **number**, and **character** are pairwise disjoint.

Examples:

- $(\text{typep } '((a.b) 'cons))$ => T
- $(\text{typep } '((a b c) 'cons))$ => T
- $(\text{zl: listp } '((a b c)))$ => T
- $(\text{subtypep } '\text{cons } '\text{list})$ => T and T
- $(\text{subtypep } '\text{list } '\text{cons})$ => NIL and T
- $(\text{sys: type-arglist } '\text{cons})$ => NIL and T
- $(\text{consp } '((a b c)))$ => T
- $(\text{type-of } '((\text{signed-byte 3}))))$ => CONS

See the section "Data Types and Type Specifiers". See the section "Type Specifiers and Type Hierarchy for Lists".

**cons-in-area** $x$ $y$ area

Function

Creates a cons, whose car is $x$ and whose cdr is $y$, in the specified area. (Areas are an advanced feature of storage management. See the section "Areas".)

Example:

- $(\text{cons-in-area } 'a 'b \text{ my-area})$ => $(a . b)$

**cons-in-area** is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**consp** object

Function
Returns \( t \) if its argument is a cons and \( \text{nil} \) otherwise. Thus, \texttt{consp} is the direct inverse of \texttt{atom}, that is, \((\texttt{consp } X)\) if and only if \((\texttt{not } (\texttt{atom } X))\). \((\texttt{consp } \text{nil})\) returns \( \text{nil} \) since \( \text{nil} \) is the empty list but not a cons. For this reason, \texttt{listp} should be used to determine whether or not an object is a list. If \texttt{consp} returns true for \( \text{object} \), the use of various functions that require a \texttt{cons} object, such as \texttt{car} and \texttt{cdr}, is legitimate.

For a table of related items: See the section "Predicates that Operate on Lists".

\begin{description}
\item[\texttt{constantp } object] \textit{Function} \\
This predicate is \( t \) if \( \text{object} \), when considered as a form to be evaluated, always evaluates to the same thing. This includes self-evaluating objects such as numbers, characters, strings, bit-vectors and keywords, as well as all constant symbols declared by \texttt{defconstant}, such as \texttt{nil}, \texttt{t}, and \texttt{pi}. In addition, a list whose \texttt{car} is \texttt{quote}, such as \((\texttt{quote rhumba})\) also returns \( t \) when it is given as \( \text{object} \) to \texttt{constantp}.

This predicate is \( \text{nil} \) if \( \text{object} \), considered as a form, may or may not always evaluate to the same thing.

If you are using CLOE, consider the following example:
\begin{verbatim}
(constantp '(quote foo)) => t
(constantp 'foo) => nil
(constantp (make-array foo '(2 3))) => t
\end{verbatim}

\item[\texttt{continue-whopper } \&\texttt{rest } \texttt{args}] \textit{Special Form} \\
Calls the combined method for the generic function that was intercepted by the whopper. Returns the values returned by the combined method.

\( \text{args} \) is the list of arguments passed to those methods. This function must be called from inside the body of a whopper. Normally the whopper passes down the same arguments that it was given. However, some whoppers might want to change the values of the arguments and pass new values; this is valid.

For more information on whoppers, including examples: See the section "Wrappers and Whoppers".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

\item[\texttt{copy-alist } \texttt{al} \&\texttt{optional } \texttt{area}] \textit{Function} \\
Returns an association list that is \texttt{equal} to \( \text{al} \), but is not \texttt{eq}. See the section "Association Lists". Only the top level of list structure is copied; that is, \texttt{copy-alist} copies in the \texttt{cdr} direction, but not in the \texttt{car} direction. Each cons of \( \text{al} \) is replaced in the copy by a new cons with the same \texttt{car} and \texttt{cdr}. See the function \texttt{copy-seq}. See the function \texttt{copy-tree}.
Here is an example of `copy-alist`:

```lisp
(copy-alist '((canoe.paddle) (rowboat.oar)))
```

returns the following association list, which is `equal` to the original association list:

```
((canoe.paddle) (rowboat.oar))
```

The optional `area` argument is the number of the area in which to create the new list. (Areas are an advanced feature of storage management. See the section "Areas".)

**Compatibility Note:** `area` is a Symbolics extension to Common Lisp. It is not supported under CLOE.

Example:

```lisp
(setq alist-1 (pairlis '(a b c d) '(1 2 3 4)))
=> ((A . 1)(B . 2)(C . 3)(D . 4))

(setq alist-2 (copy-alist alist))
=> ((A . 1)(B . 2)(C . 3)(D . 4))

(setf (cdr (assoc 'a alist-1)) 42)
=> 42

(assoc 'a alist-1)
=> (A . 42)

(assoc 'a alist-2)
=> (A . 1)
```

This function is specifically intended for copying association lists, that is, a-lists consisting of a list of conses.

For a table of related items: See the section "Functions for Copying Lists".

**copy-array-contents** *from-array to-array*  

*Function*

Copies the contents of `from-array` into the contents of `to-array`, element by element. `from-array` and `to-array` must be arrays. If `to-array` is shorter than `from-array`, the rest of `from-array` is ignored. If `from-array` is shorter than `to-array`, the rest of `to-array` is filled with `nil` if it is a general array, or 0 if it is a numeric array or `(code-char 0)` for strings. This function always returns `t`.

Note that even if `from-array` or `to-array` has a leader, the whole array is used; the convention that leader element 0 is the "active" length of the array is not used by this function. The leader itself is not copied.

`copy-array-contents` works on multidimensional arrays. `from-array` and `to-array` are "linearized" and row-major order is used. See the section "Row-major Storage of Arrays".

`copy-array-contents` does not work on conformally displaced arrays.
copy-array-contents-and-leader from-array to-array

Copy the contents and leader of from-array into the contents of to-array, element by element. copy-array-contents copies only the main part of the array.

copy-array-contents-and-leader does not work on conformally displaced arrays.

For a table of related items: See the section "Copying an Array".

copy-array-portion from-array from-start from-end to-array to-start to-end

Copies the portion of the array from-array with indices greater than or equal to from-start and less than from-end into the portion of the array to-array with indices greater than or equal to to-start and less than to-end, element by element. If there are more elements in the selected portion of to-array than in the selected portion of from-array, the extra elements are filled with the default value as by copy-array-contents. If there are more elements in the selected portion of from-array, the extra ones are ignored. Multidimensional arrays are treated the same way as copy-array-contents treats them. This function always returns t.

copy-array-portion does not work on conformally displaced arrays.

This function copies one element at a time in increasing order of subscripts. This means that when copying from and to the same array, the results might be unexpected if from-start is less than to-start. You can safely copy from and to the same array as long as from-start >= to-start.

For a table of related items: See the section "Copying an Array".

zl:copy-closure closure

Use the Symbolics Common Lisp function copy-dynamic-closure, which is equivalent to the function zl:copy-closure.

Creates and returns a new closure by copying the dynamic closure closure. zl:copy-closure generates new external value cells for each variable in the closure and initializes their contents from the external value cells of closure.

See the section "Dynamic Closure-Manipulating Functions".

copy-dynamic-closure closure

Creates and returns a new closure by copying the dynamic closure closure. copy-dynamic-closure generates new external value cells for each variable in the closure and initializes their contents from the external value cells of closure.

See the section "Dynamic Closure-Manipulating Functions".

sys:copy-if-necessary thing &optional (default-cons-area sys:working-storage-area)


Moves thing from a temporary storage area, or stack list, to a permanent area. Thing can be a string, symbol, list, tree, or &rest argument. sys:copy-if-necessary checks whether thing is in a temporary area of some kind, and moves it if it is. If thing is not in a temporary area, it is simply returned. The copy has the same type and dimensionality as the source.

This function is used especially for &rest arguments, which are not guaranteed to be in permanent storage. Sometimes the rest-argument list is stored in the function-calling stack, and loses its validity when the function returns. If you wish to return a rest-argument or make it part of a permanent list structure, you must copy it first, as you must always assume that it is one of these special lists.

Use sys:copy-if-necessary to copy a list if your only purposes are:

- To preserve a (possibly) stack-consed list outside of its stack extent.
- To copy an object in storage with dynamic extent, thus it is not suitable for guaranteeing that a given list does not share structure with any other list.

In all other cases, you should use copy-list, which copies unconditionally, thus it is suitable for making a private copy of a list. copy-list copies only lists, while sys:copy-if-necessary copies trees of conses as well as copying several other object types.

See the section "Lambda-List Keywords".

sys:copy-if-necessary is a Symbolics extension to Common Lisp.

For more information on stack lists: See the section "Consing Lists on the Control Stack". See the function with-stack-list.

For more information on temporary storage areas, see the :gc keyword of make-area. See the function make-area.

For a table of related items: See the section "Functions for Copying Lists".

**copy-list** list &optional area force-dotted

Returns a list that is equal to list, but not eq. Under Genera, the returned list is fully cdr-coded, to minimize storage. (See the section "Cdr-Coding").

Only the top level of the list structure is copied; that is, copy-list copies in the cdr direction, but not in the car direction. Each element of list that is a cons is replaced in the copy by a new cons with the same car and cdr. See also:

- copy-alist
- copy-seq
- copy-tree
- copy-tree-share

Compatibility Note: The optional arguments area and forced-dotted are Symbolics extensions to Common Lisp. Area is the number of the area in which to create the
new list. (Areas are an advanced feature of storage management. See the section "Areas"). Note that these options are not supported under CLOE.

Example:

```
(copy-list '(heron loon sandpiper))
```

Returns the following list, which is equal to list, but not eq:

```
(heron loon sandpiper)
```

Example:

```
(setq a '(one (two-a two-b)))
(setq b (list 1 a 'three))
=> (1 (ONE (TWO-A TWO-B)) THREE)
(setq c (copy-list b))
=> (1 (ONE (TWO-A TWO-B)) THREE)
(eq (last b) (last c)) => nil
(eq (cdr b) (cdr c)) => nil
(eq (cadr b) (cadr c)) => t
```

For a table of related items: See the section "Functions for Copying Lists".

**copy-list* list &optional area**

Returns a list that is equal to list, but not eq, and whose last cons is never cdr-coded.

See the function copy-list. See the section "Cdr-Coding". This increases efficiency if you add something onto the list later with nconc.

The optional area argument is the number of the area in which to create the new list. (Areas are an advanced feature of storage management. See the section "Areas").

**copy-list* is a Symbolics extension to Common Lisp.**

For a table of related items: See the section "Functions for Copying Lists".

**copy-readtable &optional (from-readtable *readtable*) to-readtable**

A copy is made of from-readtable, which defaults to the current readtable (the value of the global variable *readtable*). If from-readtable is nil, then a copy of a standard Common Lisp readtable is made. For example,

```
(setq *readtable* (copy-readtable nil))
```

will restore the input syntax to standard Common Lisp syntax, even if the original readtable has been clobbered.

If to-readtable is unsupplied or nil, a fresh copy is made. Otherwise, to-readtable must be a readtable, which is destructively copied into.
(let* ((foo "zzz\\"zzz")
    (newrt (copy-readtable))
    (*readtable* newrt)
    (result (read-from-string foo)))
  (set-syntax-from-char \" #\%
  (setq result (cons result (read-from-string foo))))
=> (ZZZ . [ZZZ"ZZZ])

zl:copy-readtable &optional from-readtable to-readtable

Function

from-readtable, which defaults to the current readtable, is copied. If to-readtable is unsupplied or nil, a fresh copy is made. Otherwise to-readtable is clobbered with the copy. Use zl:copy-readtable to get a private readtable before using the other readable functions to change the syntax of characters in it. The value of zl:readtable at the start of a session is the initial standard readtable, which usually should not be modified.

copy-seq sequence &optional area

Function

Non-destructively copies the argument sequence. Returns a new sequence which is equalp to the argument, but not eq. The function copy-seq returns the same result as the function subseq, when the value of the start argument of subseq is 0.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:

(setq name "Bill") => "Bill"

(setq a-copy (copy-seq name)) => "Bill"

a-copy => "Bill"

name => "Bill"

(equalp a-copy name) => T

(eq a-copy name) => NIL

Compatibility Note: The optional area argument is the number of the area in which to create the new alist. (Areas are an advanced feature of storage management.) area is a Symbolics extension to Common Lisp and is not supported by CLOE. See the section "Areas".

In the following example, copy-seq makes a copy of a sequence before destructively operating with replace.
(setq dated-copy (vector (get-name) (get-date) 123 456 987))

=> (SALLY 1-AUG-89 123 456 987)

(replace (copy-seq dated-copy) #((get-date) 321 654 789) :start1 1)

=> (SALLY 2-AUG-89 321 654 789)

dated-copy => (SALLY 1-AUG-89 123 456 987)

For a table of related items: See the section "Sequence Construction and Access".

copy-symbol symbol &optional copyprops Function

Returns a new uninterned symbol with the same print-name as symbol. If copyprops is non-nil, then the value and function-definition of the new symbol are the same as those of sym, and the property list of the new symbol is a copy of symbol’s. If copyprops is nil (the default), then the new symbol is unbound and undefined, and its property list is empty.

(copysymbol symbol nil) = (make-symbol (symbol-name symbol))

See the section "Functions for Creating Symbols".

copy-tree tree &optional area Function

Copies a tree of conses. The argument tree can be any Lisp object. If it is not a cons, it is returned; otherwise the result is a new cons made from the results of calling copy-tree on the car and cdr of the argument. In other words, all conses in the tree are copied recursively, stopping only when non-conses are encountered. Circularities and the sharing of substructure are not preserved. The optional area argument is the number of the area in which to create the new tree. (Areas are an advanced feature of storage management. See the section "Areas".)

area is a Symbolics extension to Common Lisp, and is not available in CLOE.

Example:

(copy-tree '((freesia) (carnation) (rose)))

returns the following tree:

((freesia) (carnation) (rose))

In the following example, we have an association list whose components are pairs of keys and association lists. A call to copy-alist only provides a true copy of the top level association list and not of the lower level a-list.
(setq keys '(monthly-cash-on-hand monthly-expense monthly-revenue))
(setq data '((pairlis '(11 12) '(52 73))
               (pairlis '(10 11) '(20 21))
               (pairlis '(10 11) '(31 42))))
(setq financial-statement (pairlis keys data))

The function what-if, defined in the following example, executes coordinated changes in the low-level association lists. These changes are made on a trial basis, and copy-tree allows the changes to occur in a copy of the data-base rather than the data base itself.

(defun what-if (a-list, revenue)
  (let ((november-cash-on-hand
         (assoc '11 (assoc 'monthly-cash-on-hand a-list)))
        (november-expense
         (assoc '11 (assoc 'monthly-expense a-list)))
        (november-revenue revenue)
        (december-cash-on-hand 0))
    (setf (cdr (assoc '11 (assoc 'monthly-revenue a-list)))
          november-revenue)
    (setq december-cash-on-hand
          (+ november-cash-on-hand (- november-revenue november-expense)))
    (setf (cdr (assoc '12 (assoc 'monthly-cash-on-hand a-list)))
          december-cash-on-hand)
    december-cash-on-hand))

(what-if (copy-tree financial-statement) 40) => 71

(assoc '12 (assoc 'monthly-cash-on-hand financial-statement))
  => (12 . 73)

For a definition and diagram of a tree: See the section "Overview of Lists".

For a table of related items: See the section "Functions for Copying Lists".

copy-tree-share tree &optional area (hash (make-hash-table :test #z!equal)) cdr-code

Function

Similar to copy-tree, it makes a copy of an arbitrary structure of conses, copying at all levels, and optimally cdr-coding. However, it also assures that all lists, or tails of lists, are optimally shared when equal.

The arguments for copy-tree-share are: the tree to be copied, and an optional storage area, an externally created hash table to be used for the equality testing and a cdr-code, which is the storage location of the conses that compose a tree or list. The default storage area for the new list is the area occupied by the old list. If cdr-code is t, lists will never be "forked" to enable sharing a tail. This wastes space, but improves locality.

Note: copy-tree-share might be very slow, in the general case, for long lists. However, applying it at the appropriate level of a specific structure-copying routine
(furnishing a common, externally created hash table) is likely to yield all the sharing possible, at a much lower computational cost. For example, **copy-tree-share** could be applied only to the branches of a long association list.

Example:

```
(copy-tree-share '((1 2 3) (1 2 3) (0 1 2 3) (0 2 3)))
```

If \( x = '(1 2 3) \), the above returns (roughly):

```
'(,(x ,x (0 . ,x) (0 . ,(cdr x))))
```

**copy-tree-share** is a Symbolics extension to Common Lisp.

---

**zl:copyalist** \( al \) &optional \( area \) 

In your new programs, we recommend that you use the function **copy-alist** which is the Common Lisp equivalent of the function **zl:copyalist**.

Copies an association list. Returns a list that is **zl:equal** to \( al \), but not **eq**. Each element of \( al \) that is a cons is replaced in the copy by a new cons with the same car and cdr. You can optionally specify the area in which to create the new copy. The default is to copy the new list into the area occupied by the old list.

For a table of related items: See the section "Functions for Copying Lists".

---

**zl:copylist** \( list \) &optional \( area \) force-dotted 

In your new programs we recommend that you use the function **copy-list** which is the Common Lisp equivalent of the function **zl:copylist**.

Returns a list that is **zl:equal** to \( list \), but not **eq**. **zl:copylist** does not copy any elements of the list: only the conses of the list itself. The returned list is fully cdr-coded, to minimize storage. See the section "Cdr-Coding": If the list is "dotted", that is, \( (cdr \ (last \ list)) \) is a non-nil atom, this is true of the returned list also. You can specify the area in which to create the new copy. The default is to copy the new list into the area occupied by the old list.

For a table of related items: See the section "Functions for Copying Lists".

---

**zl:copylist** \( list \) &optional \( area \) 

Use the Common Lisp function **copy-list***, which is equivalent to **zl:copylist***. Returns a list that is **zl:equal** to \( list \), but not **eq**. **zl:copylist** does not copy any elements of the list: only the conses of the list. The last cons of the resulting list is never cdr-coded. See the function **zl:copylist**. See the section "Cdr-Coding". This increases efficiency, if you add something onto the list later using **nconc**.

For a table of related items: See the section "Functions for Copying Lists".

---

**zl:copysymbol** \( symbol \) &optional \( copyprops \)
Function

Use the Common Lisp function **copy-symbol**, which is equivalent to **zl:copysymbol**.

Returns a new uninterned symbol with the same print-name as `symbol`. If `copyprops` is non-nil, then the value and function-definition of the new symbol are the same as those of `sym`, and the property list of the new symbol is a copy of `symbol`'s. If `copyprops` is nil (the default), then the new symbol is unbound and undefined, and its property list is empty.

\[
(\text{copy-symbol } \text{symbol } \text{nil}) = (\text{make-symbol } \text{symbol-name } \text{symbol})
\]

See the section "Functions for Creating Symbols".

\[
\text{zl:copytree tree } \&\text{optional area}
\]

Function

In your new programs, we recommend that you use the function **copy-tree**, which is the Common Lisp equivalent of the function **zl:copytree**.

Copies all the conses of a tree and makes a new tree with the same fringe. You can specify the area in which to create the new copy. The default is to copy the new list into the area occupied by the old list.

For a table of related items: See the section "Functions for Copying Lists".

\[
\text{zl:copytree-share tree } \&\text{optional area (hash } \text{cl:make-hash-table } \text{test } \#\text{equal}
\text{ :locking nil :number-of-values 0)) cdr-code}
\]

Function

Use the Symbolics Common Lisp function **copy-tree-share**, which is equivalent to **zl:copytree-share**.

**zl:copytree-share** is similar to **zl:copytree**: it makes a copy of an arbitrary structure of conses, copying at all levels, and optimally cdr-coding. However, it also assures that all lists or tails of lists are optimally shared when **zl:equal**.

**zl:copytree-share** takes as arguments the tree to be copied, and optionally a storage area, an externally created hash table to be used for the equality testing and a `cdr-code`, which is the storage location of the conses that compose a tree or list. The default storage area for the new list is the area occupied by the old list. If `cdr-code` is t, lists will never be "forked" to enable sharing a tail. This wastes space, but improves locality.

**Note**: **zl:copytree-share** might be very slow, in the general case, for long lists. However, applying it at the appropriate level of a specific structure-copying routine (furnishing a common, externally created hash table) is likely to yield all the sharing possible, at a much lower computational cost. For example, **zl:copytree-share** could be applied only to the branches of a long alist.

Example:

\[
(\text{zl:copytree-share } '((1 2 3) (1 2 3) (0 1 2 3) (0 2 3)))
\]
If \( x = '(1 2 3) \), the above returns (roughly):
\[
'(\,x\,\,x\,(0\,\,x)\,(0\,\,(\text{cdr } x)))
\]
For a table of related items: See the section "Functions for Copying Lists".

### si:coroutine-bidirectional-stream

**Flavor**

A flavor implementing bidirectional coroutine streams. Defines :next-input-buffer, :new-output-buffer, and :send-output-buffer methods. Use this to construct a bidirectional stream to a function written in terms of input and output operations.

### si:coroutine-input-stream

**Flavor**

A flavor implementing input coroutine streams. Defines a :next-input-buffer method. Use this to construct an input stream from a function written in terms of output operations.

### si:coroutine-output-stream

**Flavor**

A flavor implementing output coroutine streams. Defines :new-output-buffer and :send-output-buffer methods. Use this to construct an output stream to a function written in terms of input operations.

### cos radians

**Function**

Returns the cosine of radians. radians can be of any numeric type.

Examples:

\[
\begin{align*}
(\cos \theta) & \Rightarrow 1.0 \\
(\cos \left(\frac{\pi}{2}\right)) & \Rightarrow -0.0d0 \\
(\cos \left(\frac{\pi}{4}\right)) & \Rightarrow 0.70710677
\end{align*}
\]

For a table of related items: See the section "Trigonometric and Related Functions".

### cosd degrees

**Function**

Returns the cosine of degrees. degrees can be of any numeric type.

Examples:

\[
\begin{align*}
(\cosd 90) & \Rightarrow -0.0 \\
(\cosd 45) & \Rightarrow 0.7071068 \\
(\cosd 36.2) & \Rightarrow 0.80696034
\end{align*}
\]

For a table of related items: See the section "Trigonometric and Related Functions".

### cosh radians

**Function**
Returns the hyperbolic cosine of radians.

Example:

\[(\cosh 0) \Rightarrow 1.0\]

For a table of related items: See the section "Hyperbolic Functions".

\[
\text{count} \quad \text{item} \quad \text{sequence} \quad \text{&key} \quad \text{(:test #eql) \ :test-not (key #identity) \ :from-end (start 0) \ :end}
\]

Function

Counts the number of elements in a subsequence of \textit{sequence} satisfying the predicate specified by the \textit{:test} keyword. \textit{count} returns a non-negative integer, which represents the number of elements in the specified subsequence of \textit{sequence}.

\textit{item} is matched against the elements specified by the \textit{test} keyword. \textit{item} can be any Symbolics Common Lisp object.

\textit{sequence} can be either a list or a vector (one-dimensional array). Note that \texttt{nil} is considered to be a sequence, of length zero.

\textit{:test} specifies the test to be performed. An element of \textit{sequence} satisfies the test if \((\text{funcall testfun item (keyfn x)})\) is true, where \textit{testfun} is the test function specified by \textit{:test}, \textit{keyfn} is the function specified by \textit{:key} and \textit{x} is an element of the sequence. The default test is \texttt{eql}.

For example:

\[(\text{count} \ 'a \ '((a b c d) (a b) (b c)) \ :\text{test-not} \ #'(\texttt{eql})) \Rightarrow 3\]

\textit{:test-not} is similar to \textit{:test}, except that the sense of the test is inverted. An element of \textit{sequence} satisfies the test if \((\text{funcall testfun item (keyfn x)})\) is false.

The value of the keyword argument \textit{:key}, if non-\texttt{nil}, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element. For example:

\[(\text{count} \ 'a \ '((a b) (a b) (b c)) \ :\text{key} \ #'(\texttt{car})) \Rightarrow 2\]

\[(\text{count} \ #'(1 2 3 4 1) \ :\text{key} \ #'(\texttt{lambda (x) (- x 1)))} \Rightarrow 1\]

The \textit{:from-end} argument does not affect the result returned; it is accepted purely for compatibility with other sequence functions. For example:

\[(\text{count} \ 'a \ '(a a a b c d) \ :\text{from-end} \ t \ :\text{start} \ 3) \Rightarrow 0\]

\[(\text{count} \ 'a \ '(a a a b c d) \ :\text{from-end} \ \texttt{nil} \ :\text{start} \ 3) \Rightarrow 0\]

For the sake of efficiency, you can delimit the portion of the sequence to be operated on by the keyword arguments \textit{:start} and \textit{:end}.

\textit{:start} and \textit{:end} must be non-negative integer indices into the sequence. \textit{:start} must be less than or equal to \textit{:end}, else an error is signalled. It defaults to zero (the start of the sequence).
:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(count 'a '(a b a)) => 2
(count 'heron '(heron loon heron pelican heron stork)) => 3
(count 'a '(a a b b a a) :start 1 :end 5) => 2
(count 'a '(a a b b a a) :start 1 :end 6) => 3
(count 'a #((a b b b a)) ) => 2

For a table of related items: See the section "Searching for Sequence Items".

count keyword for loop

(count expr {into var} {data-type})

If expr evaluates non-nil, a counter is incremented. The data-type defaults to fixnum. When the epilogue of the loop is reached, var has been set to the accumulated result and can be used by the epilogue code.

It is safe to reference the values in var during the loop, but they should not be modified until the epilogue code for the loop is reached.

The forms count and counting are synonymous.

Examples:

(defun num-entry (small-list)
 (loop for x in small-list
      count t into num
      finally (return num))) => NUM-ENTRY
(num-entry '(a b c d)) => 4

is equivalent to

(defun num-entry (small-list)
 (loop for x in small-list
       counting t into num
       finally (return num))) => NUM-ENTRY
(num-entry '(a b c d)) => 4

Not only can there be multiple accumulations in a loop, but a single accumulation can come from multiple places within the same loop form, if the types of the collections are compatible. count and sum are compatible.

See the section "Accumulating Return Values for loop".
count-if predicate sequence &key :key :from-end (start 0) :end

Function

Returns a non-negative integer, which represents the number of elements in the specified subsequence of sequence satisfying the predicate.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

(count-if #'atom '((a b) ((a) b) (nil nil)) :key #'car) => 2
(count-if #'zerop #(1 2 1) :key #'(lambda (x) (- x 1))) => 2

The :from-end argument does not affect the result returned; it is accepted purely for compatibility with other sequence functions.

For example:

(count-if #'oddp '(1 1 2 2) :start 2 :from-end t) => 0
(count-if #'oddp '(1 1 2 2) :start 2 :from-end nil) => 0

For the sake of efficiency, you can delimit the portion of the sequence to be operated on by the keyword arguments :start and :end.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(count-if #'oddp '(1 2 1 2)) => 2
(count-if #'oddp '(1 1 2 2 2) :start 2 :end 4) => 1
(count-if #'numberp '(heron 1.0 a 2 #\Space)) => 2

(setq pressure-readings '(1230 1400 :over-limit 1687))
(count-if #'(lambda (x) (eq x :over-limit)) pressure-readings) => 1

For a table of related items: See the section "Searching for Sequence Items".

count-if-not predicate sequence &key :key :from-end (start 0) :end
Function

Returns a non-negative integer, which represents the number of elements in the specified subsequence of `sequence` that do not satisfy the predicate.

`predicate` is the test to be performed on each element.

`sequence` can be either a list or a vector (one-dimensional array). Note that `nil` is considered to be a sequence, of length zero.

The value of the keyword argument `:key`, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```
(count-if-not #'atom '((a b) ((a) b) (nil nil)) :key #'car) => 1
```

```
(count-if-not #'zerop #(1 2 1) :key #'(lambda (x) (- x 1))) => 1
```

The `:from-end` argument does not affect the result returned; it is accepted purely for compatibility with other sequence functions.

For example:

```
(count-if-not #'oddp '(1 1 2 2) :start 2 :from-end t) => 2
```

```
(count-if-not #'oddp '(1 1 2 2) :start 2 :from-end nil) => 2
```

For the sake of efficiency, you can delimit the portion of the sequence to be operated on by the keyword arguments `:start` and `:end`.

`:start` and `:end` must be non-negative integer indices into the sequence. `:start` must be less than or equal to `:end`, else an error is signalled. It defaults to zero (the start of the sequence).

`:start` indicates the start position for the operation within the sequence. `:end` indicates the position of the first element in the sequence beyond the end of the operation. It defaults to `nil` (the length of the sequence).

If both `:start` and `:end` are omitted, the entire sequence is processed by default.

For example:

```
(count-if-not #'numberp '(heron 1.0 a 2 #\Space)) => 3
(count-if-not #'oddp '(3 4 3 4)) => 2
```

```
(setq pressure-readings '(1230 1400 :over-limit 1687))
(count-if-not #'(lambda(x) (numberp x)) pressure-readings) => 1
```

For a table of related items: See the section "Searching for Sequence Items".

`:creation-date`
Returns the creation date of the file, as a number which is a universal time. See the section "Dates and Times". See the function \texttt{fs:directory-list}.

\texttt{ctypecase} \texttt{object} \&\texttt{body body} \hfill \textit{Special Form}

\texttt{ctypecase} is similar to \texttt{typecase}, except that it does not allow an explicit \texttt{otherwise} or \texttt{t} clause, and if no clause is satisfied it signals a proceedable error instead of returning \texttt{nil}.

\texttt{ctypecase} is a conditional that chooses one of its clauses by examining the type of an object. Its form is as follows:

\begin{verbatim}
  (ctypecase form
    (types consequent consequent ...)
    (types consequent consequent ...)
    ...
  )
\end{verbatim}

First \texttt{ctypecase} evaluates \texttt{form}, producing an object. \texttt{ctypecase} then examines each clause in sequence. \texttt{types} in each clause is a type specifier in either symbol or list form, or a list of type specifiers. The type specifier is not evaluated. If the object is of that type, or of one of those types, then the consequents are evaluated and the result of the last one is returned (or \texttt{nil} if there are no consequents in that clause). Otherwise, \texttt{ctypecase} moves on to the next clause.

If no clause is satisfied, \texttt{ctypecase} signals an error with a message constructed from the clauses. To continue from this error, supply a new value for \texttt{object}, causing \texttt{ctypecase} to store that value and restart the type tests. Subforms of \texttt{object} can be evaluated multiple times.

For an object to be of a given type means that if \texttt{typep} is applied to the object and the type, it returns \texttt{t}. That is, a type is something meaningful as a second argument to \texttt{typep}. See the section "Data Types and Type Specifiers".

It is permissible for more than one clause to specify a given type, particularly if one is a subtype of another; the earliest applicable clause is chosen. Thus, for \texttt{ctypecase}, the order of the clauses can affect the behavior of the construct.

Examples:

\begin{verbatim}
  (defun tell-about-car (x)
    (ctypecase (car x)
      (string "string") => TELL-ABOUT-CAR
      (tell-about-car '(("word" "more")) => "string"
      (tell-about-car '(a 1)) => proceedable error is signalled
    )
  )
\end{verbatim}
(defun tell-about-car (x) ; see typecase
  (ctypecase (car x)
    (fixnum "number.")
    ((or string symbol) "string or symbol.")
    (otherwise "I don’t know.")
  )) => TELL-ABOUT-CAR
(tell-about-car '(1 a))  => "number."
(tell-about-car '(a 1))  => "string or symbol."
(tell-about-car '("word" "more") )  => "string or symbol."
(tell-about-car '(1.0))   => "I don’t know."

For a table of related items: See the section "Conditional Functions".
For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

zl:cursorpos &rest args

This function exists primarily for Maclisp compatibility. It performs operations related to the cursor position, such as returning the position, moving the position, or performing another cursor operation.

zl:cursorpos normally operates on the zl:standard-output stream; however, if the last argument is a stream or t (meaning zl:terminal-io), zl:cursorpos uses that stream and ignores it when doing the operations described below. Note that zl:cursorpos works only on streams that are capable of these operations, such as windows. A stream is taken to be any argument that is not a number and not a symbol, or a symbol other than nil with a name more than one character long.

(zl:cursorpos) => (line . column), the current cursor position.
(cursorpos line column) moves the cursor to that position. It returns t if it succeeds and nil if it does not.
(cursorpos op) performs a special operation coded by op and returns t if it succeeds and nil if it does not. op is tested by string comparison, is not a keyword symbol, and can be in any package.

F        Moves one space to the right.
B        Moves one space to the left.
D        Moves one line down.
U        Moves one line up.
T        Homes up (moves to the top left corner). Note that t as the last argument to zl:cursorpos is interpreted as a stream, so a stream must be specified if the t operation is used.
Z        Homes down (moves to the bottom left corner).
A        Advances to a fresh line. See the :fresh-line stream operation.
C Clears the window.
E Clears from the cursor to the end of the window.
L Clears from the cursor to the end of the line.
K Clears the character position at the cursor.
X B then K.

**sys:debug-instance**  \textit{instance}  \textbf{Function}

Enters the Debugger in the lexical environment of \textit{instance}. This is useful in debugging. You can examine and alter instance variables, and run functions that use the instance variables.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

\textbf{*debug-io*}  \textbf{Variable}

The value of \textbf{*debug-io*} is a stream to be used for interactive debugging purposes. In CLOE-Runtime, \textbf{*debug-io*} is initially a synonym stream of \textbf{terminal-io*}.

\begin{verbatim}
(format *debug-io* "Return to top level?"
(if (positive-response (read *debug-io*)))))
\end{verbatim}

\textbf{zl:debug-io}  \textbf{Variable}

In your new programs, we recommend that you use the variable \textbf{*debug-io*}, which is the Common Lisp equivalent of \textbf{zl:debug-io*}. If not \textbf{nil}, this is the stream that the Debugger should use. The default value is a synonym stream that is synonymous with \textbf{zl:terminal-io}. If the value of \textbf{dbg:*debug-io-override*} is not \textbf{nil}, the Debugger uses the value of that variable as the stream instead of the value of \textbf{zl:debug-io*}.

The value of \textbf{zl:debug-io*} can also be a string. This causes the Debugger to use the cold-load stream; the string is the reason why the cold-load stream should be used.

No program other than the Debugger should do stream operations on the value of \textbf{zl:debug-io*}, since the value cannot be a stream. Other programs should use \textbf{zl:query-io}, \textbf{zl:error-output}, or \textbf{zl:trace-output}. \textbf{zl:debug-io} is equivalent to \textbf{*debug-io*}.

\textbf{dbg:*debugger-bindings*}  \textbf{Variable}
When the Debugger is entered, it binds some special variables under control of the list that is the value of `dbg:*debugger-bindings*`. Each element of the list is a list of two elements: a variable and a form that is evaluated to produce the value to bind it to. The bindings happen sequentially. You can push things on this list (adding to the front of it), but should not replace the list wholesale since several of the variable bindings on this list are essential to the operation of the Debugger.

**debugging-info function**

Returns the debugging info alist of `function`. Most of the elements of this alist are an internal interface between the compiler and the Debugger.

**decf access-form &optional amount**

Macro

Decrements the value of a generalized variable. `(decf ref)` decrements the value of `ref` by 1. `(decf ref amount)` subtracts `amount` from `ref` and stores the difference back into `ref`. It returns the new value of `ref`.

`access-form` can be any form acceptable to `setf`.

```
(decf (car (mumble)))
```

is almost equivalent to

```
(setf (car (mumble)) (1- (car (mumble))))
```

except that while the latter would evaluate `mumble` twice, `decf` actually expands into a `let` and `mumble` is evaluated only once.

```
(setq arr (make-array (4) :element-type 'integer
:initial-element 5))

(decf (aref arr 3) 4) => #(5 5 5 1)
```

See the section "Generalized Variables".

**declaration name1 name2 ...**

Declaration

Tells the compiler that the `names` given are valid but non-standard declarations so the compiler does not issue warnings about them. This allows you to put declarations meant for another compiler or another program processor into your program. `declaration` can only be used with `proclaim`.

See the section "Declaration Specifiers".

**declare &rest forms**

Special Form

Provides additional information to the Lisp system (interpreter and compiler).

The `declare` special form can be used in two ways: at top level or within function bodies. For information on top-level `declare` forms: See the section "How the Stream Compiler Handles Top-level Forms".
**declare** forms that appear within function bodies provide information to the Lisp system (for example, the interpreter and the compiler) about this particular function. Expressions appearing within the function-body **declare** are declarations; they are not evaluated. **declare** forms must appear at the front of the body of certain special forms, such as **let** and **defun**. Some declarations apply to function definitions and must appear as the first forms in the body of that function; otherwise they are ignored.

See the section "Function-body Declarations".

The compiler also recognizes any number of **declare** forms as the first forms in the bodies of the following macros and special forms. This means that you can have **special** declarations that are local to any of these blocks. In addition, declarations can appear at the front of the body of a function definition, like **defun**, **defmacro**, **defsubst**, and so on.

<table>
<thead>
<tr>
<th>destructuring-bind</th>
<th>multiple-value-bind</th>
</tr>
</thead>
<tbody>
<tr>
<td>let</td>
<td>let*</td>
</tr>
<tr>
<td>do</td>
<td>do*</td>
</tr>
<tr>
<td>zl:do-named (not in CLOE)</td>
<td>zl:do*-named (not in CLOE)</td>
</tr>
<tr>
<td>prog</td>
<td>prog*</td>
</tr>
<tr>
<td>lambda</td>
<td></td>
</tr>
</tbody>
</table>

See the section "Operators for Making Declarations".

**decode-float float**  

*Determine the significand, the exponent, and the sign corresponding to the floating-point argument float. The argument float is equal to:*

\[
(* \text{sign significand} (\text{expt (float-radix sign) exponent}))
\]

The significand is returned as a floating-point number of the same format as float. It is obtained by dividing the argument by an integral power of 2, the radix of the floating-point representation, so as to bring its value between 1/2 (inclusive) and 1 (exclusive). The quotient is then returned as the significand.

The second result of **decode-float** is the integer exponent e to which 2 must be raised to produce the appropriate power for the division.

The third result is a floating-point number, of the same format as the argument, whose absolute value is one and whose sign matches that of the argument.

Examples:
(decode-float 2.0) => 0.5 and 2 and 1.0
(decode-float -2.0) => 0.5 and 2 and -1.0
(decode-float 4.0) => 0.5 and 3 and 1.0
(decode-float 8.0) => 0.5 and 4 and 1.0
(decode-float 3.0) => 0.75 and 2 and 1.0
(decode-float 0.0) => 0.0 and 0 and 1.0
(decode-float -0.0) => 0.0 and 0 and -1.0
(decode-float 5.06) \rightarrow 0.6325 3 1.0
;;; a possible use of decode-float
;;; (log-\text{abs} float)≡(log (abs float))

(defun log-\text{abs} (float)
  (multiple-value-bind (significand exponent)
      (decode-float float)
    (+ (log significand) ; \text{log} ab= \text{log} a + \text{log} b
       (* exponent (log 2)))) ; \text{log (expt x y)}= y \text{log} x

(log-\text{abs} 2.0) => 0.6931472 ;(log 2) => 0.6931472

For a table of related items, see the section "Functions that Decompose and Construct Floating-point Numbers".

decode-raster-array \texttt{raster} \textbf{Function}

Returns the following attributes of the raster as values: width, height, and spanning width. In a row-major implementation, width and height are the second and first dimensions, respectively. The spanning width is the number of linear array elements needed to go from \((x,y)\) to \((x,y+1)\). For nonconformal arrays, this is the same as the width. For conformal arrays, this is the width of the underlying array that provides the storage adjusted for possibly differing numbers of bits per element.

decode-raster-array should be used rather than \texttt{array-dimensions}, \texttt{zl:array-dimension-n}, or \texttt{sys:array-row-span} for the following reasons.

\begin{itemize}
  \item \texttt{decode-raster-array} does error checking by ensuring that the array is two-dimensional.
  \item A single call to \texttt{decode-raster-array} is faster than any non-null combination of the alternatives.
  \item \texttt{decode-raster-array} always returns the \textit{width} and \textit{height}, which are not the first and second dimensions as returned by \texttt{array-dimensions} or \texttt{zl:array-dimension-n}.
\end{itemize}

For a table of related items: See the section "Operations on Rasters".

\texttt{math:decompose \textit{a} \&optional \textit{lu} \textit{ps} \textit{ignore} \textbf{Function}
Computes the LU decomposition of matrix $a$. If $lu$ is non-nil, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. The lower triangle of $lu$, with ones added along the diagonal, is $L$, and the upper triangle of $lu$ is $U$, such that the product of $L$ and $U$ is $a$. Gaussian elimination with partial pivoting is used. The $lu$ array is permuted by rows according to the permutation array $ps$, which is also produced by this function. If the argument $ps$ is supplied, the permutation array is stored into it; otherwise, an array is created to hold it. This function returns two values: the LU decomposition and the permutation array.

```
def function &rest defining-forms
```

Special Form

If a function is created in some strange way, wrapping a `def` special form around the code that creates it informs the editor of the connection. The form:

```
(def function-spec
 form1 form2 ...)
```

simply evaluates the forms $form1$, $form2$, and so on. It is assumed that these forms create or obtain a function somehow, and make it the definition of $function-spec$.

Alternatively, you could put `(def function-spec)` in front of or anywhere near the forms that define the function. The editor only uses it to tell which line to put the cursor on.

```
clos:defclass class-name superclasses slot-specifiers &rest class-options
```

Macro

Defines a class named $class-name$, and returns the class object.

If a class already exists with that name, then the existing class is redefined. A redefined class is `eq` to the original class. See the section "Redefining a CLOS Class".

$class-name$ A symbol naming the class.

$superclasses$ A list of class names. The new class inherits slots and other characteristics from each of its superclasses. See the section "CLOS Inheritance".

$slot-specifiers$ Each slot-specifier is one of the following:

```
slot-name
(slot-name slot-options ...)
```

The $slot-options$ are:

:reader $reader-name$

Defines a method for a reader generic function named $reader-name$. The reader takes a single argument (an object that is a member of this class), and returns the value of this slot.
:writer  writer-name
Defines a method for a writer generic function named writer-name. The writer takes two arguments (the new value, and an object that is a member of this class), and sets the value of this slot. writer-name can be a symbol or a list of the form (future-common-lisp:setf symbol). The following examples show the calling syntax in the two cases:

;;; if the CLOS writer's name is a symbol
(writer-name new-value instance)

;;; if the CLOS writer's name is (clos:setf symbol)
(setf (symbol instance) new-value)

Note that when defining a writer method in CLOS to use the setf syntax, the function spec must be (future-common-lisp:setf symbol). However, when calling the writer generic function, you can use either setf or future-common-lisp:setf.

:accessor  reader-name
Defines a method for a reader generic function named reader-name, and a method for a writer named (future-common-lisp:setf reader-name).

:locator  locator-name
This is a Symbolics CLOS extension, which is supported on 3600-family and Ivory-based machines only. This option defines a method for a locator generic function which enables you to get a locative pointer to the cell inside an instance that contains the value of a slot. locator-name can be a symbol or a list of the form (locf symbol). In the latter case, the locator is called with locf syntax:

(locf (symbol object))

:allocation  allocation-type
Defines the allocation type of the slot. If allocation-type is :instance, then a local slot is defined. If allocation-type is :class, then a shared slot is defined. If the :allocation option is not provided, the slot will be a local slot.

A local slot means that each instance of the class stores its own value for the slot. In other words, the storage for the slot is allocated on a per-instance basis.

A shared slot means that all members of the class share the value of the slot. The storage for the slot is allocated only once.
Both local and shared slots are inherited: See the section "Inheritance of Slots and clos:defclass Options".

:initform form
Provides a default initial value for the slot. When a new instance is created, the initform is used if the slot is not initialized in some other way, such as by providing an initialization argument in the call to clos:make-instance that initializes the slot. The form is evaluated each time it is used, in the same lexical environment in which the clos:defclass form was evaluated. For local slots, the form is evaluated in the dynamic environment in which clos:make-instance was called; for shared slots, it is evaluated in the dynamic environment in which the clos:defclass form was evaluated.

:initarg initarg-name
Provides a means to initialize the slot in a call to clos:make-instance. This slot option declares the initarg-name as a valid initialization argument to clos:make-instance. If you provide the initarg-name and a value in a call to clos:make-instance, the slot is initialized with that value. This overrides the slot’s initform.

:type type-specifier
Declares that the value of the slot is of the type type-specifier. Symbolics CLOS ignores this option.

:documentation string
Provides a documentation string describing the slot.

The following slot options may be given more than once for a single slot: :reader, :writer, :accessor, :locator, and :initarg. If any other slot option is given more than once for a single slot, an error is signaled.

class-options
Options that pertain to the class as a whole. The class-options are:

(:default-initargs initarg-list)
The initarg-list is a list of alternating initialization argument names and default initial value forms. If an initialization argument name is not provided in a call to clos:make-instance, and it does appear in the :default-initargs initarg-list, the default value form is evaluated and used. The form is evaluated in the same lexical environment as that in which the clos:defclass was evaluated, and in the same dynamic environment in which clos:make-instance was called. An error is signaled if an initialization argument name appears more than once in the initarg-list.
Provides a documentation string describing the class. You can get the documentation string of a class as follows:

\[
\text{(documentation class-name 'type)}
\]

Specifies the class of the class being defined. The default is `clos:standard-class`. In Symbolics CLOS, the effects are undefined if any other value is given to this option.

The `:default-initargs`, `:documentation`, and `:metaclass` class options may not be given more than once.

See the section "Inheritance of Slots and `clos:defclass` Options".

See the section "CLOS Class Precedence List".

:\defconst variable initial-value &optional documentation

The same as `defvar`, except that `variable` is always set to `initial-value` regardless of whether `variable` is already bound. The rationale for this is that `defvar` declares a global variable, whose value is initialized to something but is then changed by the functions that use it to maintain some state. On the other hand, `zl:defconst` declares a constant, whose value is never changed by the normal operation of the program, only by changes to the program. `zl:defconst` always sets the variable to the specified value so that if, while developing or debugging the program, you change your mind about what the constant value should be, and you then evaluate the `zl:defconst` form again, the variable gets the new value. It is not the intent of `zl:defconst` to declare that the value of `variable` never changes; for example, `zl:defconst` is not license to the compiler to build assumptions about the value of `variable` into programs being compiled. See `defconstant` for that.

See the section "Special Forms for Defining Special Variables".

:\defconstant variable initial-value &optional documentation

Declares the use of a named constant in a program. Additionally, `defconstant` indicates that the value of the constant remains the same. `initial-value` is evaluated and `variable` set to the result. The value of `variable` is then fixed. It is an error if `variable` has any special bindings at the time the `defconstant` form is executed. Once a special variable has been declared constant by `defconstant`, any further assignment to or binding of that variable is an error.

The compiler is free to build assumptions about the value of the variable into programs being compiled. If the compiler does replace references to the name of the constant by the value of the constant in code to be compiled, the compiler takes care that such "copies" appear to be `eql` to the object that is the actual value of the constant. For example, the compiler can freely make copies of numbers, but it exercises care when the value is a list.
In Symbolics Common Lisp, `defconstant` and `zl:defconst` are essentially the same if the value is other than a number, a character, or an interned symbol. However, if the variable being declared already has a value, `zl:defconst` freely changes the value, whereas `defconstant` queries before changing the value. `defconstant`'s query offers three choices: Y, N, and P.

- The Y option changes the value.
- The N option does not change the value.
- The P option changes the value and when you change any future value, prints a warning rather than a query.

The P option sets `sys:inhibit-fdefine-warnings` to `:just-warn`. `defconstant` obeys that variable, just as `query-about-redefinition` does. Use `(setq sys:inhibit-fdefine-warnings nil)` to revert to the querying mode.

When the value of a constant is changed by a patch file, a warning is printed. `defconstant` assumes that changing the value is dangerous because the old value might have been incorporated into compiled code, which is out of date if the value changed.

In general, you should use `defconstant` to declare constants whose value is a number, character, or interned symbol and is guaranteed not to change. An example is π. The compiler can optimize expressions that contain references to these constants. If the value is another type of Lisp object or if it might change, you should use `zl:defconst` instead.

documentation, if provided, should be a string. It is accessible to the documentation function.

For example:

```
(defvar *max-alarms* 1000
   "The maximum number of times alarms can be sounded."
)
```

For more information see the section "Special Forms for Defining Special Variables".

**deff** function definition

This is a simplified version of `def`. It evaluates the form `definition`, which should produce a function, and makes that function the definition of `function`, which is not evaluated. `deff` is used for giving a function spec a definition that is not obtainable with the specific defining forms such as `defun` and `macro`. For example:

```
(deff foo 'bar)
```

makes `foo` equivalent to `bar`, with an indirection so that if `bar` changes, `foo` likewise changes;

```
(deff foo (function bar))
```
copies the definition of \texttt{bar} into \texttt{foo} with no indirection, so that further changes to \texttt{bar} have no effect on \texttt{foo}.

\begin{verbatim}
defflavor name instance-variables component-flavors &rest options \end{verbatim}

\texttt{name} is a symbol that is the name of this flavor.

\texttt{defflavor} defines the name of the flavor as a type name in both the Common Lisp and Zetalisp type systems; for further information, see the section "Flavor Instances and Types". \texttt{defflavor} also defines the name of the flavor as a presentation type name; for further information, see the section "User-defined Data Types as Presentation Types".

\texttt{instance-variables} is a list of the names of the instance variables containing the local state of this flavor. Each element of this list can be written in two ways: either the name of the instance variable by itself, or a list containing the name of the instance variable and a default initial value for it. Any default initial values given here are forms that are evaluated by \texttt{make-instance} if they are not overridden by explicit arguments to \texttt{make-instance}.

If you do not supply an initial value for an instance variable as an argument to \texttt{make-instance}, and there is no default initial value provided in the \texttt{defflavor} form, the value of an instance variable remains unbound. (Another way to provide a default is by using the \texttt{:default-init-plist} option to \texttt{defflavor}.)

\texttt{component-flavors} is a list of names of the component flavors from which this flavor is built.

Each \texttt{option} can be either a keyword symbol or a list of a keyword symbol and its arguments. The syntax of the \texttt{defflavor} \texttt{options} is given below, and the semantics of the options are described in detail elsewhere: See the section "Summary of \texttt{defflavor} Options". See the section "Complete Options for \texttt{defflavor}".

Several \texttt{options} affect instance variables, including:

\begin{verbatim}
:initable-instance-variables
:gettable-instance-variables
:locatable-instance-variables (not available in CLOE)
:rightable-instance-variables
:settable-instance-variables
:special-instance-variables (not available in CLOE)
:writable-instance-variables
\end{verbatim}

The options listed above can be given in two ways:

\begin{verbatim}
keyword

(keyword var1 var2 ...)\end{verbatim}

A list containing the keyword and one or more instance variables indicates that this option refers only to the instance variables listed here.
Briefly, the syntax of the other options is as follows:

:abstract-flavor
  (:area-keyword symbol) (not available in CLOE)
  (:component-order args...)  
  (:conc-name symbol)
  (:constructor args...)  
  (:default-handler function-name)
  (:default-init-plist plist)
  (:documentation string)
  (:functions internal-function-names)
  (:init-keywords symbols...)
  (:method-combination symbol)
  (:method-order generic-function-names)
  (:mixture specs...)
  :no-vanilla-flavor (not available in CLOE)
  (:ordered-instance-variables symbols)
  (:required-flavors flavor-names)
  (:required-init-keywords init-keywords)
  (:required-instance-variables symbols)
  (:required-methods generic-function-names)
  (:special-instance-variables-binding-methods generic-function-names)
    (not available in CLOE)

The following form defines a flavor `wink` to represent tiddly-winks. The instance variables `x` and `y` store the location of the wink. The default initial value of both `x` and `y` is 0. The instance variable `color` has no default initial value. The options specify that all instance variables are :initable-instance-variables; `x` and `y` are :writable-instance-variables; and `color` is a :readable-instance-variable.

```lisp
(defflavor wink ((x 0) (y 0) color) ;x and y represent location
  () ;no component flavors
  (:initable-instance-variables)
  (:writable-instance-variables x y) ;this implies readable
  (:readable-instance-variables color))
```

You can specify that an option should alter the behavior of instance variables inherited from a component flavor. To do so, include those instance variables explicitly in the list of instance variables at the top of the defflavor form. In the following example, the variables `x` and `y` are explicitly included in this defflavor form, even though they are inherited from the component flavor, `wink`. These variables are made initable in the defflavor form for `big-wink`; they are made writable in the defflavor form for `wink`.

```lisp
(defflavor big-wink (x y size)
  (wink) ;wink is a component
  (:initable-instance-variables x y))
```

If you specify a defflavor option for an instance variable that is not included in this defflavor form, an error is signalled. Flavors assumes you misspelled the name of the instance variable.
For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

format:defformat directive (arg-type) arglist body ...

Function

Defines a new format directive.

directive is a symbol that names the directive. If directive is longer than one character, the user must enclose it in backslashes in calls to format. For example:

(format t "-\foo\" ...)  
directive is usually in the format package; if it is in another package, the user must specify the package in calls to format. For example, we've defined a format directive called si:keystroke that prints out the short names for all characters.

(defun gtest ()  
  (loop for (name char) in '(("Space" #\space)  
     ("c-Space" #\c-space)  
     ("Tab" #\tab)  
     ("Page" #\page)  
     ("Left" #\mouse-L)  
     ("c-Left" #\c-mouse-L)  
     ("A" #\A)  
     ("c-A" #\c-A))  
     do  
     (format t "--%A: %-\si:keystroke\" name char char))) =>

Space:  , Space
  c-Space: c- , c-Space
  Tab:   , Tab
  Page:  , Page
  Left: Mouse-L, Mouse-L
  c-Left: c-Mouse-L, c-Mouse-L
  A:     A
  c-A:   c-A, c-A
NIL

format:defformat defines a function to be called when format is called using directive. body is the body of the function definition. arg-type is a keyword that determines the arguments to be passed to the function as arglist:

:no-arg The directive uses no arguments. The function is passed one argument, a list of parameters to the directive. The value returned by the function is ignored.

:one-arg The directive uses one argument. The function is passed two arguments: the argument associated with the directive and a list of parameters to the directive. The value returned by the function is ignored.
:multi-arg

The directive uses a variable number of arguments. The function is passed two arguments. The first is a list of the first argument associated with the directive and all the remaining arguments to format. The second is a list of parameters to the directive. The function should cdr down the list of arguments, using as many as it wants, and return the tail of the list so that the remaining arguments can be given to other directives.

The function can examine the values of format:colon-flag and format:atsign-flag. If format:colon-flag is not nil, the directive was given a : modifier. If format:atsign-flag is not nil, the directive was given an @ modifier.

The function should send its output to the stream that is the value of format:*format-output*.

Here is an example of a format directive that takes one argument and prints a number in base 7:

```lisp
(format:defformat format:base-7 (:one-arg) (argument parameters)
parameters ; ignored
(let ((*print-base* 7))
  (princ argument format:*format-output*))
)
```

Now:

```lisp
(format nil ~> ~\base-7\ <" 8) => ~> 11 <"
```

deffunction fspec lambda-type lambda-list &body rest

Defines a function using an arbitrary lambda macro in place of lambda. A deffunction form is like a defun form, except that the function spec is immediately followed by the name of the lambda macro to be used. deffunction expands the lambda macro immediately, so the lambda macro must already be defined before deffunction is used. For example, suppose the ilisp lambda macro were defined as follows:

```lisp
(lambda-macro ilisp (x)
  `(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
)
```

Then the following example would define a function called new-list that would use the lambda macro called ilisp:

```lisp
(deffunction new-list ilisp (x y z)
  (list x y z))
```

new-list’s arguments are optional, and any extra arguments are ignored. Examples:

```lisp
(new-list 1 2) => (1 2 nil)
(new-list 1 2 3 4) => (1 2 3)
```

defgeneric name arglist &body options

Defines a generic function named name with the signature of the function spec and the options specified. The body of the function is the options. The body may be a list of lambda expressions, each followed by the body of the function. The body may also be a single lambda expression, which is repeated for each argument of the function. The body may be a form that returns a function, which is then defined as the generic function.

One special form is used to define a generic function: deffunction. A deffunction form is like a defun form, except that the function spec is immediately followed by the name of the lambda macro to be used. deffunction expands the lambda macro immediately, so the lambda macro must already be defined before deffunction is used. For example, suppose the ilisp lambda macro were defined as follows:

```lisp
(lambda-macro ilisp (x)
  `(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
)
```

Then the following example would define a function called new-list that would use the lambda macro called ilisp:

```lisp
(deffunction new-list ilisp (x y z)
  (list x y z))
```

new-list’s arguments are optional, and any extra arguments are ignored. Examples:

```lisp
(new-list 1 2) => (1 2 nil)
(new-list 1 2 3 4) => (1 2 3)
```
Defines a generic function named \textit{name} that accepts arguments defined by \textit{arglist}, a lambda-list. \textit{arglist} is required unless the :\textit{function} option is used to indicate otherwise. \textit{arglist} represents the object that is supplied as the first argument to the generic function. The flavor of the first element of \textit{arglist} determines which method is appropriate to perform this generic function on the object.

The semantics of the options for \texttt{defgeneric} are described elsewhere: See the section "Options for \texttt{defgeneric}". The syntax of the options is summarized here:

\begin{itemize}
\item [:\texttt{compatible-message} \texttt{symbol}]
\item [\texttt{declare} \texttt{declaration}]
\item [:\texttt{dispatch} \texttt{flavor-name}]
\item [:\texttt{documentation} \texttt{string}]
\item [:\texttt{function} \texttt{body}...]
\item [:\texttt{inline-methods}]
\item [:\texttt{inline-methods :recursive}]
\item [:\texttt{method} \texttt{(flavor options...)} \texttt{body}...]
\item [:\texttt{method-arglist} \texttt{args}...]
\item [:\texttt{method-combination} \texttt{name} \texttt{args}...]
\item [:\texttt{optimize} \texttt{speed}]
\end{itemize}

For example, to define a generic function \texttt{total-fuel-supply} that works on instances of \texttt{army} and \texttt{navy}, and takes one argument (\texttt{fuel-type}) in addition to the object itself, we might supply military-group as \texttt{arg1}:

\begin{verbatim}
(defgeneric total-fuel-supply (military-group fuel-type)
  "Returns today's total supply of the given type of fuel available to the given military group."
  (:method-combination :sum))
\end{verbatim}

The generic function is called as follows:

\begin{verbatim}
(total-fuel-supply blue-army ':gas)
\end{verbatim}

The argument \texttt{blue-army} is known to be of flavor \texttt{army}. Therefore, Flavors chooses the method that implements the \texttt{total-fuel-supply} generic function on instances of the \texttt{army} flavor. That method takes only one argument, \texttt{fuel-type}:

\begin{verbatim}
(defmethod (total-fuel-supply army) (fuel-type)
  body of method)
\end{verbatim}

The arguments to \texttt{defgeneric} are displayed when you give the Arglist (\texttt{\textasciicircum\textasciicircum\textasciicircum\textasciicircum\textasciicircum\textasciicircum\textasciicircum}) command or press \texttt{c-sh-A} while this generic function is current.

It is not necessary to use \texttt{defgeneric} to set up a generic function. For further discussion: See the section "Use of \texttt{defgeneric}".

The function spec of a generic function is described elsewhere: See the section "Function Specs for Flavor Functions".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".
clos:defgeneric function-specifier lambda-list &rest options

Defines a generic function and returns the generic function object. It is not always necessary to use clos:defgeneric, because using clos:defmethod will automatically create a generic function, if it does not already exist. However, clos:defgeneric is useful for defining the interface of the generic function, and for specifying options that pertain to the generic function as a whole, such as the method-combination type.

The arguments to clos:defgeneric are:

function-specifier
The name of the generic function, which is either a symbol or a list of the form (future-common-lisp:setf symbol). An error is signaled if the function-specifier indicates an ordinary Lisp function, a macro, or a special form. In other words, you cannot use clos:defgeneric to redefine an ordinary function, macro, or special form to be a generic function.

lambda-list
 Specifies the lambda-list of the generic function. This is an ordinary lambda-list with some exceptions. Default values for optional and keyword parameters may not be provided, and &aux parameters may not be specified.

options
One or more of the following options:

(:argument-precedence-order {parameter-name}+)
Specifies the precedence order of the required parameters, which is used when ordering methods from most specific to least specific. The default argument precedence order is left to right, such that the leftmost parameter is considered first, followed by the parameters to its right. The name of each required parameter must be given.

(declare {declaration}+)
Specifies one or more declarations that pertain to the generic function. CLOS recognizes the optimize declaration, which declares whether method selection should be optimized for speed or space. Symbolics CLOS recognizes the following declarations as well: arglist, values, sys:downward-funarg, and sys:function-parent.

(:documentation string)
Provides a documentation string describing the generic function. You can get the documentation string of a class as follows:

   (documentation class-name 'type)

(:method-combination symbol {arg}*):
Specifies the method-combination type to be used by the generic function, and any arguments to the method-
combination type. The args are not evaluated. The default method-combination type is clos:standard.

(:method {method-qualifier} specialized-lambda-list &body body)

Enables you to define one or more methods for this generic function in the clos:defgeneric form, rather than having separate clos:defmethod forms. Sometimes it is convenient to define default methods within the clos:defgeneric form. For information on the arguments to the :method option, see the macro clos:defmethod.

(:generic-function-class class-name)

Specifies the class of the generic function. The default is clos:standard-generic-function. In Symbolics CLOS, the effects are undefined if any other value is given to this option.

(:method-class class-name)

Specifies the class of the methods for this generic function. The default is clos:standard-method. In Symbolics CLOS, the effects are undefined if any other value is given to this option.

zl:@define &rest ignore

Macro

This macro turns into nil, doing nothing. It exists for the sake of the @ listing generation program, which uses it to declare names of special forms that define objects (such as functions) that @ should cross-reference.

si:define-character-style-families device character-set &rest plists

Function

The mechanism for defining new character styles, and for defining which font should be used for displaying characters from character-set on the specified device. plists contain the actual mapping between character styles and fonts.

It is necessary that a character style be defined in the world before you access a file that uses the character style. You should be careful not to put any characters from a style you define into a file that is shared by other users, such as sys.translations.

It is possible for plists to map from a character style into another character style; this usage is called logical character styles. It is expected that the logical style used has its own mapping, in this si:define-character-style-families form or another such form, that eventually is resolved into an actual font.

plists is a nested structure whose elements are of the form:
Each target-font is one of:

- A symbol such as fonts:ctfont, which represents a font for a black and white Symbolics console.
- A string such as "furrier7", which represents a font for an LGP2 or LGP3 printer.
- A list whose car is :font and whose cadr is an expression representing a font, such as (:font ("Furrier" "B" 9 1.17)). This is also a font for an LGP2/LGP3 printer.
- A list whose car is :style and whose cdr is a character style, such as (:style family face size). This is an example of using a logical character style (see ahead for more details).

Each size is either a symbol representing a size, such as :normal, or an asterisk * used as a wildcard to match any size. The wildcard syntax is supported for the :size element only. When you use a wildcard for size the target-font must be a character style. The size element of target-font can be :same to match whatever the size of the character style is, or :smaller or :larger.

If you define a new size, that size cannot participate in the merging of relative sizes against absolute sizes. The ordered hierarchy of sizes is predefined. See the section "Merging Character Styles".

The elements can be nested in a different order, if desired. For example:

```
(:family family
  (:size size
    (:face face target-font
      :face face target-font
      :face face target-font)
    :size size
    (:face face target-font
      :face face target-font)))
```

The first example simply maps the character style BOX.ROMAN.NORMAL into the font fonts:boxfont for the character set si:*standard-character-set* and the device si:*b&w-screen*. The face ROMAN and the size NORMAL are already valid faces and sizes, but BOX is a new family; this form makes BOX one of the valid families.

```
;;;; -*- Package:SYSTEM-INTERNALS; Mode:LISP; Base: 10 -*-

(define-character-style-families *b&w-screen* *standard-character-set*
  '((:family :box
    (:size :normal (:face :roman fonts:boxfont)))))
```
Once you have compiled this form, you can use the Zmacs command Change Style Region (invoked by \texttt{c-x c-j}) and enter \texttt{BOX.ROMAN.NORMAL}. This form does not make any other faces or sizes valid for the BOX family.

The following example uses the wildcard syntax for the :size, and associates the faces \texttt{:italic}, \texttt{:bold}, and \texttt{:bold-italic} all to the same character style of \texttt{BOX.ROMAN.NORMAL}. This is an example of using logical character styles. This form has the effect of making several more character styles valid; however, all styles that use the BOX family are associated with the same logical character style, which uses the same font.

\begin{verbatim}
;;; -*- Package:SYSTEM-INTERNALS; Mode:LISP; Base: 10 -*-

(define-character-style-families *b&w-screen* *standard-character-set*
  '(:family :box
    (:size * (:face :italic (:style :box :roman :normal)
              :bold (:style :box :roman :normal)
              :bold-italic (:style :box :roman :normal))))
\end{verbatim}

For lengthier examples: See the section "Examples of \texttt{si:define-character-style-families}".

For related information: See the section "Mapping a Character Style to a Font".

\textbf{define-global-handler} \textit{name} \textit{conditions} \textit{arglist} \&\textit{body} \textbf{Function}

\textit{name} is a symbol, and a handler function by that name is defined.

\textit{conditions} is a condition name, or a list of condition names.

\textit{arglist} is a list of one element, the name of the argument (a symbol) which is bound to the condition object.

A global handler is like a bound handler with an important exception: unlike a bound handler which is of dynamic extent, a global handler is of \textit{indefinite} extent. Once defined, a global handler must therefore be specifically removed with \texttt{undefine-global-handler}.

Similarly, since a global handler could be called in any process by any program, it cannot use a \texttt{throw} the way a bound handler can. Instead it should return \texttt{nil} (keep searching for another handler), or return multiple values where the first one is the name of a proceed-type, as with bound handlers.

A note of caution: The global handler functions do not maintain the order of the global handler list in any way. If there are two handlers whose conditions overlap each other in such a way that some instantiable condition could be handled by either, then either handler might run, depending on the order in which they were defined. When there is more experience with use of global handlers we will try to develop a good approach to this problem.

Example:
(define-global-handler infinity-is-three sys:divide-by-zero
  (error)
  (values :return-values '(3)))

(/ 1 0) ==> 3

For a table of related items, see the section "Basic Forms for Global Handlers".

**define-loop-macro**  
*keyword*  
*Macro*

Can be used to make *keyword*, a *loop* keyword (such as *for*), into a Lisp macro that can introduce a *loop* form. For example, after evaluating:

```
(define-loop-macro for) => T
```

you can now write an iteration as:

```
(for i from 1 below n do ...)
```

```
(for i from 1 to 5
  do
    (print i)) =>

1
2
3
4
5 NIL
```

This facility exists primarily for diehard users of a predecessor of *loop*. Its unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

See the macro *loop*.

**define-loop-path**  
*Macro*

Allows a function to generate code for a path to be declared to *loop*:

```
(define-loop-path path-name-or-names path-function
  list-of-allowable-prepositions
  datum-1 datum-2 ...)
```

This defines *path-function* to be the handler for the path(s) *path-name-or-names*, which can be either a symbol or a list of symbols. Such a handler should follow the conventions described below. The *datum-i* are optional; they are passed in to *path-function* as a list.

*path-name*  
The name of the path that caused the path function to be invoked.
variable

The "iteration variable".

data-type

The data type supplied with the iteration variable, or nil if none was supplied.

prepositional-phrases

A list with entries of the form (preposition expression), in the order in which they were collected. This can also include some supplied implicitly (for example, an of phrase when the iteration is inclusive, and an in phrase for the default-loop-path path); the ordering shows the order of evaluation that should be followed for the expressions.

inclusive?

\(t\) if \(variable\) should have the starting point of the path as its value on the first iteration (by virtue of being specified with syntax like for var being expr and its path-name, nil otherwise. When \(t\), expr appears in prepositional-phrases with the of preposition; for example, for x being foo and its cdrs gets prepositional-phrases of (of foo)).

allowed-prepositions

The list of allowable prepositions declared for the path-name that caused the path function to be invoked. It and data can be used by the path function such that a single function can handle similar paths.

data

The list of "data" declared for the path-name that caused the path function to be invoked. It might, for instance, contain a canonicalized path-name, or a set of functions or flags to aid the path function in determining what to do. In this way, the same path function might be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings

A list of variables that need to be bound. The entries in it can be of the form variable, (variable expression), or (variable expression data-type). Note that it is the responsibility of the handler to make sure the iteration variable gets bound. All of these variables are bound in parallel; if initialization of one depends on others, it should be done with a setq in the prologue-forms. Returning only the variable without any initialization expression is not allowed if the variable is a destructuring pattern.

prologue-forms

A list of forms that should be included in the loop prologue.

the four items of the iteration specification

The four items: pre-step-endtest, steps, post-step-endtest, and pseudo-steps. See the section "The Iteration Framework".

another four items of iteration specification

If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to all iterations.
See the section "Iteration Paths for loop".

**define-loop-sequence-path** path-name-or-names fetchfun sizefun &optional sequence-type element-type

Macro

One very common form of iteration is that over the elements of some object that is accessible by means of an integer index. **loop** defines an iteration path function for doing this in a general way, and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

*path-name-or-names* is either an atomic path name or list of path names.

*fetchfun* is a function of two arguments: the sequence, and the index of the item to be fetched. (Indexing is assumed to be zero-origined.)

*sizefun* is a function of one argument, the sequence; it should return the number of elements in the sequence. *sequence-type* is the name of the data-type of the sequence, and *element-type* the name of the data-type of the elements of the sequence. These last two items are optional.

Examples:

```lisp
(define-loop-sequence-path ascii-char
  (lambda (string i)
    (ascii-code (aref string i)))
  length) => NIL

(loop for x being the ascii-char of "ABC"
  doing
    (print x)) =>
  65
  66
  67 NIL ; 65 is the ascii equivalent of "A"
```

The Symbolics Common Lisp implementation of **loop** utilizes the Symbolics Common Lisp array manipulation primitives to define both **array-element** and **array-elements** as iteration paths:

```lisp
(define-loop-sequence-path (array-element array-elements)
  aref array-active-length)
```

Then, the **loop** clause:

```lisp
  for var being the array-elements of array
  steps var over the elements of array, starting from 0. The sequence path function also accepts in as a synonym for of.
```

The range and stepping of the iteration can be specified with the use of all the same keywords that are accepted by the **loop** arithmetic stepper (**for var from ...**); they are **by**, **to**, **downto**, **from**, **downfrom**, **below**, and **above**, and are interpreted in the same manner. Thus:
(loop for var being the array-elements of array
    from 1 by 2
    ...)
steps var over all of the odd elements of array, and:
(loop for var being the array-elements of array
downto 0
    ...)
steps in "reverse" order.

All such sequence iteration paths allow you to specify the variable to be used as
the index variable, by use of the index keyword with the using prepositional
phrase. You can also use the sequence keyword with the using prepositional
phrase to specify the variable to be bound to the sequence.

See the section "Iteration Paths for loop".

define-method-combination name parameters method-patterns &body body Function
Provides a rich declarative syntax for defining new types of method combination.
This is more flexible and powerful than define-simple-method-combination.

name is a symbol that is the name of the new method combination type. parameters resembles the parameter list of a defmacro; it is matched against the parameters specified in the :method-combination option to defe generic or defflavor.

method-patterns is a list of method pattern specifications. Each method pattern selects some subset of the available methods and binds a variable to a list of the function specs for these methods. Two of the method patterns select only a single method and bind the variable to the chosen method's function spec if a method is found and otherwise to nil. The variables bound by method patterns are lexically available while executing the body forms. See the section "Method-Patterns Option to define-method-combination". Each option is a list whose car is a keyword. These can be inserted in front of the body forms to select special options. See the section "Options Available in define-method-combination". The body forms are evaluated to produce the body of a combined method. Thus the body forms of define-method-combination resemble the body forms of defmacro. Backquote is used in the same way. The body forms of define-method-combination usually produce a form that includes invocations of flavor:call-component-method and/or flavor:call-component-methods. These functions hide the implementation-dependent details of the calling of component methods by the combined method.

Flavors performs some optimizations on the combined method body. This makes it possible to write the body forms in a simple and easy-to-understand style, without being concerned about the efficiency of the generated code. For example, if a combined method chooses a single method and calls it and does nothing else, Flavors implements the called method as the handler rather than constructing a combined method. Flavors removes redundant invocations of progn and multiple-value-prog1 and performs similar optimizations.
The variables `flavor:generic` and `flavor:flavor` are lexically available to the body forms. The values of both variables are symbols:

- `flavor:generic` value is the name of the generic operation whose handler is being computed.
- `flavor:flavor` value is the name of the flavor.

The `body` forms are permitted to `setq` the variables defined by the `method-patterns`, if further filtering of the available methods is required, beyond the filtering provided by the built-in filters of the `method-patterns` mechanism. It is rarely necessary to resort to this. Flavors assumes that the values of the variables defined by the method patterns (after evaluating the body forms) reflect the actual methods that will be called by the combined method body.

`body` forms must not signal errors. Signalling an error (such as a complaint about one of the available methods) would interfere with the use of flavor examining tools, which call the user-supplied method combination routine to study the structure of the erroneous flavor. If it is absolutely necessary to signal an error, the variable `flavor:error-p` is lexically available to the body forms; its value must be obeyed. If `nil`, errors should be ignored.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

clos:define-method-combination name &rest rest

Macro

Defines a new method-combination type. There are two forms of `clos:define-method-combination`: a short form, for defining simple method-combination types; and a long form, for defining more complex method-combination types.

clos:define-method-combination returns the new method-combination object.

Short-form Syntax

clos:define-method-combination name short-form-option *

None of the subforms are evaluated. The arguments are:

- `name` The name of the method-combination type, a symbol. If the :operator option is not provided, the name of the method-combination type must also name a Lisp operator, such as a function, macro, or special form. The new method-combination type combines applicable primary methods in a call to this operator:

\[
\text{operator} \quad (\text{primary-method-1 args}) \\
(\text{primary-method-2 args}) \\
(...)
\]

- `short-form-option` These options are:
Provides a documentation string for the method-combination type.

Identity with one argument

If true, then an optimization is enabled for the case where there is only one applicable method, and it is a primary method. In that case, the operator is not called, and the value of the method is returned as the value of the generic function. This optimization makes sense for operators such as prog, +, and max, and others.

Operator

This option is used when you want the name of the method-combination type to be different than the name of the operator.

None of these options may be given more than once.

A simple method-combination type defined by the short form of clos:define-method-combination has the same semantics as the simple built-in method-combination types. For more information, see the section "CLOS Built-in Method-Combination Types".

Long-form Syntax

clos:define-method-combination name lambda-list

((method-group-specifier)*)

[[:arguments . lambda-list]]

[[:generic-function generic-function-symbol]]

documentation

doc-string

Each method-group-specifier is of the form:

(variable {{qualifier-pattern}+ | predicate} {option}*)

The options are:

:documentation string

:identity-with-one-argument boolean

:operator operator

None of these options may be given more than once.

Long-form Syntax

clos:define-method-combination name lambda-list

((method-group-specifier)*)

[[:arguments . lambda-list]]

[[:generic-function generic-function-symbol]]

documentation

doc-string

Each method-group-specifier is of the form:

(variable {{qualifier-pattern}+ | predicate} {option}*)

The options are:

:documentation string

:identity-with-one-argument boolean

:name is the name of the method-combination type, a symbol.

The lambda-list argument is an ordinary lambda-list. It receives any arguments provided after the name of the method-combination type in the :method-combination option to clos:defgeneric.

The next argument is a list of method-group-specifiers. Each method group specifier selects a subset of the applicable methods to play a particular role, either by
matching their qualifiers against some patterns or by testing their qualifiers with a predicate. These method group specifiers define all the method qualifiers that can be used with this type of method combination. If an applicable method does not fall into any method group, the system signals the error that the method is invalid for the kind of method combination in use.

Each method group specifier names a variable. During the execution of the forms in the body of `clos:define-method-combination`, this variable is bound to a list of the methods in the method group. The order of the methods in this list is most-specific-first, unless this is changed by :order.

A qualifier pattern is a list or the symbol * . A method matches a qualifier pattern if the method’s list of qualifiers is equal to the qualifier pattern (except that the symbol * in a qualifier pattern matches anything). Thus a qualifier pattern can be one of the following:

- The empty list (), which matches unqualified methods.

- The symbol *, which matches all methods.

- A true list, which matches methods with the same number of qualifiers as the length of the list when each qualifier matches the corresponding list element.

- A dotted list that ends in the symbol *. The * matches any number of additional qualifiers.

Each applicable method is tested against the qualifier patterns and predicates in left-to-right order. As soon as a qualifier pattern matches or a predicate returns true, the method becomes a member of the corresponding method group and no further tests are made. Thus if a method could be a member of more than one method group, it joins only the first such group. If a method group has more than one qualifier pattern, a method need only satisfy one of the qualifier patterns to be a member of the group.

The name of a predicate function can appear instead of qualifier patterns in a method group specifier. The predicate is called for each method that has not been assigned to an earlier method group; it is called with one argument, the method’s qualifier list. The predicate should return true if the method is to be a member of the method group. A predicate can be distinguished from a qualifier pattern because it is a symbol other than nil or *.

If there is an applicable method whose qualifiers are not valid for the method-combination type (that is, the qualifiers do not match any qualifier patterns, nor do they satisfy any predicate, nor do they fit any method group), the function `clos:invalid-method-error` is called.

Method group specifiers can have keyword options following the qualifier patterns or predicate. Keyword options can be distinguished from additional qualifier patterns because they are neither lists nor the symbol *. Note that none of these options may appear more than once in a method group specifier. The keyword options are as follows:
:description format-string

Provides a description of the role of methods in the method
group. Programming environment tools use

```
(apply #'format stream format-string (method-qualifiers method))
```

to print this description, which is expected to be concise. This
keyword option allows the description of a method qualifier to
be defined in the same module that defines the meaning of the
method qualifier. In most cases, format-string will not contain
any format directives, but they are available for generality. If :
description is not specified, a default description is generated
based on the variable name and the qualifier patterns and on
whether this method group includes the unqualified methods.
The argument format-string is not evaluated.

:order order

Specifies the order of methods. The order argument is a form
that evaluates to :most-specific-first or :most-specific-last. If
it evaluates to any other value, an error is signaled. This key-
word option is a convenience and does not add any expressive
power. If :order is not specified, it defaults to :most-specific-
first.

:required boolean

Specifies whether at least one method in this method group is
required. If the boolean argument is non-nil and the method
group is empty (that is, no applicable methods match the quali-
fier patterns or satisfy the predicate), an error is signaled. This key-
word option is a convenience and does not add any expressive
power. If :required is not specified, it defaults to nil.
The boolean argument is not evaluated.

The use of method group specifiers provides a convenient syntax to select methods,
to divide them among the possible roles, and to perform the necessary error
checking. It is possible to perform further filtering of methods in the body forms
by using normal list-processing operations and the functions clos:method-
qualifiers and clos:invalid-method-error. It is permissible to use setq on the vari-
bles named in the method group specifiers and to bind additional variables. It is
also possible to bypass the method group specifier mechanism and do everything in
the body forms. This is accomplished by writing a single method group with * as
its only qualifier pattern; the variable is then bound to a list of all of the applica-
ble methods, in most-specific-first order.

The body forms compute and return the Lisp form that specifies how the methods
are combined, that is, the effective method. The effective method uses the macro
clos:call-method. This macro has lexical scope and is available only in an effective
method form. Given a method object in one of the lists produced by the method
group specifiers and a list of next methods, the macro clos:call-method will invoke
the method such that clos:call-next-method has available the next methods.

When clos:call-method is called and the next-method-list argument is unsupplied,
it means that semantically there is no such thing as a "next method"; for example,
this is true for before-methods and after-methods in clos:standard method combination. Thus, when the next-method-list is unsupplied, clos:call-next-method is not allowed inside the method, and the behavior of clos:next-method-p is undefined. If the next-method-list argument is supplied as nil, and the method uses clos:call-next-method, then clos:no-next-method is called.

When an effective method has no effect other than to call a single method, CLOS can employ an optimization that uses the single method directly as the effective method, thus avoiding the need to create a new effective method. This optimization is active when the effective method form consists entirely of an invocation of the clos:call-method macro whose first subform is a method object and whose second subform is nil. Each clos:define-method-combination body is responsible for stripping off redundant invocations of prog1, and, multiple-value-prog1, and the like, if this optimization is desired.

The list (:arguments lambda-list) can appear before any declarations or documentation string. This form is useful when the method-combination type performs some specific behavior as part of the combined method and that behavior needs access to the arguments to the generic function. Each parameter variable defined by lambda-list is bound to a form that can be inserted into the effective method. When this form is evaluated during execution of the effective method, its value is the corresponding argument to the generic function.

The arguments to the generic function might not match the lambda-list. If there are too few arguments, nil is assumed for missing arguments. If there are too many arguments, the extra arguments are ignored. If there are unhandled keyword arguments, they are ignored. Supplied-p parameters work in the normal fashion. Default value forms are evaluated in the null lexical environment (except for bindings of :arguments parameters to their left).

If the effective method form returned by the body forms includes (setq variable ...), or (setf variable ...), or (future-common-lisp:setf variable ...), and variable is one of the :arguments parameters, the consequences are undefined.

Erroneous conditions detected by the body should be reported with clos:method-combination-error or clos:invalid-method-error; these functions add any necessary contextual information to the error message and will signal the appropriate error.

The body forms are evaluated inside of the bindings created by the lambda-list and method group specifiers. Declarations at the head of the body are positioned directly inside of bindings created by the lambda-list and outside of the bindings of the method group variables. Thus method group variables cannot be declared.

If the list (:generic-function generic-function-symbol) is provided, then within the body forms, generic-function-symbol is bound to the generic function object.

If a doc-string argument is present, it provides the documentation for the method-combination type.

The functions clos:method-combination-error and clos:invalid-method-error can be called from the body forms or from functions called by the body forms.
Examples

;;; Examples of the short form of define-method-combination

(define-method-combination and :identity-with-one-argument t)

(defun func and ((x class1) y) ...)

;;; The equivalent of this example in the long form is:

(define-method-combination and
  (&optional (order ':most-specific-first))
  ((around (:around))
   (primary (and) :order order :required t))
  (let ((form (if (rest primary)
         `(and ,0(mapcar #'(lambda (method)
                             `(call-method ,method ()
                              primary)))
      `(call-method ,(first primary) ()))))
    (if around
      `(call-method ,(first around)
          ,(0(rest around)
           (make-method ,form)))
      form)))

;;; Examples of the long form of define-method-combination
The default method-combination technique
(define-method-combination standard ()
  ((around (:around))
   (before (:before))
   (primary () :required t)
   (after (:after)))
(flet ((call-methods (methods)
         (mapcar #'(lambda (method)
                     (call-method ,method)))
         methods))
  (let ((form (if (or before after (rest primary))
               `(multiple-value-prog1
                  (progn ,@(call-methods before)
                     (call-method ,(first primary)
                     ,(rest primary)))
                ,@(call-methods (reverse after)))
             `(call-method ,(first primary))))
    (if around
      `(call-method ,(first around)
                    ,(rest around)
                    (make-method ,form))
    form)))

A simple way to try several methods until one returns non-nil
(define-method-combination or ()
  ((methods (or)))
  `(or ,@(mapcar #'(lambda (method)
                      `(call-method ,method))
                methods)))
(define-method-combination or
  (&optional (order ':most-specific-first))
  ((around (:around))
   (primary (or)))
  ;; Process the order argument
  (case order
    (:most-specific-first)
    (:most-specific-last (setq primary (reverse primary)))
    (otherwise (method-combination-error "~S is an invalid order.~@
      :most-specific-first and :most-specific-last are the possible values." order)))
  ;; Must have a primary method
  (unless primary
    (method-combination-error "A primary method is required."))
  ;; Construct the form that calls the primary methods
  (let ((form (if (rest primary)
                   '(or ,@(mapcar #'(lambda (method)
                                     '(call-method ,method))
                                 primary))
                 '(call-method ,(first primary))))
    ;; Wrap the around methods around that form
    (if around
      '(call-method ,(first around)
        ,(rest around)
        (make-method ,form))
      form)))

;; The same thing, using the :order and :required keyword options
(define-method-combination or
  (&optional (order ':most-specific-first))
  ((around (:around))
   (primary (or) :order order :required t))
  (let ((form (if (rest primary)
                 '(or ,@(mapcar #'(lambda (method)
                                   '(call-method ,method))
                               primary))
                 '(call-method ,(first primary))))
    (if around
      '(call-method ,(first around)
        ,(rest around)
        (make-method ,form))
      form)))
This short-form call is behaviorally identical to the preceding

```
(define-method-combination or :identity-with-one-argument t)
```

Order methods by positive integer qualifiers
:around methods are disallowed to keep the example small

```
(define-method-combination example-method-combination ()
  ((methods positive-integer-qualifier-p))
  '(progn ,@(mapcar #'(lambda (method)
                        '(call-method ,method))
            (stable-sort methods #'<
                          :key #'(lambda (method)
                                (first (method-qualifiers method)))))))
```

```
(defun positive-integer-qualifier-p (method-qualifiers)
  (and (= (length method-qualifiers) 1)
       (typep (first method-qualifiers) '(integer 0 *)))))
```

;; Example of the use of :arguments

```
(define-method-combination progn-with-lock ()
  ((methods ()))
  (:arguments object)
  '(unwind-protect
      (progn (lock (object-lock ,object))
             ,@(mapcar #'(lambda (method)
                           '(call-method ,method))
              methods))
       (unlock (object-lock ,object))))
```

define-modify-macro name args function &rest documentation-and-declarations

**Macro**

Defines a read-modify-write macro named `name`. An example of such a macro is `incf`. The first subform of the macro will be a generalized-variable reference. The `function` is literally the function to apply to the old contents of the generalized-variable to get the new contents; it is not evaluated. `args` describes the remaining arguments for the `name`; these arguments come from the remaining subforms of the macro after the generalized-variable reference. `args` may contain `&optional` and `&rest` markers. (The `&key` marker is not permitted here; `&rest` suffices for the purposes of `define-modify-macro`.) `documentation-and-declarations` is documentation for the macro `name` being defined.

The expansion of a `define-modify-macro` is equivalent to the following, except that it generates code that follows the semantic rules outlined above.
(defmacro name (reference . lambda-list)
  documentation-and-declarations
  '(setf ,reference
    (function ,reference ,arg1 ,arg2 ...)))
where arg1, arg2, ..., are the parameters appearing in args; appropriate provision is
made for a &rest parameter.

As an example, incf could have been defined by:

(define-modify-macro incf (&optional (delta 1)) +)

A similar read-modify-write macro for the logior operation of taking the logical
and of a number can be created by

(define-modify-macro logiorf (arg2) logior)

(setq first 5 second 6)

(logiorf first second) => 7

first => 7

In the previous example, the lambda list only refers to the second argument to
logior because these macros are presumed to take at least one argument, and only
additional arguments require specification. The unspecified first argument is up-
dated by the macro.

define-setf-method access-function subforms &body body

In this context, the word "method" has nothing to do with flavors.

This macro defines how to setf a generalized-variable reference that is of the form
(access-function . . .). The value of the generalized-variable reference can always be
obtained by evaluating it, so access-function should be the name of a function or a
macro.

subforms is a lambda list that describes the subforms of the generalized-variable
reference, as with defmacro. The result of evaluating body must be five values
representing the setf method. (The five values are described in detail at the end of
this discussion.) Note that define-setf-method differs from the complex form of
defsetf in that while the body is being executed the variables in subforms are
bound to parts of the generalized-variable reference, not to temporary variables
that will be bound to the values of such parts. In addition, define-setf-method
does not have the defsetf restriction that access-function must be a function or a
function-like macro. An arbitrary defmacro destructuring pattern is permitted in
subforms.

By definition, there are no good small examples of define-setf-method because the
easy cases can all be handled by defsetf. A typical use is to define the setf method
for ldb.
(define-setf-method ldb (bytespec int)
  (multiple-value-bind (temps vals stores
    store-form access-form)
    (get-setf-method int) ; Get SETF method for int.
    (let ((btemp (gensym)) ; Temp var for byte specifier.
      (store (gensym)) ; Temp var for byte to store.
      (stemp (first stores))) ; Temp var for int to store.
      ;; Return the SETF method for LDB as five values.
      (values (cons btemp temps) ; Temporary variables.
        (cons bytespec vals) ; Value forms.
        (list store) ; Store variables.
        `(let ((,stemp (dpb ,store ,btemp ,access-form)))
          ,store-form
          ,store) ; Storing form.
        `(ldb ,btemp ,access-form); Accessing form.
        ))))

Here are the five values that express a setf method for a given access form.

- A list of temporary variables.
- A list of value forms (subforms of the given form) to whose values the temporary variables are to be bound.
- A second list of temporary variable, called store variables.
- A storing form.
- An accessing form.

The temporary variables are bound to the value forms as if by let*; that is, the value forms are evaluated in the order given and may refer to the values of earlier value forms by using the corresponding variable.

The store variables are to be bound to the values of the newvalue form, that is, the values to be stored into the generalized variable. In almost all cases, only a single value is stored, and there is only one store variable.

The storing form and the accessing form may contain references to the temporary variables (and also, in the case of the storing form, to the store variables). The accessing form returns the value of the generalized variable. The storing form modifies the value of the generalized variable and guarantees to return the values of the store variables as its values. These are the correct values for setf to return. (Again, in most cases there is a single store variable and thus a single value to be returned.) The value returned by the accessing form is, of course, affected by execution of the storing form, but either of these forms may be evaluated any number
of times, and therefore should be free of side effects (other than the storing action of the storing form).

The temporary variables and the store variables are generated names, as if by `gensym` or `gentemp`, so that there is never any problem of name clashes among them, or between them and other variables in the program. This is necessary to make the special forms that do more than one `setf` in parallel work properly. These are `psetf`, `shiftf` and `rotatef`.

Here are some examples of `setf` methods for particular forms:

- For a variable `x`:
  
  ()
  ()
  (g0001)
  (setq x g0001)
  x

- For `(car exp)`:
  
  (g0002)
  (exp)
  (g0003)
  (progn (rplaca g0002 g0003) g0003)
  (car g0002)

- For `(subseq seq s e)`:
  
  (g0004 g0005 g0006)
  (seq s e)
  (g0007)
  (progn (replace g0004 g0007 :start1 g0005 :end1 g0006) g0007)
  (subseq g0004 g0005 g0006)

**define-simple-method-combination**  
`name operator` &optional `single-arg-is-value` pretty-name  
Special Form

Defines a new type of method combination that simply calls all the methods, passing the values they return to the function named `operator`.

It is also legal for `operator` to be the name of a special form. In this case, each subform is a call to a method. It is legal to use a lambda expression as `operator`.

`name` is the name of the method-combination type to be defined. It takes one optional parameter, the order of methods. The order can be either `:most-specific-first` (the default) or `:most-specific-last`.

When you use a new type of method combination defined by `define-simple-method-combination`, you can give the argument `:most-specific-first` or `:most-`
specific-last to override the order that this type of method combination uses by default.

If single-arg-is-value is specified and not nil, and if there is exactly one method, it is called directly and operator is not called. For example, single-arg-is-value makes sense when operator is +.

pretty-name is a string that describes how to print method names concisely. It defaults to (string-downcase name).

Most of the simple types of built-in method combination are defined with define-simple-method-combination. For example:

```
(define-simple-method-combination :and and t)
(define-simple-method-combination :or or t)
(define-simple-method-combination :list list)
(define-simple-method-combination :progn progn t)
(define-simple-method-combination :append append t)
```

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

define-symbol-macro name form

**Special Form**

Defines a symbol macro. name is a symbol to be defined as a symbol macro. form is a Lisp form to be substituted for the symbol when the symbol is evaluated. A symbol macro is more like an inline function than a macro: form is the form to be substituted for the symbol, not a form whose evaluation results in the substitute form.

Example:

```
(define-symbol-macro foo (+ 3 bar))
(setq bar 2)
foo => 5
```

A symbol defined as a symbol macro cannot be used in the context of a variable. You cannot use `setq` on it, and you cannot bind it. You can use `setf` on it: `setf` substitutes the replacement form, which should access something, and expands into the appropriate update function.

For example, suppose you want to define some new instance variables and methods for a flavor. Then, you want to test the methods using existing instances of the flavor. For testing purposes, you might use hash tables to simulate the instance variables, using one hash table per instance variable with the instance as the key. You could then implement an instance variable `x` as a symbol macro:

```
(defvar x-hash-table (make-hash-table))
(define-symbol-macro x (gethash self x-hash-table))
```

To simulate setting a new value for `x`, you could use `(setf x value)`, which would expand into `(setf (gethash self x-hash-table) value)`.
**deflambda-macro** name pattern &body body

*Function*

Like **defmacro**, but defines a lambda macro instead of a normal macro.

`name` is the name of the lambda macro to be defined; it can be any function spec. See the section "Function Specs". The `pattern` can be anything made up out of symbols and conses. It is matched against the body of the lambda macro form; both `pattern` and the form are `car`ed and `cdr`ed identically, and whenever a non-nil symbol occurs in `pattern`, the symbol is bound to the corresponding part of the form. If the corresponding part of the form is `nil`, it goes off the end of the form. &optional, &rest, &key, and &body can be used to indicate where optional pattern elements are allowed.

All of the symbols in `pattern` can be used as variables within `body`. `body` is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro.

Here is an example of **deflambda-macro** used to define a lambda macro:

```lisp
(defun deflambda-macro ilisp (arglist &rest body)
  '(lambda (&optional ,@arglist) ,@body))
```

This defines a lambda macro called `ilisp`. After it has been defined, the following list is a valid Lisp function:

```lisp
(ilisp (x y z) (list x y z))
```

**zl:deflambda-macro-displace** name pattern &body body

*Special Form*

Like **zl:defmacro-displace**, but defines a displacing lambda macro instead of a displacing normal macro.

**deflocf** access-function locate-function-or-subforms &body body

*Function*

Defines how **locf** creates a locative pointer to a cell referred to by `access-function`, similar to the way **defsetf** defines how **setf** sets a generalized-variable. See the macro **defsetf**.

Subforms of the `access-function` are evaluated exactly once and in the proper left-to-right order. A **locf** of a call on `access-function` will also evaluate all of `access-function`’s arguments; it cannot treat any of them specially.

A **deflocf** function has two forms: a simple case and a slightly more complicated one. In the simplest case, `locate-function-or-subforms` is the name of a function or macro. In the more complicated case, `locate-function-or-subforms` is a lambda list of arguments.

The simple form of **deflocf** is

```lisp
(defun deflocf array-leader ap-leader)
```

This says that the form to create a locative pointer to `array-leader` is the function `ap-leader`.
If the access-function and the locate-function-or-subforms take their arguments in a different order or do anything special with their arguments, the more complicated form must be used, for example:

```
(deflocf fs:pathname-property-list (pathname)
  '(send ,pathname :property-list-location))
```

**defmacro** name pattern &body body

A general-purpose macro-defining macro. A **defmacro** form looks like:

```
(defmacro name pattern . body)
```

name is the name of the macro to be defined; it can be any function spec. See the section "Function Specs". Specifies the expansion of forms characterized by calling name with arguments as indicated in pattern. The expansion function is stored as the macro definition associated with name. The macro definition is evaluated in the context of the global environment. (To establish macros in the current lexical environment, **macrolet** may be used instead of **defmacro**). The pattern argument specifies an extension to Common Lisp syntax by characterizing a structured form whose car is name. The chief distinction between macro lambda-lists and those used in function definitions is that macro lambda-lists recursively specify list-forms (also lambda-lists) that represent list forms actually appearing in the call. Consider the macro do in the following example:

```
(do ((i 0 (+ i 1))
    (j 10 (- j 2)))
   ((<= j 0) j)
  (setf (aref *glob* i) j))
```

The outer parentheses in the variable initialization and step form

```
(i 0 (+ i 1))
```

are explicitly represented in the lambda-list of the do definition. The inner set surrounding the + form is simply an argument form for the step parameter. This is similar to a form argument paired to a defun parameter. However, in the latter case the form is evaluated to produce a value for the parameter, while in the macro case the form represents a textual replacement for the step parameter.

The pattern can be anything made up out of symbols and conses. It is matched against the body of the macro form; both pattern and the form are car'ed and cdr'ed identically, and whenever a non-nil symbol occurs in pattern, the symbol is bound to the corresponding part of the form. If the corresponding part of the form is nil, it goes off the end of the form. **&optional**, **&rest**, **&key**, and **&body** can be used to indicate where optional pattern elements are allowed.

Of the existing limitations on this extension to the lambda-list function called de-structuring, most notable is that a lambda-list-form may not be used where a list-form appears in a defun-style lambda-list. For example, following the **&optional** lambda-list keyword. All of the symbols in pattern can be used as variables within body.
body is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro. Macro lambda-lists may also contain three additional lambda-list keywords: &body, &environment, and &whole.

defmacro could have been defined in terms of destructuring-bind as follows, except that the following is a simplified example of defmacro showing no error-checking and omitting the &environment and &whole features.

```
(defun defmacro (name pattern &body body)
  `(macro ,name (form env)
     (destructuring-bind ,pattern (cdr form)
       ,@body)))
```

The pattern in a defmacro is like the lambda-list of a normal function. defmacro is allowed to contain certain &-keywords.

defmacro destructures all levels of patterns in a consistent way. The inside patterns can also contain &-keywords and there is checking of the matching of lengths of the pattern and the subform. See the special form destructuring-bind.

This behavior exists for all of defmacro’s parameters, except for &environment, &whole, and &aux.

You must use &optional in the parameter list if you want to call the macro with less than its full complement of subforms. There must be an exact one-to-one correspondence between the pattern and the data unless you use &optional in the parameter destructuring pattern.

```
(defun nand (&rest args) '(not (and ,&args)))
```

```
(defun with-output-to-string
  ((var &optional string &key index) &body body)
  `(let ((with-output-to-string-internal-string
           ,(or string '(make-array 100 :type 'art-string)))
        ...
        ,@body))
```

defmacro accepts these keywords:

&optional &optional is followed by variable, (variable), (variable default), or (variable default present-p), exactly the same as in a function. Note that default is still a form to be evaluated, even though variable is not being bound to the value of a form. variable does not have to be a symbol; it can be a pattern. In this case the first form is disallowed because it is syntactically ambiguous. The pattern must be enclosed in a singleton list.

&rest The same as using a dotted list as the pattern, except that it might be easier to read and leaves a place to put &aux.

&key Separates the positional parameters and rest parameter from the keyword parameters. See the section "Evaluating a Function Form".
&allow-other-keys In a lambda-list that accepts keyword arguments, says that keywords that are not specifically listed after &key are allowed. They and the corresponding values are ignored, as far as keyword arguments are concerned, but they do become part of the rest argument, if there is one.

&aux The same in a macro as in a function, and has nothing to do with pattern matching. It separates the destructuring pattern of a macro from the auxiliary variables. Following &aux you can put entries of the form:

```
(variable initial-value-form)
```

or just variable if you want it initialized to nil or do not care what the initial value is.

&body Identical to &rest except that it informs the editor and the grinder that the remaining subforms constitute a "body" rather than "arguments" and should be indented accordingly. The &body keyword should be used when the body is an implicit `progn` to signal printing routines to indent the body of macro calls as in an implicit `progn`.

&whole For macros defined by defmacro or macrolet only. &whole is followed by variable, which is bound to the entire macro-call form or subform variable is the value that the macro-expander function receives as its first argument. &whole is allowed only in the top-level pattern, not in inside patterns.

```
(defun abc (&whole form arg1 arg2)
  (if (and arg2 (not arg1))
      `(cde ,(cdr form) ,arg2)
      `(efg ,arg1 ,arg2)))
```

&environment For macros defined by defmacro or macrolet only. &environment is followed by variable, which is bound to an object representing the lexical environment where the macro call is to be interpreted. This environment might not be the complete lexical environment. It should be used only with the macroexpand function for any local macro definitions that the macrolet construct might have established within that lexical environment. &environment is allowed only in the top-level pattern, not in inside patterns. See the section "Lexical Environment Objects and Arguments". See the macro defmacro.

&list-of is not supported as a result of making defmacro Common-Lisp compatible. Use zl:loop or mapcar instead of &list-of.

See the special form destructuring-bind.

zl:defmacro-displace name pattern &body body
Like `defmacro`, except that it defines a displacing macro, using the `zl:displace` function.

**defmacro-in-flavor**

*(function-name flavor-name) arglist body...*)  

Function

Defines a macro inside a flavor. Functions inside the flavor can use this macro, but the macro is not accessible in the global environment.

See the section “Defining Functions Internal to Flavors”.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section “Summary of Flavor Functions and Variables”.

**defmethod**

Special Form

A method is the code that performs a generic function on an instance of a particular flavor. It is defined by a form such as:

```
(defmethod (generic-function flavor options...) (arg1 arg2...) body...)
```

The method defined by such a form performs the generic function named by `generic-function`, when that generic function is applied to an instance of the given `flavor`. (The name of the generic function should not be a keyword, unless you want to define a message to be used with the old `send` syntax.) You can include a documentation string and `declare` forms after the argument list and before the body.

A generic function is called as follows:

```
(generic-function g-f-arg1 g-f-arg2...)
```

Usually the flavor of `g-f-arg1` determines which method is called to perform the function. When the appropriate method is called, `self` is bound to the object itself (which was the first argument to the generic function). The arguments of the method are bound to any additional arguments given to the generic function. A method’s argument list has the same syntax as in `defun`.

The `body` of a `defmethod` form behaves like the body of a `defun`, except that the lexical environment enables you to access instance variables by their names, and the instance by `self`.

For example, we can define a method for the generic function `list-position` that works on the flavor `wink`. `list-position` prints the representation of the object and returns a list of its x and y position.

```
(defmethod (list-position wink) () ; no args other than object
  "Returns a list of x and y position."
  (print self) ; self is bound to the instance
  (list x y)) ; instance vars are accessible
```

The generic function `list-position` is now defined, with a method that implements it on instances of `wink`. We can use it as follows:
If no options are supplied, you are defining a primary method. Any options given are interpreted by the type of method combination declared with the :method-combination argument to either defgeneric or defflavor. See the section "Defining Special-Purpose Methods". For example, :before or :after can be supplied to indicate that this is a before-daemon or an after-daemon. For more information: See the section "Defining Before- and After-Daemons".

If the generic function has not already been defined by defgeneric, defmethod sets up a generic function with no special options. If you call defgeneric for the name generic-function later, the generic function is updated to include any new options specified in the defgeneric form.

Several other sections of the documentation contain information related to defmethod: See the section "defmethod Declarations". See the section "Writing Methods for make-instance". See the section "Function Specs for Flavor Functions". See the section "Setter and Locator Function Specs". See the section "Implicit Blocks for Methods". See the section "Variant Syntax of defmethod". See the section "Defining Methods to Be Called by Message-Passing".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

---

**clos:defmethod function-specifier (method-qualifier)* specialized-lambda-list &body body**

Macro

Defines a method for a generic function and returns the method object.

If the generic function has not been defined, then clos:defmethod defines the generic function with the default argument precedence order, method-combination type, method class, and generic function class. The lambda-list of the generic function is congruent with that of the method. If the method's lambda-list has keyword parameters, then the generic function's lambda-list will specify &key, but not name any keyword parameters.

If the generic function has a method with the same parameter specializers and qualifiers, then that method is redefined.

CLOS requires that the lambda-lists of a generic function and all its methods must be congruent. If a method violates the congruency pattern of its generic function, an error is signaled.

The arguments to clos:defmethod are:

- **function-specifier**  The name of the generic function, which is either a symbol or a list of the form (future-common-lisp:setf symbol). An error is signaled if the function-specifier indicates an ordinary Lisp function, a macro, or a special form. In other words, you cannot use clos:defmethod to redefine an ordinary function, macro, or special form to be a generic function.
The method’s qualifier or qualifiers state the role of this method in performing the work of the generic function. They are non-nil atoms that are used by the method-combination type. The clos:standard method-combination type supports the qualifiers :around, :before, and :after, as well as unqualified methods.

A specialized lambda-list is an extension of an ordinary lambda-list that can specialize any of its required parameters. The specialized lambda-list states the set of arguments for which this method will be applicable, as described below.

A specialized parameter is a list in one of the following formats:

(variable-name (eql form))
(variable-name class-name)

An unspecialized parameter appears as a variable name; this is the same as if the parameter were specialized on the class named t.

When a generic function is called with a set of arguments, CLOS determines which methods are applicable, based on the required arguments and the lambda-lists of the methods for the generic function. For a method to be applicable, each required argument must satisfy the corresponding parameter in the method’s lambda-list.

When a parameter is specialized with (eql form), the form is evaluated once, at the time that the clos:defmethod form is evaluated. The form is not evaluated each time the generic function is called.

If the value of form is object, then the argument satisfies the specialized parameter if the following form returns true:

(eql argument 'object)

When a parameter is specialized with a class name, the argument satisfies the specialized parameter if the following form returns true:

(typep argument 'class-name)

When a parameter is unspecialized (the variable-name appears as a lone symbol which is not enclosed within a list), any argument satisfies the parameter.

Note that if you are defining a future-common-lisp:setf method, then the order of parameters in the specialized lambda-list is as shown:

(new-value args...)
As in other methods, in **future-common-lisp:setf** methods, any of the required parameters may be specialized.

**declarations, documentation**

The **clos:defmethod** syntax allows for declarations and/or documentation strings to appear after the **specialized-lambda-list** and before the **body**.

**body**

The body contains forms that do the work of the generic function. When methods are defined to work together (via different roles), each method implements some portion of the work of the generic function. Often the body needs to access slots of instances that are given as arguments to the generic function. There are several ways to access slots: using reader or writer generic functions, using **clos:with-accessors**, or using **clos:with-slots**.

The **body** has an implicit **block** around it. If the generic function's name is a symbol, the block has the same name as the generic function. If the generic function's name is **(future-common-lisp:setf symbol)**, the block has the name **symbol**.

**Examples**

The following examples show the applicability of methods:

```lisp
;;; Applicable when first arg is a ship, second arg is a plane
(clos:defmethod collide ((s ship) (p plane) location)
  body)

;;; Applicable when first arg is a plane, second arg is a plane
(clos:defmethod collide :after ((p plane) (p plane) location)
  body)

;;; Applicable when second arg is a plane
(clos:defmethod collide (vehicle (p plane) location)
  body)

;;; Applicable when first arg is eql to the value of *Enterprise*
(clos:defmethod collide ((ent (eql *Enterprise*)) vehicle location)
  body)
```

The **:accessor** and **:writer** options to **clos:defclass** enable you to define **future-common-lisp:setf** methods for slots automatically, but you can also do it by using **clos:defmethod**, as shown in this example:

```lisp
(clos:defclass boat () (speed location))

(clos:defmethod (future-common-lisp:setf 'speed) (new-value (b boat))
  (setf (slot-value b) new-value))
```
defpackage name options...

Defines a package named name; the name must be a symbol so that the source file name of the package can be recorded and the editor can correctly sectionize the definition. If no package by that name already exists, a new package is created according to the specified options. If a package by that name already exists, its characteristics are altered according to the options specified. If any characteristic cannot be altered, an error is signalled. If the existing package was defined by a different file, you are queried before it is changed, as with any other type of definition.

Each option is a keyword or a list of a keyword and arguments. A keyword by itself is equivalent to a list of that keyword and one argument, t; this syntax really only makes sense for the :external-only and :hash-inherited-symbols keywords.

Wherever an argument is said to be a name or a package, it can be either a symbol or a string. Usually symbols are preferred, because the reader standardizes their alphabetic case and because readability is increased by not cluttering up the defpackage form with string quote (" characters.

None of the arguments are evaluated. The keywords arguments, most of which are identical to make-package’s, are:

(:nicknames name name...) for defpackage
:nicknames '(name name...) for make-package

The package is given these nicknames, in addition to its primary name.

(:prefix-name name) for defpackage
:prefix-name name for make-package

This name is used when printing a qualified name for a symbol in this package. You should make the specified name one of the nicknames of the package or its primary name. If you do not specify :prefix-name, it defaults to the shortest of the package’s names (the primary name plus the nicknames).

(:use package package...)

External symbols and relative name mappings of the specified packages are inherited. If this option is not specified, it defaults to (:use CL) (:use global) in Zetalisp). To inherit nothing, specify (:use).

(:shadow name name...) for defpackage
:shadow '(name name...) for make-package

Symbols with the specified names are created in this package and declared to be shadowing.

(:export name name...) for defpackage
:export '(name name...) for make-package

Symbols with the specified names are created in this package, or inherited from the packages it uses, and declared to be external.

(:import symbol symbol...) for defpackage
:import '(name name...) for make-package
   The specified symbols are imported into the package. Note that unlike
   :export, :import requires symbols, not names; it matters in which package
   this argument is read.

(:shadowing-import symbol symbol...) for defpackage
:shadowing-import '(symbol symbol...) for make-package
   The same as :import but no name conflicts are possible; the symbols are
   declared to be shadowing.

:import-from package name name... for defpackage
:import-from '(package name name...) for make-package
   The specified symbols are imported into the package. The symbols to be im-
   ported are obtained by looking up each name in package.

(defpackage only) This option exists primarily for system bootstrapping,
   since the same thing can normally be done by :import. The difference be-
   tween :import and :import-from can be visible if the file containing a
   defpackage is compiled; when :import is used the symbols are looked up at
   compile time, but when :import-from is used the symbols are looked up at
   load time. If the package structure has been changed between the time the
   file was compiled and the time it is loaded, there might be a difference.

(:relative-names (name package) (name package)... - defpackage
:relative-names '((name package) ...) - make-package
   Declares relative names by which this package can refer to other packages.
   The package being created cannot be one of the packages, since it has not
   been created yet. For example, to be able to refer to symbols in the
   common-lisp package print with the prefix lisp: instead of cl: when they
   need a package prefix (for instance, when they are shadowed), you would
   use :relative-names like this:

   (defpackage my-package (:use cl)
      (:shadow error)
      (:relative-names (lisp common-lisp)))

   (let ((*package* (find-package 'my-package)))
      (print (list 'my-package::error 'cl:error)))

(:relative-names-for-me (package name) ...) for defpackage
:relative-names-for-me '((package name) ...) for make-package
   Declares relative names by which other packages can refer to this package.

(defpackage only) It is valid to use the name of the package being created
   as a package here; this is useful when a package has a relative name for
   itself.

(:size number) for defpackage
:size number for make-package
   The number of symbols expected in the package. This controls the initial
   size of the package's hash table. You can make the :size specification an
   underestimate; the hash table is expanded as necessary.
(:hash-inherited-symbols boolean) for defpackage
:hash-inherited-symbols boolean for make-package
  If true, inherited symbols are entered into the package’s hash table to
  speed up symbol lookup. If false (the default), looking up a symbol in this
  package searches the hash table of each package it uses.

(:external-only boolean) for defpackage
:external-only boolean for make-package
  If true, all symbols in this package are external and the package is locked.
  This feature is only used to simulate the old package system that was used
  before Release 5.0. See the section "External-only Packages and Locking".

(:include package package...) for defpackage
:include '(package package...) for make-package
  Any package that uses this package also uses the specified packages. Note
  that if the :include list is changed, the change is not propagated to users
  of this package. This feature is used only to simulate the old package sys-
  tem that was used before Release 5.0.

(:new-symbol-function function) for defpackage
:new-symbol-function function for make-package
  function is called when a new symbol is to be made present in the package.
  The default is si:pkg-new-symbol unless :external-only is specified. Do not
  specify this option unless you understand the internal details of the package
  system.

(:colon-mode mode) for defpackage
:colon-mode mode for make-package
  If mode is :external, qualified names mentioning this package behave dif-
  ferently depending on whether ":" or "::*" is used, as in Common Lisp. ":" names access only external symbols. If mode is :internal, ":*" names access all symbols. :external is the default. See the section "Specifying Internal
  and External Symbols in Packages".

(:prefix-intern-function function) for defpackage
:prefix-intern-function function for make-package
  The function to call to convert a qualified name referencing this package
  with ":*" (rather than "::*") to a symbol. The default is intern unless (:colon-
  mode :external) is specified. Do not specify this option unless you under-
  stand the internal details of the package system.

defparameter variable initial-value &optional documentation Special Form
  The same as defvar, except that variable is always set to initial-value regardless of
  whether variable is already bound. The rationale for this is that defvar declares a
  global variable, whose value is initialized to something but is then changed by the
  functions that use it to maintain some state. On the other hand, defparameter de-
clares a constant, whose value is never changed by the normal operation of the program, only by changes to the program. **defparameter** always sets the variable to the specified value so that if, while developing or debugging the program, you change your mind about what the constant value should be, and you then evaluate the **defparameter** form again, the variable gets the new value. It is not the intent of **defparameter** to declare that the value of variable never changes; for example, **defparameter** is not a license to the compiler to build assumptions about the value of variable into programs being compiled. See **defconstant** for that.

For example:

```
(defparameter *alarms-limit* 10
  "The number of alarms allowed to sound before a special message is printed."
)
```

See the section "Special Forms for Defining Special Variables".

**defprop** sym value indicator

*Special Form*

Gives sym's property list an indicator-property corresponding to value. After this is done, (get sym indicator) returns value. See the section "Property Lists".

**defprop** is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions That Operate on Property Lists".

**defselect** fspec &body methods

*Special Form*

Defines a function that is a select-method. This function contains a table of subfunctions; when it is called, the first argument, a symbol on the keyword package called the message name, is looked up in the table to determine which subfunction to call. Each subfunction can take a different number of arguments, and have a different pattern of &optional and &rest arguments. **defselect** is useful for a variety of "dispatching" jobs. By analogy with the more general message passing facilities in flavors, the subfunctions are sometimes called methods and the first argument is sometimes called a message.

The special form looks like:

```
(defun select (function-spec default-handler no-which-operations)
  (message-name (args...)
    body...)
  (message-name (args...)
    body...)
  ...
)
```

function-spec is the name of the function to be defined. default-handler is optional; it must be a symbol and is a function that gets called if the select-method is called with an unknown message. If default-handler is unsupplied or nil, then an error occurs if an unknown message is sent. If no-which-operations is non-nil, the :which-operations method that would normally be supplied automatically is sup-
pressed. The :which-operations method takes no arguments and returns a list of all the message names in the defselect.

The :operation-handled-p and :send-if-handles methods are automatically supplied. See the message :operation-handled-p. See the message :send-if-handles.

If function-spec is a symbol, and default-handler and no-which-operations are not supplied, then the first subform of the defselect can be just function-spec by itself, not enclosed in a list.

The remaining subforms in a defselect define methods. message-name is the message name, or a list of several message names if several messages are to be handled by the same subfunction. args is a lambda-list; it should not include the first argument, which is the message name. body is the body of the function.

A method subform can instead look like:

(message-name . symbol)

In this case, symbol is the name of a function that is called when the message-name message is received. It is called with the same arguments as the select-method, including the message symbol itself.

**defsetf** access-function storing-function-or-args &optional store-variables &body body

*Macro*

Defines how to setf a generalized-variable reference of the form (access-function .). The value of a generalized-variable reference can always be obtained by evaluating it, so access-function should be the name of a function or macro that evaluates its arguments, behaving like a function.

The user of defsetf provides a description of how to store into the generalized-variable reference and return the value that was stored (because setf is defined to return this value). Subforms of the reference are evaluated exactly once and in the proper left-to-right order. A setf of a call on access-function will also evaluate all of access-function's arguments; it cannot treat any of them specially. This means that defsetf cannot be used to describe how to store into a generalized variable that is a byte, such as (ldb field reference). To handle situations that do not fit the restrictions of defsetf, use define-setf-method, which gives the user additional control at the cost of additional complexity.

A defsetf function can take two forms, simple and complex. In the simple case, storing-function-or-args is the name of a function or macro. In the complex case, storing-function-or-args is a lambda list of arguments.

The simple form of defsetf is

(defsetf access-function storing-function-or-args)

storing-function-or-args names a function or macro that takes one more argument than access-function takes. When setf is given a place that is a call on access-function, it expands into a call on storing-function-or-args that is given all the ar-
arguments to access-function and also, as its last argument, the new value (which must be returned by storing-function-or-args as its value).

For example, the effect of

\begin{verbatim}
(defun symbol-value set)
\end{verbatim}

is built into the Common Lisp system. This causes the form `(setf (symbol-value foo) fu)` to expand into `(set foo fu)`. Note that

```
(defun car rplaca)
```

would be incorrect because rplaca does not return its last argument.

The complex form of defsetf looks like

\begin{verbatim}
(defun access-function storing-function-or-args
  (store-variables) . body)
\end{verbatim}

and resembles defmacro. The body must compute the expansion of a setf of a call on access-function. storing-function-or-args is a lambda list that describes the arguments of access-function and may include &optional, &rest, and &key markers. Optional arguments can have defaults and "supplied-p" flags. store-variables describes the value to be stored into the generalized-variable reference.

The body forms can be written as if the variables in storing-function-or-args were bound to subforms of the call on access-function and the store-variables were bound to the second subform of setf. However, this is not actually the case. During the evaluation of the body forms, these variables are bound to names of temporary variables, generated as if by gensym or gentemp, that will be bound by the expansion of setf to the values of those subforms. This binding permits the body forms to be written without regard for order of evaluation. defsetf arranges for the temporary variables to be optimized out of the final results in cases where that is possible. In other words, an attempt is made by defsetf to generate the best code possible.

Note that the code generated by the body forms must include provision for returning the correct value (the value of store-variables). This is handled by the body forms rather than by defsetf because in many cases this value can be returned at no extra cost, by calling a function that simultaneously stores into the generalized variable and returns the correct value.

Here is an example of the complex form of defsetf.
(defsetf subseq (sequence start &optional end) (new-sequence)
  '(progn (replace ,sequence ,new-sequence
    :start1 ,start :end1 ,end)
    ,new-sequence))

For even more complex operations on setf: See the macro define-setf-method.

**defstruct** options &body items

Defines a record-structure data type. A call to **defstruct** looks like:

```lisp
(defstruct (name option-1 option-2 ...)  
  slot-description-1  
  slot-description-2  
  ...)
```

*name* must be a symbol; it is the name of the structure. It is given a **si:defstruct-description** property that describes the attributes and elements of the structure; this is intended to be used by programs that examine other Lisp programs and that want to display the contents of structures in a helpful way. *name* is used for other things; for more information, see the section "Named Structures".

Because evaluation of a **defstruct** form causes many functions and macros to be defined, you must take care not to define the same name with two different **defstruct** forms. A name can only have one function definition at a time. If a name is redefined, the later definition is the one that takes effect, destroying the earlier definition. (This is the same as the requirement that each **defun** that is intended to define a distinct function must have a distinct name.)

Each *option* can be either a symbol, which should be one of the recognized option names, or a list containing an option name followed by the arguments to the option. Some options have arguments that default; others require that arguments be given explicitly. For more information about options, see the section "Options for **defstruct**".

Each *slot-description* can be in any of three forms:

1:   slot-name
2:   (slot-name default-init)
3:   ((slot-name-1 byte-spec-1 default-init-1)  
    (slot-name-2 byte-spec-2 default-init-2)  
    ...)

Each *slot-description* allocates one element of the physical structure, even though several slots may be in one form, as in form 3 above.

Each *slot-name* must always be a symbol; an accessor function is defined for each slot.

In the example above, form 1 simply defines a slot with the given name *slot-name*. An accessor function is defined with the name *slot-name*. The :conc-name option allows you to specify a prefix and have it concatenated onto the front of all the
slot names to make the names of the accessor functions. Form 2 is similar, but allows a default initialization for the slot. Form 3 lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of defstruct.

For a table of related items: See the section "Functions Related to defstruct Structures".

---

**future-common-lisp:defstruct**

*name-and-options* &body *slot-descriptions*  

Macro

Defines a record-structure data type, and a corresponding class of the same name. You can define methods that specialize on structure classes.

The syntax and semantics of future-common-lisp:defstruct adhere to the draft ANSI Common Lisp specification.

---

**zl:defstruct**

Macro

Defines a record-structure data type. Use the Common Lisp macro defstruct. defstruct accepts all standard Common Lisp options, and accepts several additional options. zl:defstruct is supported only for compatibility with pre-Genera 7.0 releases. See the section "Differences Between defstruct and zl:defstruct".

The basic syntax of zl:defstruct is the same as defstruct: See the macro defstruct.

For information on the options that can be given to zl:defstruct as well as defstruct: See the section "Options for defstruct".

The :export option is accepted by zl:defstruct but not by defstruct. Stylistically, it is preferable to export any external interfaces in the package declarations instead of scattering :export options throughout a program’s source files.

---

**:export**

Exports the specified symbols from the package in which the structure is defined. This option accepts as arguments slot names and the following options: :alterant, :accessors, :constructor, :copier, :predicate, :size-macro, and :size-symbol.

The following example shows the use of :export.
(zl:defstruct (2d-moving-object
 (:type :array)
 :conc-name
 ;; export all accessors and the
 ;; make-2d-moving-object constructor
 (:export :accessors :constructor))
  
mass
 x-pos
 y-pos
 x-velocity
 y-velocity)

See the section "Importing and Exporting Symbols".

**defstruct-define-type** type &body options  

*Macro*

Teaches **defstruct** and **zl:defstruct** about new types that it can use to implement structures.

The body of this function is shown in the following example:

```lisp
(defstruct-define-type type
 option-1
 option-2
 ...)
```

where each **option** is either the symbolic name of an option or a list of the form **(option-name . rest)**. See the section "Options to **defstruct-define-type**".

Different options interpret **rest** in different ways. The symbol **type** is given an **si:defstruct-type-description** property of a structure that describes the type completely.

For a table of related items: See the section "Functions Related to **defstruct** Structures".

**defsusb** function lambda-list &body body  

*Special Form*

Defines inline functions. It is used just like **defun** and does almost the same thing.

```lisp
(defsubst name lambda-list . body)
```

**defsubst** defines a function that executes identically to the one that a similar call to **defun** would define. The difference comes when a function that calls this one is compiled. Then, the call is open-coded by substituting the inline function’s definition into the code being compiled. Such a function is called an inline function. For example, if we define:

```lisp
(defsubst square (x) (* x x))
```

```lisp
(defun foo (a b) (square (+ a b)))
```

then if **foo** is used interpreted, **square** works just as if it had been defined by **defun**. If **foo** is compiled, however, the squaring is substituted into it and it com-
piles just like:
\begin{verbatim}
(defun foo (a b) (* (+ a b) (+ a b)))
\end{verbatim}

\textbf{square} could have been defined as:
\begin{verbatim}
(defun square (x) (* x x))
(proclaim '(inline square))
(defun foo ...)
\end{verbatim}

See the declaration \textbf{inline}.

A similar \textbf{square} could be defined as a macro, with:
\begin{verbatim}
(defmacro square (x) '(* ,x ,x))
\end{verbatim}

When the compiler open-codes an \textbf{inline} function, it binds the argument variables to the argument values with \texttt{let}, so they get evaluated only once and in the right order. Then, when possible, the compiler optimizes out the variables. In general, anything that is implemented as an \textbf{inline} function can be reimplemented as a macro, just by changing the \texttt{defsubst} to a \texttt{defmacro} and putting in the appropriate backquote and commas, except that this does not get the simultaneous guarantee of argument evaluation order and generation of optimal code with no unnecessary temporary variables. The disadvantage of macros is that they are not functions, and so cannot be applied to arguments. Their advantage is that they can do much more powerful things than\textbf{inline} functions can. This is also a disadvantage since macros provide more ways to get into trouble. If something can be implemented either as a macro or as an \textbf{inline} function, it is generally better to make it an \textbf{inline} function.

As with \texttt{defun}, \texttt{name} can be any function spec, but you get the "\texttt{subst}" effect only when \texttt{name} is a symbol.

The difference between an \textbf{inline} function and one not declared inline is the way the calls to them are handled by the compiler. A call to a normal function is compiled as a \textit{closed subroutine}; the compiler generates code to compute the values of the arguments and then apply the function to those values. A call to an \textbf{inline} function is compiled as an \textit{open subroutine}; the compiler incorporates the body forms of the \textbf{inline} function into the function being compiled, substituting the argument forms for references to the variables in the function's \texttt{lambda-list}.

\begin{verbatim}
defsubst-in-flavor (function-name flavor-name) arglist &body body Function
\end{verbatim}
Defines a function inside a flavor to be inline-coded in its callers. There is no analogous form for methods, since the caller cannot know at compile-time which method is going to be selected by the generic function mechanism.

See the section "Defining Functions Internal to Flavors".
For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

**defun**

Defines a function that is part of a program. A defun form looks like:

\[
\text{(defun name lambda-list body ...)}
\]

*name* is the function spec you wish to define as a function. The *lambda-list* is a list of the names to give to the arguments of the function. Actually, it is a little more general than that; it can contain *lambda-list keywords* such as &optional and &rest. (Keywords are explained elsewhere. See the section "Evaluating a Function Form". See the section "Lambda-List Keywords"). Additional syntactic features of defun are explained elsewhere. See the section "Function-Defining Special Forms".

In Genera, defun creates a list which looks like:

\[
\text{(si:digested-lambda...)}
\]

and puts it in the function cell of *name*. *name* is now defined as a function and can be called by other forms.

Examples:

\[
\begin{align*}
\text{(defun addone (x)} \\
& (1+ x))
\end{align*}
\]

\[
\begin{align*}
\text{(defun add-a-number (x &optional (inc 1))} \\
& (+ x inc))
\end{align*}
\]

\[
\begin{align*}
\text{(defun average (&rest numbers &aux (total 0))} \\
& \text{ loop for n in numbers} \\
& \text{ do (setq total (+ total n)))} \\
& \text{// total (length numbers))}
\end{align*}
\]

*addone* is a function that expects a number as an argument, and returns a number one larger. *add-a-number* takes one required argument and one optional argument. *average* takes any number of additional arguments that are given to the function as a list named *numbers*.

If you are using Genera, a declaration (a list starting with declare) can appear as the first element of the body. It is equivalent to a zl:local-declare surrounding the entire defun form. For example:

\[
\begin{align*}
\text{(defun foo (x)} \\
& (declare (special x)) \\
& \text{(bar))} \text{ ; bar uses x free.}
\end{align*}
\]

is equivalent to and preferable to:

\[
\begin{align*}
\text{(zl:local-declare ((special x))} \\
& \text{(defun foo (x)} \\
& \text{(bar))}
\end{align*}
\]
(It is preferable because the editor expects the open parenthesis of a top-level function definition to be the first character on a line, which isn't possible in the second form without incorrect indentation.)

A documentation string can also appear as the first element of the body (following the declaration, if there is one). (It shouldn't be the only thing in the body; otherwise it is the value returned by the function and so is not interpreted as documentation. A string as an element of a body other than the last element is only evaluated for side effect, and since evaluation of strings has no side effects, they are not useful in this position to do any computation, so they are interpreted as documentation.) This documentation string becomes part of the function's debugging info and can be obtained with the function documentation. The first line of the string should be a complete sentence that makes sense read by itself, since there are two editor commands to get at the documentation, one of which is "brief" and prints only the first line. Example:

```lisp
(defun my-append (&rest lists)
  "Like append but copies all the lists.
  This is like the Lisp function append, except that
  append copies all lists except the last, whereas
  this function copies all of its arguments
  including the last one."
  ...
)
```

If you are using CLOE, consider this example:

```lisp
(defun new-function (arg1 arg2 arg3)
  "returns substring of arg1 from position arg2+1 to position arg3-1."
  (declare (string arg1))
  (subseq arg1 (+ arg2 1) (- arg3 1)))
```

**defun-in-flavor** (function-name flavor-name) arglist &body body

Function

Defines an internal function of a flavor. The syntax of defun-in-flavor is similar to the syntax of defmethod; the difference is the way the function is called and the scoping of function-name.

See the section "Defining Functions Internal to Flavors".

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

**zl:defunp**

Macro

Usually when a function uses prog, the prog form is the entire body of the function; the definition of such a function looks like (defun name arglist (prog varlist ...)). Although the use of prog is generally discouraged, prog fans might want to use this special form. For convenience, the zl:defunp macro can be used to produce such definitions. A zl:defunp form such as:
(zl:defunp fctn (args)
  form1
  form2
  ...
  formn)
expands into:
  (zl:defun fctn (args)
    (prog ()
      form1
      form2
      ...
      (return formn)))
You can think of zl:defunp as being like defun except that you can return out of the middle of the function's body.

defvar name &optional initial-value documentation-or-first-key &key :documentation :localize
  Special Form
Declares name special and records its location for the sake of the editor so that you can ask to see where the variable is defined. This is the recommended way to declare the use of a global variable in a program. If a second subform is supplied,
  (defvar name initial-value)
name is initialized to the result of evaluating the form initial-value unless it already has a value, in which case it keeps that value. initial-value is not evaluated unless it is used; this is useful if it does something expensive like creating a large data structure. See the special form sys:defvar-resettable. See the special form sys:defvar-standard.
defvar should be used only at top level, never in function definitions, and only for global variables (those used by more than one function). (defvar foo 'bar) is roughly equivalent to:
  (declare (special foo))
  (if (not (boundp 'foo))
    (setq foo 'bar))
  (defvar variable initial-value "documentation string")
allows you to include a documentation string that describes what the variable is for or how it is to be used. Using such a documentation string is even better than commenting the use of the variable, because the documentation string is accessible to system programs that can show the documentation to you while you are using the machine.
  (defvar variable initial-value :documentation "string")
is an alternate syntax for defvar. The :localize keyword is used for optimizing memory usage at the time of Symbolics distribution world building and is reserved for Symbolics use only.
If `defvar` is used in a patch file or is a single form (not a region) evaluated with
the editor’s compile/evaluate from buffer commands, if there is an initial-value the
variable is always set to it regardless of whether it is already bound. See the sec-
tion "Patch Facility". See the section “Special Forms for Defining Special
Variables”.

```
sys:defvar-resettable name initial-value &optional warm-boot-value documentation
   Special Form
```

Like `defvar`, except that it also maintains a `warm-boot value`. During a warm-boot,
the system sets the variable to its warm-boot value. You can use this function to
assure that a variable is at a pre-determined state even after warm booting. See
the section “Warm Booting”.

```
sys:defvar-standard name initial-value &optional ignore standard-value validation-
   predicate documentation
   Special Form
```

Like `sys:defvar-resettable`, except that it also defines a `standard value` that the
variable should be bound to in command and breakpoint loops. For example, the
standard values of `zl:base` and `zl:ibase` are 10. The `validation-predicate` is used to
ensure that the value of the variable is valid when it is bound in command loops.

For example, `zl:base` is defined like this:
```
(sys:defvar-standard zl:base 10. 10. 10. validate-base)
(defun validate-base (b)
  (and (fixnump b) (< 1 b 37.)))
```

See the section “Standard Variables”.

```
defwhopper
   Special Form
```

The following form defines a whopper for a given `generic-function` when applied to
the specified `flavor`:
```
(defwhopper (generic-function flavor) (arg1 arg2..)
  body)
```

The arguments should be the same as the arguments for any method performing
the generic function.

When a generic function is called on an object of some flavor, and a whopper is
defined for that function, the arguments are passed to the whopper, and the code
of the whopper is executed.

Most whoppers run the methods for the generic function. To make this happen,
the body of the whopper calls one of the following two functions: `continue-
whopper` or `lexpr-continue-whopper`. At that point, the before daemons, primary
methods, and after daemons are executed. Both `continue-whopper` and `lexpr-
continue-whopper` return the values returned by the combined method, so the rest
of the body of the whopper can use those values.
If the whopper does not use `continue-whopper` or `lexpr-continue-whopper`, the methods themselves are never executed, and the result of the whopper is returned as the result of calling the generic function.

Whoppers return their own values. If a generic function is called for value rather than effect, the whopper itself takes responsibility for getting the value back to the caller.

For more information on whoppers, including examples: See the section "Wrappers and Whoppers".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**defwhopper-subst** *(flavor generic-function) lambda-list &body body*  
Macro

Defines a wrapper for the `generic-function` when applied to the given `flavor` by combining the use of `defwhopper` with the efficiency of `defwrapper`.

The following example shows the use of `defwhopper-subst`.

```lisp
(defun defwhopper-subst (flavor generic-function)
  (lambda (arg1 arg2)
    (body))
)
```

The body is expanded in-line in the combined method, providing improved time efficiency but decreased space efficiency, unless the body is small.

See the section "Wrappers and Whoppers".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**defwrapper**  
Macro

Offers an alternative to the daemon system of method combination, for cases in which `:before` and `:after` daemons are not powerful enough.

`defwrapper` defines a macro that expands into code that is wrapped around the invocation of the methods. `defwrapper` is used in forms such as:

```lisp
(defun defwrapper (generic-function flavor) ((arg1 arg2) form)
  body)
```

The wrapper created by this form is wrapped around the method that performs `generic-function` for the given `flavor`. `body` is the code of the wrapper; it is analogous to the body of a `defmacro`. During the evaluation of `body`, the variable `form` is bound to a form that invokes the enclosed method. The result returned by `body` should be a replacement form that contains `form` as a subform. During the evaluation of this replacement form, the variables `arg1`, `arg2`, and so on are bound to the arguments given to the generic function when it is called. As with methods, `self` is implied as the first argument.
The symbol **ignore** can be used in place of the list  (**arg1 arg2**)  if the arguments to the generic function do not matter. This usage is common.

For more information on wrappers, including examples: See the section "Wrappers and Whoppers".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**zl:del pred item list &optional (ntimes -1)**

Function

Returns the list with all occurrences of item removed. pred is used to match elements of the list against item. The argument list is actually modified (replaced) when instances of item are spliced out. zl:del should be used for value, not for effect.

For a table of related items: See the section "Functions for Modifying Lists".

**delete item sequence &key (:test #’eql) :test-not (#’identity) :from-end (:start 0) :end :count**

Function

Removes a sequence of those items in the subsequence of sequence delimited by :start and :end which satisfy the predicate specified by the :test keyword argument. This is a destructive operation. The argument sequence can be destroyed and used to construct the result; however, the returned form may or may not be eq to sequence. The elements that are not deleted occur in the same order in the result that they did in the argument.

For example:

```
(setq nums '(1 2 3)) => (1 2 3)
(delete 1 nums) => (2 3)
nums => (1 2 3)
```

However,

```
(nums => (1 2 3)
(delete 2 nums) => (1 3)
nums => (1 3)
```

item is matched against the elements specified by the test keyword. The item can be any Symbolics Common Lisp object.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

:test specifies the test to be performed. An element of sequence satisfies the test if  (**funcall testfun item (keyfn x)**)  is true. Where testfun is the test function specified by :test, keyfn is the function specified by :key and x is an element of the sequence. The default test is eql.

For example:

```
(delete 4 '(6 1 6 4) :test #’>) => (6 6 4)
```
:test-not is similar to :test, except that the sense of the test is inverted. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is false.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

Example:

(delete 0 '((0 1) (0 1) (1 0)) :key #'second) => ((0 1) (0 1))
(delete 0 #(1 2 1) :key #'(lambda (x) (- x 1))) => #(2)

If the value of the :from-end argument is non-nil, it only affects the result when the :count argument is specified. In that case only the rightmost :count elements that satisfy the predicate are deleted.

For example:

(delete 4 '(4 2 4 1) :count 1) => (2 4 1)
(delete 4 #(4 2 4 1) :count 1 :from-end t) => #(4 2 1)

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(delete 'a #(a b c a)) => #(B C)
(delete 4 '(4 4 1)) => (1)
(delete 4 '(4 1 4) :start 1 :end 2) => (4 1 4)
(delete 4 '(4 1 4) :start 0 :end 3) => (1)

The :count argument, if supplied, limits the number of elements deleted. If more than :count elements of sequence satisfy the predicate, then only the leftmost :count of those elements are deleted. A negative :count argument is equivalent to a :count of 0.

For example:

(delete 4 '(4 2 4 1) :count 1) => (2 4 1)

delete is the destructive version of remove.

For a table of related items: See the section "Functions for Modifying Lists".

For a table of related items: See the section "Sequence Modification".

:delete
Deletes the file open on this stream. The file does not really go away until the stream is closed. You should not use \texttt{:delete}. Instead, use \texttt{delete-file}.

\texttt{zl:delete item list \&optional (ntimes \text{-1})}

Returns \texttt{list} with all occurrences of \texttt{item} removed. \texttt{zl:equal} is used for the comparison. The argument \texttt{list} is actually modified (\texttt{rplacd}ed) when instances of \texttt{item} are spliced out. \texttt{zl:delete} should be used for value, not effect. That is, use:

\begin{verbatim}
(setq a (delete 'b a))
\end{verbatim}

rather than:

\begin{verbatim}
(delete 'b a)
\end{verbatim}

\texttt{ntimes} instances of \texttt{item} are deleted. \texttt{ntimes} is allowed to be zero. If \texttt{ntimes} is greater than or equal to the number of occurrences of \texttt{item} in the list, all occurrences of \texttt{item} in the list are deleted.

Use the Common Lisp function, \texttt{delete}.

For a table of related items: See the section "Functions for Modifying Lists".

For a table of related items: See the section "Sequence Modification".

\texttt{delete-duplicates sequence \&key (test \#eql) :test-not (start 0) :end :from-end :key :replace}

Compares the elements of \texttt{sequence} pairwise, and if any two match, the one occurring earlier in the sequence is discarded. The returned form is \texttt{sequence}, with enough elements removed such that no two of the remaining elements match. \texttt{delete-duplicates} is a destructive function.

\texttt{sequence} can be either a list or a vector (one-dimensional array). Note that \texttt{nil} is considered to be a sequence, of length zero.

\texttt{:test} specifies the test to be performed. An element of \texttt{sequence} satisfies the test if \texttt{(funcall testfun item (keyfn x))} is true. Where \texttt{testfun} is the test function specified by \texttt{:test}, \texttt{keyfn} is the function specified by \texttt{:key} and \texttt{x} is an element of the sequence. The default test is \texttt{eql}.

For example:

\begin{verbatim}
(delete-duplicates '(1 1 2 2 2 3 3 3) :test #'>) => (1 1 2 2 2 3 3 3)
(delete-duplicates '(1 1 2 2 2 3 3 3) :test '=) => (1 2 3)
\end{verbatim}

\texttt{:test-not} is similar to \texttt{:test}, except that the sense of the test is inverted. An element of \texttt{sequence} satisfies the test if \texttt{(funcall testfun item (keyfn x))} is false.

Use the keyword arguments \texttt{:start} and \texttt{:end} to delimit the portion of the sequence to be operated on.

\texttt{:start} and \texttt{:end} must be non-negative integer indices into the sequence. \texttt{:start} must be less than or equal to \texttt{:end}, else an error is signalled. It defaults to zero (the start of the sequence).
:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

```
(delete-duplicates '(a a b b c c)) => (A B C)
(delete-duplicates #(1 1 1 1 1)) => #(1)
(delete-duplicates #(1 1 1 2 2 2) :start 3) => #(1 1 1 2)
(delete-duplicates #(1 1 1 2 2 2) :start 2 :end 4) => #(1 1 1 2 2 2)
```

The function normally processes the sequence in the forward direction, but if a non-nil value is specified for :from-end, processing starts from the reverse direction. If the :from-end argument is true, then the one later in the sequence is discarded.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```
(delete-duplicates '((Smith S) (Jones J) (Taylor T) (Smith S)) :key #'second) => ((JONES J) (TAYLOR T) (SMITH S))
```

When the :replace keyword is specified, elements that stay are moved up to the position of elements that are deleted. :replace is not meaningful if the value of :from-end is t.

Compatibility Note: :replace is a Symbolics extension to Common Lisp, and is not available in CLOE.

For example:

```
(delete-duplicates '((1 a) (2 b) (3 c) (1 d) (4 e) (3 f))
 :key #'car :replace t) =>
  ((1 d) (2 b) (3 f) (4 e))

(delete-duplicates '((1 a) (2 b) (3 c) (1 d) (4 e) (3 f))
 :key #'car :replace nil) =>
  ((2 b) (1 d) (4 e) (3 f))

(delete-duplicates '((1 a) (2 b) (3 c) (1 d) (4 e) (3 f))
 :key #'car :replace nil :from-end t) =>
  ((1 a) (2 b) (3 c) (4 e))
```

**delete-duplicates** is the destructive version of **remove-duplicates**.

For a table of related items: See the section "Sequence Modification".

```
(flavor:method :delete-by-item si:heap) item &optional (equal-predicate #'=) Method
```
Finds the first item that satisfies `equal-predicate`, and deletes it, returning the item and key if it was found, otherwise it signals `si:heap-item-not-found`. `equal-predicate` should be a function that takes two arguments. The first argument to `equal-predicate` is the current item from the heap and the second argument is `item`.

For a table of related items: See the section "Heap Functions and Methods".

```
(delete-if predicate sequence &key :key :from-end (:start 0) :end :count)  Function
```

Removes a sequence of those items in the subsequence of `sequence` delimited by :start and :end which satisfy `predicate`. The elements that are not deleted occur in the same order in the result that they did in the argument. This is a destructive operation. The argument `sequence` can be destroyed and used to construct the result; however, the returned form may or may not be eq to `sequence`.

For example:

```
(setq a-list '(1 a b c)) => (1 A B C)
(delete-if #'numberp a-list) => (A B C)
 a-list => (1 A B C)
```

However,

```
(setq my-list '(0 1 0)) => (0 1 0)
 (delete-if #'zerop my-list) => (1)
  my-list => (0 1)
```

`predicate` is the test to be performed on each element.

`sequence` can be either a list or a vector (one-dimensional array). Note that `nil` is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```
(delete-if #'atom '(((book 1) (math (room c)) (text 3)) :key #'second)
 => (((MATH (ROOM C)))))
```

```
(delete-if #'zerop #'(1 2 1) :key #'(lambda (x) (- x 1)))
 => #((2))
```
If the value of the :from-end argument is non-nil, it only affects the result when the :count argument is specified. In that case only the rightmost :count elements that satisfy the predicate are deleted.

For example:

```lisp
(deDelete-with-if #'numberp '(4 2 4 1) :count 1) => (2 4 1)
(deDelete-with-if #'numberp '(4 2 4 1) :count 1 :from-end t) => (4 2 4)
```

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

```lisp
(deDelete-with-if #'atom '('a 1 "list")) => ('A)
(deDelete-with-if #'numberp '(4 1 4) :start 1 :end 2) => (4 4)
(deDelete-with-if #'evenp '(4 1 4) :start 0 :end 3) => (1)
```

The :count argument, if supplied, limits the number of elements deleted. If more than :count elements of sequence satisfy the predicate, then only the leftmost :count of those elements are deleted. A negative :count argument is equivalent to a :count of 0.

For example:

```lisp
(deDelete-with-if #'oddp '(1 1 2 2) :count 1) => (1 2 2)
```

```lisp
(setq text "Some, text; with too, much punctuation!.")

(deDelete-with-if 'general (lambda (x) (member x '#\( #\? #\! #\;))) text)

a => "Some text with too much punctuation."
```

**delete-if** is the destructive version of **remove-if**.

For a table of related items: See the section "Sequence Modification".

**delete-if-not** predicate sequence &key :key :from-end (:start 0) :end :count

Removes a sequence of those items in the subsequence of sequence delimited by :start and :end which satisfy predicate. The elements that are not deleted occur in the same order in the result that they did in the argument. This is a destructive operation. The argument sequence can be destroyed and used to construct the result; however, the returned form may or may not be eq to sequence.
For example:

```lisp
(setq a-list '('s a b c)) => ('S A B C)
(delete-if-not #'atom a-list) => (A B C)
a-list => ('S A B C)
```

However,

```lisp
(setq my-list '(0 1 0)) => (0 1 0)
(delete-if-not #'zerop my-list) => (0 0)
my-list => (0 1)
```

**predicate** is the test to be performed on each element.

**sequence** can be either a list or a vector (one-dimensional array). Note that **nil** is considered to be a sequence, of length zero.

The value of the keyword argument **:key**, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```lisp
(delete-if-not #'atom '((book 1) (math (room c)) (text 3)) :key #'second)
=> ((BOOK 1) (TEXT 3))
```

```lisp
(delete-if-not #'zerop #(1 2 1) :key #'(lambda (x) (- x 1))) => #(1 1)
```

If the value of the **:from-end** argument is non-nil, it only affects the result when the **:count** argument is specified. In that case only the rightmost **:count** elements that satisfy the predicate are deleted.

For example:

```lisp
(delete-if-not #'oddp '(4 2 4 1) :count 1 )  => (2 4 1)
(delete-if-not #'oddp '(4 2 4 1) :count 1 :from-end t)  => (4 2 1)
```

Use the keyword arguments **:start** and **:end** to delimit the portion of the sequence to be operated on.

**:start** and **:end** must be non-negative integer indices into the sequence. **:start** must be less than or equal to **:end**, else an error is signalled. It defaults to zero (the start of the sequence).

**:start** indicates the start position for the operation within the sequence. **:end** indicates the position of the first element in the sequence beyond the end of the operation. It defaults to **nil** (the length of the sequence).

If both **:start** and **:end** are omitted, the entire sequence is processed by default.

For example:

```lisp
(delete-if-not #'atom ('a 1 "list")) => (1 "list")
(delete-if-not #'numberp '(4 1 4) :start 1 :end 2) => (4 1 4)
(delete-if-not #'evenp '(4 1 4) :start 0 :end 3) => (4 4)
```
The :count argument, if supplied, limits the number of elements deleted. If more than :count elements of sequence satisfy the predicate, then only the leftmost :count of those elements are deleted. A negative :count argument is equivalent to a :count of 0.

For example:

```
(delete-if-not #'oddp '(1 1 2 2) :count 1) => (1 1 2)
```

delete-if-not is the destructive version of remove-if-not.

For a table of related items: See the section "Sequence Modification".

**zl:del-if** pred list

Makes a modified list is made by applying pred (a function of one argument) to all the elements of list and removing the ones for which the predicate returns non-nil. zl:del-if is the destructive version of zl:rem-if, without the extra-lists &rest argument.

For a table of related items: See the section "Functions for Modifying Lists".

**zl:del-if-not** pred list

Applies pred to all elements of list and removes those for which the pred returns nil. Returns the modified list. zl:del-if-not is the destructive version zl:rem-if-not, without the extra-lists &rest argument.

For a table of related items: See the section "Functions for Modifying Lists".

**zl:delq** item list &optional (ntimes -1)

Returns list with all occurrences of item removed. eq is used to match the elements of list against item. The argument list is actually modified (replaced'ed) when instances of item are spliced out. zl:delq should be used for value, not for effect.

For a table of related items: See the section "Functions for Modifying Lists".

**denominator** rational

If rational is a ratio, denominator returns the denominator of rational. If rational is an integer, denominator returns 1.

Examples:

```
(denominator 4/5) => 5
(denominator 3) => 1
(denominator 4/8) => 2
(denominator (/ 12 -17)) => 17
(denominator (rational 0.200)) => 67108864
```

For a table of related items: See the section "Functions that Extract Components From a Rational Number".
**deposit-byte** into-value position size byte-value

*Function*

Like `dpb`, except that instead of using a byte specifier, the bit position and size are passed as separate arguments. The argument order is not analogous to that of `dpb` so that `deposit-byte` can be compatible with older versions of Lisp.

For a table of related items: See the section "Summary of Byte Manipulation Functions".

**deposit-field** newbyte bytespec integer

*Function*

Returns an integer that is the same as `integer` except for the bits specified by `bytespec` which are taken from `newbyte`.

This is like function `dpb` ("deposit byte"), except that `newbyte` is not taken to be right-justified; the `bytespec` bits of `newbyte` are used for the `bytespec` bits of the result, with the rest of the bits taken from `integer`. `integer` must be an integer.

`bytespec` is built using function `byte` with bit size and position arguments.

**deposit-field** could have been defined as follows:

\[
(deposit-field newbyte bytespec integer) \Rightarrow \(dpb (ldb \ bytespec \ newbyte) \ bytespec \ integer)\]

\[
(deposit-field \ 320 \ (\text{byte} \ 3 \ 6) \ 1088) \Rightarrow (+ 1088 \ 256) \Rightarrow 1344\]

\[
(setq \ \text{place-numb} \ \#b100) \Rightarrow 4\]
\[
(deposit-field \ \#b100111 \ (\text{byte} \ 8 \ 3) \ \text{place-numb}) \Rightarrow 36\]
\[
\text{place-numb} \Rightarrow 4\]

Example:

\[
(deposit-field \ \#o230 \ (\text{byte} \ 6 \ 3) \ \#o4567) \Rightarrow \#o4237\]

For a table of related items: See the section "Summary of Byte Manipulation Functions".

**describe** anything &optional no-complaints

*Function*

Provides all the interesting information about any object (except array contents). `describe` knows about arrays, symbols, all types of numbers, packages, stack groups, closures, instances, structures, compiled functions, and locatives, and prints out the attributes of each in human-readable form. For example,

\[
(describe \ 5) \ 5 \text{ is an odd fixnum}\]

Sometimes it describes something that it finds inside something else; such recursive descriptions are indented appropriately. For instance, `describe` of a symbol tells you about the symbol’s value, its definition, and each of its properties. `describe` of a floating-point number shows you its internal representation in a way that is useful for tracking down roundoff errors and the like.
If *anything* is a named-structure, `describe` handles it specially. To understand this: See the section "Named Structures". First it gets the named-structure symbol, and sees whether its function knows about the `:describe` operation. If the operation is known, it applies the function to two arguments: the symbol `:describe`, and the named-structure itself. Otherwise, it looks on the named-structure symbol for information that might have been left by `defstruct`; this information would tell it the symbolic names for the entries in the structure. `describe` knows how to use the names to print out each field’s name and contents.

`describe` describes an instance by sending it the `:describe` message. The default method prints the names and values of the instance variables.

This is the same as the Show Object command.

`describe` always returns its argument, in case you want to do something else to it.

**Compatibility Note:** The optional argument *no-complaints* is an extension to Common Lisp, which might not work in other implementations of Common Lisp.

---

**:`describe`**

*Message*

The object that receives this message should describe itself, printing a description onto the `*standard-output*` stream. The `describe` function sends this message when it encounters an instance.

The `:describe` method of `flavor:vanilla` calls `flavor:describe-instance`, which prints the following information onto the `*standard-output*` stream: a description of the instance, the name of its flavor, and the names and values of its instance variables. It returns the instance. For example:

```
(send cell-object :describe)
-->#<CELL 1160762135>, an object of flavor CELL,
has instance variable values:
  X:                       24
  Y:                       3
  STATUS:                  :ALIVE
  NEXT-STATUS:             unbound
  NEIGHBORS:               unbound
=> #<CELL 1160762135>
```

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

---

**(flavor:method :describe si:heap) &optional (stream zl:standard-output)**

*Method*

Describes the heap, giving the predicate, number of elements, and optionally the contents. If `stream` is given, the output of `:describe` is printed on `stream`.

For a table of related items: See the section "Heap Functions and Methods".

---

**`describe-defstruct instance &optional name`**

*Function*
Takes an instance of a structure and prints out a description of the instance, including the contents of each of its slots. name should be the name of the structure; you must provide this name so that describe-defstruct can know of what structure instance is an instance, and thus figure out the names of instance’s slots.

If instance is a named structure, you do not have to provide name, since it is just the named structure symbol of instance. Normally the describe function calls describe-defstruct if it is asked to describe a named structure; however, some named structures have their own idea of how to describe themselves. See the section "Named Structures".

For a table of related items: See the section "Functions Related to defstruct Structures".

describe-function fspec &key (stream *standard-output*)
Function

Shows the arglist, values and proclaims for the compiled function fspec. The :stream argument enables you to output the description to any stream.

(describe-function 'locativep) =>
Debugging info:
  ARGLIST (OBJECT)
  SYS:FUNCTION-PARENT (LOCATIVEP DEFINE-TYPE-PREDICATE)
Proclaimed properties:
  NOTINLINE
  NIL

See the section "Operations the User Can Perform on Functions".

dbg:describe-global-handlers
Function

Displays the list of conditions for which global handlers have been defined, as well as a list of these handlers.

flavor:describe-instance instance
Function

Prints the following information onto the *standard-output* stream: a description of the instance, the name of its flavor, and the names and values of its instance variables. It returns the instance. For example:

(flavor:describe-instance cell-object)
-->#<CELL 1160762135>, an object of flavor CELL, has instance variable values:
  X: 24
  Y: 3
  STATUS: :ALIVE
  NEXT-STATUS: unbound
  NEIGHBORS: unbound
=> #<CELL 1160762135>
When you use `describe` on an instance, a default method (implemented for `flavor:vanilla`) performs the `flavor:describe-instance` function.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

### clos:describe-object object stream

Provides a mechanism for users to control what happens when `describe` is called for instances of a class. `clos:describe-object` is called by `describe` and should not be called by users.

- **object**: Any Lisp object.
- **stream**: A stream (this cannot be `t` or `nil`).

The default method lists the slot names and values.

The `stream` argument passed to `clos:describe-object` is not necessarily the same as the stream passed to `describe` (it might be an intermediate stream that implements parts of `describe`). Therefore, methods for `clos:describe-object` should not depend on the identity of the `stream`.

### describe-package package

Print a description of `package`'s attributes and the size of its hash table of symbols on `*standard-output*`. `package` can be a package object or the name of a package. The `describe` function calls `describe-package` when its argument is a package.

### zl:desetq (variable-pattern value-pattern)...  

lets you assign values to variables through destructuring patterns. In place of a variable to be assigned, you can provide a tree of variables. The value to be assigned must be a tree of the same shape. The trees are destructured into their component parts, and each variable is assigned to the corresponding part of the value tree.

The first `value-pattern` is evaluated. If `variable-pattern` is a symbol, it is set to the result of evaluating `value-pattern`. If `variable-pattern` is a tree, the result of evaluating `value-pattern` should be a tree of the same shape. The trees are destructured, and each variable that is a component of `variable-pattern` is set to the value that is the corresponding element of the tree that results from evaluating `value-pattern`. This process is repeated for each pair of `variable-pattern` and `value-pattern`. `zl:desetq` returns the last value. Example:

```lisp
(desetq (a b) '((x y) z) c b)
```

=> `z`

`a` is set to `(x y)`, `b` is set to `z`, and `c` is set to `z`. The form returns the value of the last form, which is the symbol `z`. 
**destructuring-bind** pattern datum &body body

_Binds variables to values, using defmacro's destructuring facilities, and evaluates the body forms in the context of those bindings._

First _datum_ is evaluated. If _pattern_ is a symbol, it is bound to the result of evaluating _datum_. If _pattern_ is a tree, the result of evaluating _datum_ should be a tree of the same shape. It signals an error if the trees do not match. The trees are disassembled, and each variable that is a component of _pattern_ is bound to the value that is the corresponding element of the tree that results from evaluating _datum_. Finally, the _body_ forms are evaluated sequentially, the old values of the variables are restored, and the result of the last _body_ form is returned.

As with the pattern in a defmacro form, _pattern_ actually resembles the lambda-list of a function; it can have &-keywords. See the macro defmacro.

Example:

```lisp
(destructuring-bind (a (b) &optional (c 'd))
  '((x y) (z))
  (values a b c))
```

=> (x y) z d

Under Genera, _zl:destructuring-bind_ also exists. It is the same as _destructuring-bind_ except that it does not signal an error if the trees _datum_ and _pattern_ do not match. If not enough values are supplied, the remaining variables are bound to nil. If too many values are supplied, the excess values are ignored.

**math:determinant** matrix

_Returns the determinant of _matrix_. _matrix_ must be a two-dimensional square matrix._

**zl:dfloat** x

_Converts any noncomplex number to a double-precision floating-point number._

For a table of related items: See the section "Functions that Convert Numbers to Floating-point Numbers".

**zl:difference** arg &rest args

_Returns its first argument minus the sum of the rest of its arguments. Arguments of different numeric types are converted to a common type, which is also the type of the result. See the section "Coercion Rules for Numbers"._

_zl:difference_ is similar to the function - used with more than one argument.

For a table of related items, see the section "Arithmetic Functions".
**digit-char weight &optional (radix 10) (style-index 0)**  

Function  

Returns the character that represents a digit with a specified weight weight. Returns nil if weight is not between 0 and (1- radix) or radix is not between 2 and 36.

See the function **digit-char-p**.

For a table of related items, see the section "Character Conversions".

**digit-char-p char &optional (radix 10)**  

Function  

char must be a character object. **digit-char-p** returns the weight of that digit character (a number from zero to one less than the radix) if it is a valid digit in the specified radix. It returns nil if char is not a valid digit in the specified radix; it cannot return t.

- `(digit-char-p #\Q) => nil`  
- `(digit-char-p #\8) => 8`  
- `(digit-char-p (character 'b) 16) => 11`

See the function **digit-char**.

For a table of related items, see the section "Character Predicates".

**:direction**  

Message  

Returns one of the keyword symbols :input, :output, or :bidirectional.

**disassemble function &optional from-pc to-pc**  

Function  

Prints out a human-readable version of the macroinstructions in function. function is either a compiled function, or a symbol or function spec whose definition is a compiled function.

**Compatibility Note**: The optional arguments from-pc and to-pc, are Symbolics extensions to Common Lisp, which might not work in other implementations of Common Lisp. Note that they are not available if you are using CLOE on a 386 based machine.

The CLOE primitive takes a name, a lambda expression, or a compiled function object as an argument. The function definition is retrieved and compiled if not already compiled. The compiled function object is then disassembled, and pretty printed.

**zl:dispatch ppss word &body clauses**  

Special Form  

**(zl:dispatch byte-specifier number clauses...)** is the same as select (not zl:selectq), but the key is obtained by evaluating (ldb byte-specifier number). byte-specifier and number are both evaluated. See the section "Byte Manipulation Functions". Byte specifiers and ldb are explained in that section. Example:
(princ (dispatch 0202 cat-type
     (0 "Siamese."
     (1 "Persian."
     (2 "Alley."
     (3 (ferror nil
          "S is not a known cat type."
          cat-type)))

It is not necessary to include all possible values of the byte that is dispatched on. For a table of related items: See the section "Conditional Functions".

**zl:displace** *form expansion*

Replaces the **car** and **cdr** of *form* so that it looks like:

```lisp
(s:displaced original-form expansion)
```

*form* must be a list. *original-form* is equal to *form* but has a different top-level **cons** so that the replacing mentioned above does not affect it. **s:displaced** is a macro, which returns the **caddr** of its own macro form. So when the **s:displaced** form is given to the evaluator, it "expands" to *expansion*. **zl:displace** returns *expansion*.

**zl:dlet** ((variable-pattern value-pattern)...) **body**...

Binds variables to values, using destructuring, and evaluates the body forms in the context of those bindings. In place of a variable to be assigned, you can provide a tree of variables. The value to be assigned must be a tree of the same shape. The trees are destructured into their component parts, and each variable is assigned to the corresponding part of the value tree.

First the **variable-pattern** is evaluated. If **variable-pattern** is a symbol, it is bound to the result of evaluating the corresponding **value-pattern**. If **variable-pattern** is a tree, the result of evaluating **value-pattern** should be a tree of the same shape. The trees are destructured, and each variable that is a component of **variable-pattern** is bound to the value that is the corresponding element of the tree that results from evaluating **value-pattern**. The bindings happen in parallel; all the **value-patterns** are evaluated before any variables are bound. Finally, the body forms are evaluated sequentially, the old values of the variables are restored, and the result of the last body form is returned. Example:

```lisp
(zl:dlet (((a b) '((x y) z))
          (c 'd))
       (values a b c))
```

=> (x y) z d

**zl:dlet** ((variable-pattern value-pattern)...) **body**...

Special Form
Binds variables to values, using destructuring, and evaluates the body forms in the context of those bindings. In place of a variable to be assigned, you can provide a tree of variables. The value to be assigned must be a tree of the same shape. The trees are destructured into their component parts, and each variable is assigned to the corresponding part of the value tree.

The first value-pattern is evaluated. If variable-pattern is a symbol, it is bound to the result of evaluating value-pattern. If variable-pattern is a tree, the result of evaluating value-pattern should be a tree of the same shape. The trees are destructured, and each variable that is a component of variable-pattern is bound to the value that is the corresponding element of the tree that results from evaluating value-pattern. The process is repeated for each pair of variable-pattern and value-pattern. The bindings happen sequentially; the variables in each variable-pattern are bound before the next value-pattern is evaluated. Finally, the body forms are evaluated sequentially, the old values of the variables are restored, and the result of the last body form is returned. Example:

```lisp
(zl:dlet* (((a b) '((x y) z)) (c b)) (values a b c))
```

=> (x y) z z

**do** vars endtest &body body

*Special Form*

Provides a simple generalized iteration facility, with an arbitrary number of "index variables" whose values are saved when the **do** is entered and restored when it is left, that is, they are bound by the **do**. The index variables are used in the iteration performed by **do**. At the beginning, they are initialized to specified values, and then at the end of each trip around the loop the values of the index variables are changed according to specified rules. **do** allows you to specify a predicate that determines when the iteration terminates. The value to be returned as the result of the form can, optionally, be specified.

**do** looks like this:

```
(do ((var init repeat) ...) 
    (end-test exit-form ...) 
    body...)
```

The first item in the form is a list of zero or more index variable specifiers. Each index variable specifier is a list of the name of a variable **var**, an initial value form **init**, which defaults to **nil** if it is omitted, and a repeat value form **repeat**. If **repeat** is omitted, the **var** is not changed between repetitions. If **init** is omitted, the **var** is initialized to **nil**.

An index variable specifier can also be just the name of a variable, rather than a list. In this case, the variable has an initial value of **nil**, and is not changed between repetitions.

All assignment to the index variables is done in parallel. At the beginning of the first iteration, all the **init** forms are evaluated, then the **vars** are bound to the val-
ues of the init forms, their old values being saved in the usual way. The init forms are evaluated before the vars are bound, that is, lexically outside of the do. At the beginning of each succeeding iteration those vars that have repeat forms get set to the values of their respective repeat forms. All the repeat forms are evaluated before any of the vars is set.

The second element of the do-form is a list of an end-testing predicate form end-test, and zero or more forms, called the exit-forms. This resembles a cond clause. At the beginning of each iteration, after processing of the variable specifiers, the end-test is evaluated. If the result is nil, execution proceeds with the body of the do. If the result is not nil, the exit-forms are evaluated from left to right and then do returns. The value of the do is the value of the last exit-form, or nil if there were no exit-forms (not the value of the end-test as you might expect by analogy with cond).

Note that the end-test gets evaluated before the first time the body is evaluated. do first initializes the variables from the init forms, then it checks the end-test, then it processes the body, then it deals with the repeat forms, then it tests the end-test again, and so on. If the end-test returns a non-nil value the first time, then the body is never processed.

If the second element of the form is (nil), the end-test is never true and there are no exit-forms. The body of the do is executed over and over. The infinite loop can be terminated by use of return or throw.

Example:

(do ((count 1 (+ count 1)))
   (nil) ; Do forever.
   (let ((item (read)))
      (if (null item) (return) (princ item)))) => ABCDEFGNIL ;typed - abcdefg();

If a return special form is evaluated inside the body of a do, the do immediately stops, unbinds its variables, and returns the values given to return. See the special form return.

return and its variants are explained in more detail in that section. go special forms and prog-tags can also be used inside the body of a do and they mean the same thing that they do inside prog forms, but we discourage their use since they make your program complicated and hard to understand.

Examples:
(setq foo-array (make-array '(2 2) :initial-element 'a))
=> #2A((A A) (A A))
(do ((x 0 (+ x 1)) ; prints out array
     (n (array-dimension foo-array 0))
     (= x n))
  (do ((y 0 (+ y 1))
       (n (array-dimension foo-array 1))
       (= y n))
    (princ (aref foo-array x y)))
  ) => AAAA
NIL
(arglist 'cl:array-dimensions) => (ARRAY) and NIL and NIL
(setq a-vector #(1 2 3)) => #(1 2 3)
(do ((i 0 (+ i 1)) ; changes every 2 in vector into a 0
     (n (length a-vector))
     (= i n))
  (if (= 2 (aref a-vector i))
    (setf (aref a-vector i) 0)))
  ) => NIL
A-VECTOR => #(1 0 3)

The following construction exploits parallel assignment to index variables:

(do ((x e (cdr x))
     (oldx x x))
  ((null x))
  body)

On the first iteration, the value of oldx is whatever value x had before the do was entered. On succeeding iterations, oldx contains the value that x had on the previous iteration.

body can contain no forms at all. Very often an iterative algorithm can be most clearly expressed entirely in the repeats and exit-forms of a new-style do, and the body is empty.

The following example is like (maplist 'f x y). (See the section "Mapping").

(do ((x x (cdr x))
     (y y (cdr y))
     (z nil (cons (f x y) z))) ; exploits parallel assignment.
  ((or (null x) (null y))
   (nreverse z)) ; typical use of nreverse.
) ; no do-body required.

For information about a general iteration facility based on a keyword syntax rather than a list-structure syntax:
See the section "The loop Iteration Macro". See the section "The CLOE Loop Iteration Macro".

**Zetalisp Note**: Zetalisp supports another, "old-style" version of do. This form is incompatible with the language specification presented in Guy Steele's *Common Lisp: the Language*.

The older do looks like this:

```
(do var init repeat end-test body...)
```

The first time through the loop var gets the value of the init form; the remaining times through the loop it gets the value of the repeat form, which is reevaluated each time. Note that the init form is evaluated before var is bound, that is, lexically outside of the do. Each time around the loop, after var is set, end-test is evaluated. If it is non-nil, the do finishes and returns nil. If the end-test evaluated to nil, the body of the loop is executed.

If the second element of the form is nil, there is no end-test nor exit-forms, and the body of the do is executed only once. In this type of do it is an error to have repeats. This type of do is no more powerful than let; it is obsolete and provided only for Maclisp compatibility.

return and go can be used in the body. It is possible for body to contain no forms at all.

Examples:

```
(do ((i 0 (+ 1 i)) ; searches list for Dan.
     (names '(Adam Brian Carla Dan Eric Fred) (cdr names)))
   ((null names)
    (if (equal 'Dan (car names))
     (princ "Hey Danny Boooooy "))) => Hey Danny Boooooy NIL

(do ((zz x (cdr zz)))
   ((or (null zz)
     (zerop (f (car zz)))))) ;this applies f to each element of x
   ;continuously until f returns zero.
   ;Note that the do has no body.

(defun list-splice (a b)
  (do ((x a (cdr x))
       (y b (cdr y))
       (xy '()) (append xy (list (car x) (car y))))
      ((endp x) (endp y) (append xy x y))) => LIST-СПЛИСЕ
  (list-splice '(1 2 3) '(a b c)) => (1 A 2 B 3 C)
  (list-splice '(1 2 3) '(a b c d e)) => (1 A 2 B 3 C D E)
```

return forms are often useful to do simple searches:
(setq a-vector #(1 2 3)) => #(1 2 3)
(do ((i 0 (+ i 1))) ; Iterate over the length of vector
    ((and (= 3 (aref a-vector i)) ; If we find an element that = 3
        (return i)))) ; then return its index.
=> 2 ; note (aref a-vector 2) => 3

Example:
(do ((i 5 (+ i 1))
     (list (cdr *data-list*) (cdr list))
     (item (car *data-list*) (car list))
     (>= i (length *data-vector*)) t)
    (unless (= (aref *data-vector* i) item)
        (return nil)))

For a table of related items: See the section "Iteration Functions".

do keyword for loop

do expression

eexpression is evaluated each time through the loop, as shown in the following example:
(defun print-elements-of-list (list-of-elements)
    (loop for element in list-of-elements
do (print element)))
=> PRINT-ELEMENTS-OF-LIST
print-elements-of-list prints each element in its argument, which should be a list. It returns nil.
The forms do and doing are synonymous. Examples

(defun print-list (small-list)
    (loop for element in small-list
do
        (princ element)
        (princ " A ")))
=> PRINT-LIST
(print-list '(1 2 3)) => 1 A 2 A 3 A NIL

This is equivalent to

(defun print-list (small-list)
    (loop for element in small-list
doing
        (princ element)
        (princ " A ")))
=> PRINT-LIST
(print-list '(1 2 3)) => 1 A 2 A 3 A NIL

See the macro loop.
**do* vars endtest &body body**

*Special Form*

Just like `do`, except that the variable clauses are evaluated sequentially rather than in parallel. When a `do` starts, all the initialization forms are evaluated before any of the variables are set to the results; when a `do*` starts, the first initialization form is evaluated, then the first variable is set to the result, then the second initialization form is evaluated, and so on. The stepping forms work analogously.

Examples:

```
(do ( (i 0 (+ 1 i))
     (i 0 (+ 1 i)))
     (= i 10))
(princ i)) => 0123456789NIL
```

```
(do* ( (i 0 (+ 1 i))
     (i 0 (+ 1 i)))
     (= i 10))
(princ i)) => 02468NIL
```

Provides a comprehensive iteration control construct, and is a powerful analog to iteration control loops as found in Algol derivative languages. composed of zero or more variable specifiers, an end test and zero or more result forms, zero or more declarations, and a body.

The variable specifier is a list of variable bindings, including optional initialization values and an optional step form. All the variable binding initializations are executed sequentially, as are evaluation of the step forms. During initialization, later variable specifiers and evaluation of step forms have the ability to refer to the most current value of pre-specified variables. If init is omitted, then the variable is bound to `nil`; if step is omitted, the variable value is not automatically changed during `do*` iterations. Declarations may apply to any of the other three major parts of the `do*` form.

The body of the `do*` form is an implicit tagbody that contains both statement forms and tags that are targets of go statements in the body. The go statements that refer to tags in the body of the `do*` are not allowed in the variable specifiers, end-test, or result forms.

After the variable specifiers are initialized, and after each variable specifier step form evaluation (but before the body forms are evaluated) end-test is evaluated. If the result is `nil`, the body of the `do` is evaluated. If the result is not `nil`, the result forms of the `do*` are evaluated, and the value of the last one is returned as the value of the `do*` form. No returns is `nil`.

The `do*` form is wrapped in an implicit block whose name is `nil`, so that values can be explicitly returned from `do*`, using return.
(do* ((i 5 (+ i 1))
     (list *data-list* (cdr list))
     (item (car list) (car list)))
   ((or (endp list) (>= i (length *data-vector*))) t)
   (unless (= (aref *data-vector* i) item)
     (return nil)))

For a table of related items: See the section "Iteration Functions".

**do-all-symbols** *(var &optional result-form) &body body*  
*Special Form*

Evaluates the *body* forms repeatedly with *var* bound to each symbol present in any package (excluding invisible packages).

When the iteration terminates, *result-form* is evaluated and its values are returned. The value of *var* is **nil** during the evaluation of *result-form*. If *result-form* is not specified, the value returned is **nil**.

The **return** special form can be used to cause a premature exit from the iteration.

The following code **uninterns** all the symbols accessible in **my-package**, and returns the list of symbols:

```
(let ((symbol-list nil))
  (do-symbols (symbol my package symbol-list)
    (unintern symbol)
    (setq symbol-list (cons symbol symbol-list))))
```

**do-external-symbols** *(var &optional pkg result-form) &body body*  
*Special Form*

Evaluates the *body* forms repeatedly with *var* bound to each external symbol exported by *pkg*. *pkg* can be a package object or a string or symbol that is the name of a package, or it can be omitted, in which case the value of *package* is used by default.

When the iteration terminates, *result-form* is evaluated and its values are returned. The value of *var* is **nil** during the evaluation of *result-form*. If *result-form* is not specified, the value returned is **nil**.

The **return** special form can be used to cause a premature exit from the iteration.

The following code makes all the external symbols of the turbine-package accessible in the generator-package:

```
(do-external-symbols (symbol 'turbine-package)
  (import symbol 'generator-package))
```

**do-external-symbols** has an implicit tagbody.

CLOE Note: This is a macro in CLOE.

**do-local-symbols** *(var &optional pkg result-form) &body body*  
*Special Form*
Evaluates the body forms repeatedly with var bound to each symbol present in package. pkg can be a package object or a string or symbol that is the name of a package, or it can be omitted, in which case the value of *package* is used by default.

When the iteration terminates, result-form is evaluated and its values are returned. The value of var is nil during the evaluation of result. If result-form is not specified, the value returned is nil.

The return special form can be used to cause a premature exit from the iteration.

```
(zl:do-named block-name vars endtest &body body)               Special Form
```

Sometimes one do is contained inside the body of an outer do. The return function always returns from the innermost surrounding do, but sometimes you want to return from an outer do while within an inner do. You can do this by giving the outer do a name. You use zl:do-named instead of do for the outer do, and use return-from, specifying that name, to return from the zl:do-named.

The syntax of zl:do-named is like do except that the symbol do is immediately followed by the name, which should be a symbol. Example:

```
(zl:do-named out
  ((x 1 (+ x 1)))
  ((= x 4))
  (do (((y 1 (+ y 1)))
       ((= y 4))
       (if (= y 2) (zl:return-from out (values x y))))) => 1 and 2

(zl:do-named george ((a 1 (1+ a))
                      (d 'foo))
                      ((> a 4) 7)
  (do (((c b (cdr c)))
       ((null c))
       ...
       (return-from george (cons b d))
       ...)))
```

If the symbol t is used as the name, it is made "invisible" to returns; that is, returns inside that zl:do-named return to the next outermost level whose name is not t. (return-from t ...) returns from a zl:do-named named t. You can also make a zl:do-named invisible to returns by including immediately inside it the form (declare (st:invisible-block t)). This feature is not intended to be used by user-written code; it is for macros to expand into.

If the symbol nil is used as the name, it is as if this were a regular do. Not having a name is the same as being named nil.

progs and zl:loops can have names just as dos can. Since the same functions are used to return from all of these forms, all of these names are in the same namespace; a return returns from the innermost enclosing iteration form, no matter
which of these it is, and so you need to use names if you nest any of them within
any other and want to return to an outer one from inside an inner one.

For a table of related items: See the section "Iteration Functions".

\[ \text{zl:do*-named block-name vars endtest &body body} \]

\textit{Special Form}

Just like \texttt{zl:do-named}, except that the variable clauses are evaluated sequentially,
rather than in parallel. See the special form \texttt{do*}.

Examples:

\begin{verbatim}
(zl:do-named who-do
  ((i 0 (+ 1 i))
   (i 0 (+ 1 i)))
  ((= i 10))
  (princ i)) => 0123456789NIL
\end{verbatim}

\begin{verbatim}
(zl:do*-named who-do
  ((i 0 (+ 1 i))
   (i 0 (+ 1 i)))
  ((= i 10))
  (princ i)) => 0123456789NIL
\end{verbatim}

For a table of related items: See the section "Iteration Functions".

\[ \text{do-symbols (var &optional pkg result-form) &body body} \]

\textit{Special Form}

Evaluates the \textit{body} forms repeatedly with \textit{var} bound to each symbol accessible in
\textit{pkg}. \textit{pkg} can be a package object or a string or symbol that is the name of a
package, or it can be omitted, in which case the value of \texttt{*package*} is used by
default.

When the iteration terminates, \textit{result-form} is evaluated and its values are returned.
The value of \textit{var} is \texttt{nil} during the evaluation of \textit{result}. If \textit{result-form} is not speci-
fied, the value returned is \texttt{nil}.

The \texttt{return} special form can be used to cause a premature exit from the iteration.

\[ \text{dbg:document-proceed-type condition proceed-type stream} \]

\textit{Generic Function}

Prints out a description of what it means to proceed, using the given \textit{proceed-type},
from this condition, on \textit{stream}. This is used mainly by the Debugger to create its
prompt messages. Phrase such a message as an imperative sentence, without any
leading or trailing \texttt{\#\textbackslash return} characters. This sentence is for the human users of
the machine who read this when they have just been dumped unexpectedly into the
Debugger. It should be composed so that it makes sense to a person to issue that
sentence as a command to the system.

The compatible message for \texttt{dbg:document-proceed-type} is:
:document-proceed-type

For a table of related items, see the section "Basic Condition Methods and Init Options".

**dbg:document-special-command** condition special-command

*Generic Function*

Prints the documentation of `special-command` onto stream. If you don’t provide your own method explicitly, the default handler uses the documentation string from the `dbg:special-command` method. You can, however, provide this method in order to print a prompt string that has to be computed at run-time. This is analogous to `dbg:document-proceed-type`. The syntax is:

```lisp
(defmethod (dbg:document-special-command my-flavor :my-command-keyword)
  (stream)
  body...)
```

The compatible message for `dbg:document-special-command` is:

:document-special-command

For a table of related items: See the section "Debugger Special Command Functions".

**documentation** name &optional (type 'defun)

*Function*

Finds the documentation string of the symbol, `name`, which is stored in various different places depending on the symbol type. If there is no documentation, `nil` is returned.

Symbolics Common Lisp provides the optional argument `type`. `type` can be `variable`, `function`, `structure`, `type`, or `setf`, according to the construct represented by `name`. `Type` is a required argument in other implementations of Common Lisp, including CLOE Runtime.

If you are using CLOE, consider the following example:

```lisp
(defstruct person "The physical parts of a person"
  (head *default-head*)
  (right-arm *default-right-arm*)
  (left-arm *default-left-arm*)
  (right-leg *default-right-leg*)
  (left-leg *default-left-leg*)
  (other '() :type list))
```

```lisp
(dolist var listform &optional resultform) &body forms
```
A convenient abbreviation for the most common list iteration.

**dolist** performs *forms* once for each element in the list that is the value of *list-form*, with *var* bound to the successive elements.

You can use **return** and **go** and **prog**-tags inside the body, as with **do**.

**dolist** returns **nil**, or the value of *resultform*, if the latter is specified.

Examples:

```
(dolist (people '(mary ann claire cindy) 4) (print people)) =>
MARY
ANN
CLAIRE
CINDY

(dolist (z '(1 2 3 4) "hi") (princ (+ z 2))) => 3456"hi"

(dolist (j '(1 2 3 4) t) (princ (- 1 j)) (if (= j 3)(return)))
=> 0-1-2NIL
```

For a table of related items: See the section "Iteration Functions".

**zl:dolist** *(var form) &body body*

A convenient abbreviation for the most common list iteration. **zl:dolist** performs *body* once for each element in the list that is the value of *form*, with *var* bound to the successive elements.

Examples:

```
(zl:dolist (people '(mary ann claire cindy)) (print people)) =>
MARY
ANN
CLAIRE
CINDY
NIL

(zl:dolist (z '(1 2 3 4) (+ z 2))) => 3456NIL

(zl:dolist (j '(1 2 3 4) (- 1 j)) (= j 3)(return)))
=> 0-1-2NIL
```

**Where**

```
(zl:dolist (item (frobs foo))
 (mung item))
```
is equivalent to:

```lisp
(do ((lst (frobs foo) (cdr lst))
     (item))
    ((null lst))
     (setq item (car lst))
     (mung item))
```

except that the name `lst` is not used. You can use `return` and `go` and `prog`-tags inside the body, as with `do`. `zil:dolist` forms return `nil` unless returned from explicitly with `return`.

See the special form `dolist`.

For a table of related items: See the section "Iteration Functions".

---

**See Also**: `CLtL 126, do, do*, loop, tagbody, dotimes`

---

### dotimes

**Special Form**

A convenient abbreviation for the most common integer iteration.

**dotimes** performs **forms** the number of times given by the value of **countform**, with **var** bound to 0, 1, and so forth on successive iterations.

You can use `return` and `go` and `prog`-tags inside the body, as with `do`.

The function returns `nil`, or the value of `resultform` if the latter is specified.

**Examples:**
(dotimes (i 5 10)
  (princ i)(princ " ") => 0 1 2 3 4 10

(dotimes (j 5 t)
  (princ j)(if (= j 3) (return))) => 0123NIL

Note that in CLOE, the iteration control variable var is required to take on only fixnum values.

For a table of related items: See the section "Iteration Functions".

zl:dotimes (var form) &body body

A convenient abbreviation for the most common integer iteration. zl:dotimes performs body the number of times given by the value of count, with index bound to 0, 1, and so forth on successive iterations.

Example:

(zl:dotimes (i 5)
  (princ i)(princ " ") => 0 1 2 3 4 NIL

(zl:dotimes (j 5)
  (princ j)(if (= j 3) (return))) => 0123NIL

Where

(zl:dotimes (i (/\ m n))
  (frob i))

is equivalent to:

(do ((i 0 (1+ i))
     (count (/\ m n)))
    ((≥ i count))
    (frob i))

except that the name count is not used. Note that i takes on values starting at 0 rather than 1, and that it stops before taking the value (/\ m n) rather than after.

You can use return and go and prog-tags inside the body, as with do. zl:dotimes forms return nil unless returned from explicitly with return. For example:

(zl:dotimes (i 5)
  (if (eq (aref a i) 'foo)
      (return i))

This form searches the array that is the value of a, looking for the symbol foo. It returns the fixnum index of the first element of a that is foo, or else nil if none of the elements are foo.

See the special form dotimes.

For a table of related items: See the section "Iteration Functions".
provides an control device for iteration over a sequence of natural numbers. It is composed of a single variable specifier, zero or more *declarations*, and an implicit *tagbody*.

The variable specifier is composed a binding of a variable to zero, and specification of a form, *countform* which must evaluate to an integer. If *countform* is negative or zero, *result* is evaluated and *dotimes* exits. After each iteration, the value of the control variable is incremented by one. A single, optional *result* form is permitted, and is the value returned by *dotimes*. If *result* is omitted, *dotimes* returns nil (unless an explicit return is done).

*Declarations* may apply to either of the other major parts of the *dotimes* form.

The body of the *dotimes* form is an implicit *tagbody*, containing both statement forms, and tags which are targets of *go* statements in the body. *Go* statements referring to *tags* in the body of the *dotimes* are not allowed in the variable specifier. The body of the *dotimes* is evaluated once for each integer value of the control variable, up to but not including the number returned by *countform*. After the last iteration, and during the evaluation of *result*, the control variable *countform* has a value, which is the number of times the body was evaluated.

The *dotimes* form is wrapped in an implicit block whose name is nil, so that values can be explicitly returned from *dotimes*, using *return*.

```
(dotimes (i 20 t)
  (unless (= (aref *data-vector-a* i) (aref *data-vector-b* i))
    (return nil)))
```

See Also: CLtL 126, *do*, *do*, *loop*, *tagbody*, *dolist*

**double-float**

*double-float* is the type specifier symbol for the predefined Lisp double-precision floating-point number type.

The type *double-float* is a subtype of the type *float*. In Symbolics Common Lisp, the type *double-float* is equivalent to the type *long-float*.

The type *double-float* is disjoint with the types *short-float*, and *single-float*.

Examples:

```
(typep -13D2 'double-float) => T
(zl:typenp -12D4) => :DOUBLE-FLOAT
(subtypep 'double-float 'float) => T and T ; subtype and certain
(commonp 0d0) => T
(sys:double-float-p 6.03e23) => NIL
(sys:double-float-p 1.5d9) => T
(equal-typep 'double-float 'long-float) => T
(sys:type-arglist 'double-float) => NIL and T
```

See the section "Data Types and Type Specifiers".
See the section "Numbers".

**double-float-epsilon**

*Constant*

The value of this constant is the smallest positive floating-point number \( e \) of a format such that it satisfies the expression:

\[
\text{(not (= (float 1 \( e \)) (+ (float 1 \( e \)) \( e \))))}
\]

The current value of **double-float-epsilon** is: 1.1102230246251568d-16.

**double-float-negative-epsilon**

*Constant*

The value of this constant is the smallest positive floating-point number \( e \) of a format such that it satisfies the expression:

\[
\text{(not (= (float 1 \( e \)) (- (float 1 \( e \)) \( e \))))}
\]

The current value of **double-float-negative-epsilon** is: 5.551115123125784d-17.

**sys:double-float-p** *object*

*Function*

Returns \( t \) if \( object \) is a double-precision floating-point number, otherwise \( \text{nil} \).

For a table of related items, see the section "Numeric Type-checking Predicates".

**dpb** *newbyte bytespec integer*

*Function*

The name of this function stands for "Deposit byte".

Returns a number that is the same as \( integer \) except in the bits specified by \( bytespec \).

\( bytespec \) is built using function **byte** with bit \( size \) and \( position \) arguments. Here \( size \) indicates the number of low bits of \( newbyte \) to be placed in the result.

\( newbyte \) is interpreted as being right-justified, as if it were the result of \( \text{ldb} \) ("load byte").

\( integer \) must be an integer.

Examples:

\[
\begin{align*}
\text{dpb 1 (byte 1 2) 1} & \Rightarrow 5 \\
\text{dpb 0 (byte 1 31.) -1_31.} & \Rightarrow -4294967296. \quad \text{;; a bignum \(-1_{32}\)} \\
\text{dpb -1 (byte 40. 0) -1_32.} & \Rightarrow -1. \\
\text{dpb #b030 (byte 6 3) #o4567} & \Rightarrow #o4307 \\
\text{dpb 320 (byte 7 0) 1024} & = (\text{dpb (logior 256 64) (byte 7 0) 1024}) \\
& = (\text{dpb #b10100000 (byte 7 0) #b1000000000}) = (\text{logior 1024 64}) \Rightarrow 1088
\end{align*}
\]

For a table of related items: See the section "Summary of Byte Manipulation Functions".
**dribble** &optional pathname editor-p

Function

Opens **pathname** as a "dribble file". It rebinds \*standard-input\*, \*standard-output\*, \*trace-output\*, \*error-output\*, and \*query-io\* so that all of the terminal interaction is directed to the file as well as to the terminal. If **editor-p** is non-nil, it does not open **pathname** on the file computer, instead it directs the terminal interaction into a Zmacs buffer whose name is **pathname**, creating it if it does not exist.

To terminate the recording, reset the I/O streams, and close the file (if any), call **dribble** again with no arguments:

(dribble)

**Compatibility Note**: The optional argument **editor-p** is a Symbolics extension to Common Lisp which might not work in other implementations of Common Lisp, and does not work in CLOE Runtime.

**zl:dribble-end**

Function

Closes the file opened by **zl:dribble-start** and resets the I/O streams.

**zl:dribble-start** pathname &optional editor-p (concatenate-p t) (debugger-p nil)

Function

Opens **pathname** as a "dribble file". It rebinds \*standard-input\*, \*standard-output\*, \*trace-output\*, \*error-output\*, and \*query-io\* so that all of the terminal interaction is directed to the file as well as to the terminal. If **editor-p** is non-nil, it does not open **pathname** on the file computer, instead it directs the terminal interaction into a Zmacs buffer whose name is **pathname**, creating it if it does not exist.

**sys:dynamic-closure**

Type Specifier

**sys:dynamic-closure** is the type specifier symbol for the predefined Lisp object of that name.

See the section "Data Types and Type Specifiers". See the section "Scoping".

Examples:

```lisp
(setq four
  (let ((x 4))
    (closure '(x) 'zerop))) => #<DTP-CLOSURE 1510647>

(typep four 'sys:dynamic-closure) => T

(subtypep 'sys:dynamic-closure 'common) => NIL and NIL
```

**dynamic-closure-alist** closure

Function
Returns an alist of (symbol . value) pairs describing the bindings which the dynamic closure performs when it is called. This list is not the same one that is actually stored in the closure; that one contains pointers to value cells rather than symbols, and dynamic-closure-alist translates them back to symbols so you can understand them. As a result, clobbering part of this list does not change the closure.

If any variable in the closure is unbound, this function signals an error. See the section "Dynamic Closure-Manipulating Functions".

dynamic-closure-variables closure

Function

Creates and returns a list of all of the variables in the dynamic closure closure. It returns a copy of the list that was passed as the first argument to make-dynamic-closure when closure was created. See the section "Dynamic Closure-Manipulating Functions".

ecase object &body body

Special Form

The name of this function stands for "exhaustive case" or "error-checking case".

Structurally ecase is much like case, and it behaves like case in selecting one clause and then executing all consequents of that clause. However, ecase does not permit an explicit otherwise or t clause. The form of ecase is as follows:

(ecase key-form
  (test consequent consequent ...)
  (test consequent consequent ...)
  (test consequent consequent ...)
  ...
)

The first thing ecase does is to evaluate object, to produce an object called the key object.

Then ecase considers each of the clauses in turn. If key is eql to any item in the clause, ecase evaluates the consequents of that clause as an implicit progn.

ecase returns the value of the last consequent of the clause evaluated, or nil if there are no consequents to that clause.

The keys in the clauses are not evaluated; literal key values must appear in the clauses. It is an error for the same key to appear in more than one clause. The order of the clauses does not affect the behavior of the ecase construct.

If there is only one key for a clause, that key can be written in place of a list of that key, provided that no ambiguity results. Such a "singleton key" can not be nil (which is confusable with nil, a list of no keys), t, otherwise, or a cons.

If no clause is satisfied, ecase uses an implicit otherwise clause to signal an error with a message constructed from the clauses. It is not permissible to continue from this error. To supply your own error message, use case with an otherwise clause containing a call to error.
Examples:

(let ((num 24))
  (ecase num
    ((1 2 3) "integer")
    ((4 5 6) "integer"))) => non-proceedable error is signalled

(let ((num 3))
  (ecase num
    ((1 2) "one two")
    ((3 4 5 6) (princ "numbers") (princ " three") (terpri) )
    (t "not today"))) => numbers three
  T

For a table of related items: See the section "Conditional Functions".

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

\textbf{eighth list}  \hspace{1cm} \textit{Function}

Returns the eighth element of the list \textit{list}. \textbf{eighth} is equivalent to (\texttt{nth 7 list}).

\begin{verbatim}
For example:
  (setq letters '(a b c d e f g h i j)) =>
  (A B C D E F G H I J)

  (eighth letters) => H
\end{verbatim}

This function is provided because it makes more sense than using \texttt{nth} when you are thinking of the argument as a list rather than just as a cons.

For a table of related items: See the section "Functions for Extracting from Lists".

\textbf{elt sequence index}  \hspace{1cm} \textit{Function}

Extracts an element from \textit{sequence} at position \textit{index}. Returns a new sequence. \textit{sequence} can be either a list or a vector (one-dimensional array). Note that \texttt{nil} is considered to be a sequence, of length zero.

\textit{index} must be a non-negative integer less than the length of \textit{sequence} as returned by \texttt{length}. The first element of a sequence has index 0.

\begin{verbatim}
For example:
  (setq bird-list '(heron stork pelican turkey)) =>
  (HERON STORK PELICAN TURKEY)
\end{verbatim}
(elt bird-list 2) => PELICAN

(equalp (elt bird-list 2) (third bird-list)) => T

Note that elt observes the fill pointer in those vectors that have fill pointers. The array-specific function aref can be used to access vector elements that are beyond the vector's fill pointer.

setf can be used with elt to destructively replace a sequence element with a new value. For example:

(setq bird-list (elt bird-list 2) 'hawk) => HAWK

bird-list => (HERON STORK HAWK TURKEY)

The following example demonstrates the use of elt to reference array components of either type list or type vector.

(setq seqarr

(make-array 5 :element-type 'sequence
 :initial-contents
 '((a b c)
 , (vector 'd 'e 'f)
  (x y)
  (y z)
  (z)))))

(elt (aref seqarr 0) 1) => B

(elt (aref seqarr 1) 1) => E

(setf (elt (aref seqarr 0) 1) 'g) => G

(aref seqarr 1) => #(D G F)

For a table of related items: See the section "Sequence Construction and Access".

(flavor:method :empty-p si:heap)  Method

Returns t if the heap is empty, otherwise returns nil.

For a table of related items: See the section "Heap Functions and Methods".

si:enable-who-calls &optional mode  Function

mode describes how the who-calls database should record the callers of any function. For more information about the who-calls database, see the section "Enabling the Who-Calls Database".

:all If you want to include callers of the Symbolics-supplied software (that is, software contained in the distribution world) in
the database, use :all. This enables you to create the database once and then save it when you save the world. (When used with this argument, si:full-gc would discard the existing database and then remake it).

:all-remake

Includes callers of the Symbolics-supplied and site-specific software in the database. Use this if you do not want to perform a si:full-gc. (When used with this argument, si:full-gc would discard the existing database and then remake it).

:new

Enables the who-calls database to record the callers in any layered products, special software, or programs loaded into the world (after the site has been set). The Set Site command uses this argument by default. :new does not cause the callers of software in the distribution world to be recorded.

:all-no-make

Enables the who-calls database to record the callers in any layered products, special software, or programs loaded into the world (after the site has been set), and does not cause the callers of software in the distribution world to be recorded until si:full-gc is performed. Once si:full-gc is performed, those callers (for software in the distribution world) are recorded.

:explicit

If you want only explicitly-named files to be in the database, use the function si:enable-who-calls with the argument :explicit.

Note: Creating a full database takes a long time and about 2000 pages of storage.

si:encapsulate function outer-function type body &optional extra-debugging-info

Macro

A call to si:encapsulate looks like:

(si:encapsulate function-spec outer-function type body-form extra-debugging-info)

All the subforms of this macro are evaluated. In fact, the macro could almost be replaced with an ordinary function, except for the way body-form is handled.

function-spec evaluates to the function spec whose definition the new encapsulation should become. outer-function is another function spec, which should often be the same one. Its only purpose is to be used in any error messages from si:encapsulate.

type evaluates to a symbol that identifies the purpose of the encapsulation; it says what the application is. For example, it could be advise or trace. The list of possible types is defined by the system because encapsulations are supposed to be kept in an order according to their type. See the variable si:encapsulation-standard-order. type should have an si:encapsulation-grind-function property that tells grinddef what to do with an encapsulation of this type.
body-form is a form that evaluates to the body of the encapsulation-definition, the code to be executed when it is called. Backquote is typically used for this expression. See the section "Backquote-Comma Syntax". si:encapsulate is a macro because, while body is being evaluated, the variable si:encapsulated-function is bound to a list of the form (function uninterned-symbol), referring to the uninterned symbol used to hold the prior definition of function-spec. If si:encapsulate were a function, body-form would just get evaluated normally by the evaluator before si:encapsulate ever got invoked, and so there would be no opportunity to bind si:encapsulated-function. The form body-form should contain (apply si:encapsulated-function arglist) somewhere if the encapsulation is to live up to its name and truly serve to encapsulate the original definition. (The variable arglist is bound by some of the code that the si:encapsulate macro produces automatically. When the body of the encapsulation is run, arglist's value is the list of the arguments that the encapsulation received.)

extra-debugging-info evaluates to a list of extra items to put into the debugging info alist of the encapsulation function (besides the one starting with si:encapsulated-definition that every encapsulation must have). Some applications find this useful for recording information about the encapsulation for their own later use.

When a special function is encapsulated, the encapsulation is itself a special function with the same argument quoting pattern. (Not all quoting patterns can be handled; if a particular special form's quoting pattern cannot be handled, si:encapsulate signals an error.) Therefore, when the outermost encapsulation is started, each argument has been evaluated or not as appropriate. Because each encapsulation calls the prior definition with apply, no further evaluation takes place, and the basic definition of the special form also finds the arguments evaluated or not as appropriate. The basic definition can call eval on some of these arguments or parts of them; the encapsulations should not.

Macros cannot be encapsulated, but their expander functions can be; if the definition of function-spec is a macro, then si:encapsulate automatically encapsulates the expander function instead. In this case, the definition of the uninterned symbol is the original macro definition, not just the original expander function. It would not work for the encapsulation to apply the macro definition. So during the evaluation of body-form, si:encapsulated-function is bound to the form (cdr (function uninterned-symbol)), which extracts the expander function from the prior definition of the macro.

Because only the expander function is actually encapsulated, the encapsulation does not see the evaluation or compilation of the expansion itself. The value returned by the encapsulation is the expansion of the macro call, not the value computed by the expansion.

si:encapsulation-standard-order

The value of this variable is a list of the allowed encapsulation types, in the order that the encapsulations are supposed to be kept in (innermost encapsulations first). If you want to add new kinds of encapsulations, you should add another symbol to
this list. Initially its value is:

(advise breakon trace si:rename-within)

**advise** encapsulations are used to hold advice. **breakon** and **trace** encapsulations are used for implementing tracing. **si:rename-within** encapsulations are used to record the fact that function specs of the form (**:within within-function altered-function**) have been defined. The encapsulation goes on **within-function**. See the section "Rename-Within Encapsulations".

**endp object**

Function

Tests for the end of a list. Returns **nil** when applied to a cons, and **t** when it is applied to **nil**. **endp** signals an error when object is not a cons or **nil**.

Example:

```
(advise breakon trace si:rename-within)
```

```
(advise breakon trace si:rename-within)
```

returns **nil**, since **endp** here is applied to a list. But:

```
(endp ())
```

returns **t**, since **endp** is applied to an empty list.

Under Clos on the 386, **endp** signals an error, when the safety level is three, for an atomic argument other than **nil**. If the safety level is less than three, **endp**, depending upon the values of other optimization parameters, might signal an error when given inappropriate arguments.

```
(setq a '(a1 a2 a3 a4)) => (A1 A2 A3 A4)
(endp a) => NIL
(endp (cdddr a)) => NIL
(endp (cddddr a)) => T
```

Because of its type checking properties, **endp** is the preferred predicate when testing for the end of a list.

```
(proclaim '(optimize (safety 3)))
(defun my-reverse-list( list )
  "reverses a true list, endp signals error"
  " if arg is not true list."
  (let ((curcon nil)
        (ptr list))
    (tagbody loop
      (unless (endp ptr)
        (setq curcon (cons (car ptr) curcon))
        (setq ptr (cdr ptr))
        (go loop))))
    curcon))
```

```
(my-reverse-list '(a b c d)) => (D C B A)
```

```
(my-reverse-list 'abcd)
```

**ERROR**: **ARGUMENT NOT A LIST**
For a table of related items: See the section "Predicates that Operate on Lists".


Function

Defines a new generic function, or modifies an existing one. This function is part of the underlying implementation of clos:defgeneric and clos:defmethod. clos:ensure-generic-function returns the generic function object.

function-specifier Either a symbol or a list of the form (future-common-lisp:setf symbol); this names the generic function.

keywords The keywords have the same semantics as the options documented in clos:defgeneric.

The :method-class and :generic-function-class keywords can be either class objects or names (in clos:defgeneric, they must be names). Symbolics CLOS supports only the value clos:standard-method for :method-class and the value clos:standard-generic-function for :generic-function-class.

There is an additional keyword, :environment, which is the same as the &environment argument to macro expansion functions. It is typically used to distinguish between compile-time and run-time environments.

If function-specifier does not name a generic function (or any other kind of function), then a new generic function is created. If function-specifier names an ordinary Lisp function, a macro, or a special form, an error is signaled.

If function-specifier names an existing generic function, then that generic function is modified, according to the keyword arguments :argument-precedence-order, :declare, :documentation, :generic-function-class, :method-combination, and :method-class. If any of those keyword values differ from the corresponding options in the generic function, then the keyword value replaces the existing option.

If the :lambda-list keyword is unsupplied and the generic function already exists, then the existing lambda-list is left alone. If the :lambda-list keyword is unsupplied and the generic function does not already exist, then the generic function is created with no lambda-list; the lambda-list will be created from the first method defined for the generic function. If the :lambda-list keyword is supplied with a value of nil, then the generic function accepts no arguments.

An error is signaled if the value of :lambda-list is not congruent with the lambdalists of all existing methods.

&environment Lambda List Keyword

This keyword is used with macros only. It should be followed by a single variable that is bound to an environment representing the lexical environment in which the
macro call is to be interpreted. This environment is not required to be the complete lexical environment; it should be used only with the function macroexpand for the sake of any local macro definitions that the macrolet construct may have established within that lexical environment. &environment is useful primarily in the rare cases where a macro definition must explicitly expand any macros in a subform of the macro call before computing its own expansion.

:eof

Message

Indicates the end of data on an output stream. This is different from :close because some devices allow multiple data files to be transmitted without closing. :close implies :eof when the stream is an output stream and the close mode is not :abort.

eq x y

Function

Returns t if and only if x and y are the same object. Note that things that print the same are not necessarily eq to each other. In particular, numbers with the same value need not be eq, and two similar lists are usually not eq. Examples:

\[
\begin{align*}
(eq 'a 'b) &= \texttt{nil} \\
(eq 'a 'a) &= \texttt{t} \\
(eq (cons 'a 'b) (cons 'a 'b)) &= \texttt{nil} \\
(setq x (cons 'a 'b)) (eq x x) &= \texttt{t}
\end{align*}
\]

Note that in Symbolics Common Lisp and CLOE equal fixnums are eq; this is not true in Maclisp. Equality does not imply eqness for other types of numbers. To compare numbers, use =.

eq is implemented by comparing pointers. Certain datatypes, such as small integers and characters, can be stored locally in a pointer space. For these data objects, the same number or character object will yield true when compared by eq. However, numbers with the same value are usually not the same object. Exercise caution in these cases. Consider this function when comparing numbers and characters.

See the section "Numeric Comparisons".

si:eq-hash-table

Flavor

Creates an old style Zetalisp hash table using the eq function for comparison of the hash keys. This flavor is superseded by table:basic-table. It accepts the following init options:

:size

Sets the initial size of the hash table in entries, as an integer. The default is 100 (decimal). The actual size is rounded up from the size you specify to the next size that is good for the hashing algorithm. An automatic rehash of the hash table might occur before this many entries are stored in the table depending upon the keys being stored.
:area
Specifies the area in which the hash table should be created. This is just like the :area option to zl:make-array. See the function zl:make-array. The default is sys:working-storage-area.

growth-factor
Specifies how much to increase the size of the hash table when it becomes full. If it is an integer, the hash table is increased by that number. If it is a floating-point number greater than one, the new size of the hash table is the old size multiplied by that number.

:rehash-before-cold
Causes zl:disk-save to rehash this hash table if its hashing has been invalidated. (This is part of the before-cold initializations.) Thus every user of the saved world does not have to waste the overhead of rehashing the first time they use the hash table after cold booting.

For eq hash tables, the hashing is invalidated whenever garbage collection or world compression occurs because the hash function is sensitive to addresses of objects, and those operations move objects to different addresses. For equal hash tables, the hash function is not sensitive to addresses of objects that sxhash knows how to hash but it is sensitive to addresses of other objects. The hash table remembers whether it contains any such objects.

Normally a hash table is automatically rehashed "on demand" the first time it is used after the hashing has become invalidated. This first :get-hash operation is therefore much slower than normal.

The :rehash-before-cold option should be used on hash tables that are a permanent part of your world, likely to be saved in a world saved by zl:disk-save, and to be touched by users of that world. This applies both to hash tables in Genera and to hash tables in user-written subsystems saved in a world.

**eql x y**

*Function*

Returns t if its arguments are eq, if they are numbers of the same type with the same value, or if they are character objects that represent the same character. The predicate = compares the values of two numbers even if the numbers are of different types.

**Examples:**
\[(\text{eql } \text{a } \text{a}) \Rightarrow \text{t}\]
\[(\text{eql } 3 \ 3) \Rightarrow \text{t}\]
\[(\text{eql } 3.0 \ 3.0) \Rightarrow \text{t}\]
\[(\text{eql } \#\text{/a} \#\text{/a}) \Rightarrow \text{t}\]
\[(\text{eql } (\text{cons } \text{a } \text{b}) (\text{cons } \text{a } \text{b})) \Rightarrow \text{nil}\]
\[(\text{eql } \text{"foo" } \text{"FOO"}) \Rightarrow \text{nil}\]

The following expressions might return either \text{t} or \text{nil}:

\[(\text{eql } '(\text{a . b}) '(\text{a . b}))\]
\[(\text{eql } \text{"foo" } \text{"foo"})\]

In Symbolics Common Lisp:

\[(\text{eql } 1.0s0 \ 1.0d0) \Rightarrow \text{nil}\]
\[(\text{eql } 0.0 \ -0.0) \Rightarrow \text{nil}\]

\textbf{equal \ x \ y}

\textit{Function}

Returns \text{t} if its arguments are structurally similar (isomorphic) objects. If the two objects are \text{eql}, then they are also \text{equal}. If the objects are of different data types, then they are not \text{equal}.

Objects of each data type are compared differently for \text{equal}. \text{equal} returns \text{t} in the following cases:

\textbf{Conses} \quad \text{The two cars are \textbf{equal} and the two cdrs are \textbf{equal}.}

\textbf{Strings} \quad \text{The strings are of the same length, and corresponding characters of each string are \textbf{char}=.}

\textbf{Bit-vectors} \quad \text{The vectors are of the same length, and corresponding elements of each vector are =.}

\textbf{Numbers} \quad \text{The numbers are \textbf{eql}; that is, they must have the same type and the same value.}

\textbf{Characters} \quad \text{The characters are \textbf{eql}; that is, they must be character objects representing the same character. The code and bits information are taken into account for \textbf{equal}, but font information is not.}

\textbf{Symbols} \quad \text{The symbols are \textbf{eq}; that is, they must be addressing the same memory location.}

\textbf{Arrays} \quad \text{The arrays are \textbf{eq}; that is, they must be addressing the same array in memory.}

\textbf{Pathnames} \quad \text{The pathname objects are equivalent; that is, all of the corresponding components (host, device, directory name, and so on) are the same. The sensitivity of the case of the pathname object is dependent on the file naming conventions of the file system the pathname object resides in.}
For example:

\[
\begin{align*}
\text{(equal 'a 'a)} &= T \\
\text{(equal 'a 'b)} &= \text{NIL} \\
\text{(equal 3.0 3.0)} &= T \\
\text{(equal 3 3.0)} &= \text{NIL} \\
\text{(equal #c(3 -4.0) #c(3 -4))} &= \text{NIL} \\
\text{(equal '(a . b) '(a . b))} &= T \\
\text{(equal (cons 'a 'b) (cons 'a 'c))} &= \text{NIL} \\
\text{(progn (setq x '(a . b)) (equal x x))} &= T \\
\text{(equal #\A #\a)} &= \text{NIL} \\
\text{(equal #\A #\A)} &= T \\
\text{(equal #\c-A #\a)} &= \text{NIL} \\
\text{(equal "Foo" "Foo")} &= T \\
\text{(equal "FOO" "foo")} &= \text{NIL}
\end{align*}
\]

An intuitive definition, which is not quite correct, is that two objects are **equal** if their printed representation is the same. For example:

\[
\begin{align*}
\text{(setq a '(1 2 3))} \\
\text{(setq b '(1 2 3))} \\
\text{(eq a b)} &= \text{NIL} \\
\text{(equal a b)} &= T
\end{align*}
\]

\[
\begin{align*}
\text{(setq a 'a)} &= A \\
\text{(setq b a)} &= A \\
\text{(equal a b)} &= T
\end{align*}
\]

**zl:equal x y**

*Function*

Returns t if its arguments are similar (isomorphic) objects. See the function **eq**. Two numbers are **zl:equal** if they have the same value and type (for example, a flonum is never **zl:equal** to an integer, even if = is true of them). For conses, **zl:equal** is defined recursively as the two **car**s being **zl:equal** and the two **cdr**s being equal. Two strings are **zl:equal** if they have the same length, and the characters composing them are the same. See the function **string-equal**. Alphabetic case is ignored. All other objects are **zl:equal** if and only if they are **eq**. Thus **zl:equal** could have been defined by:

\[
\begin{align*}
\text{(defun zl:equal (x y)} \\
\text{\quad (cond ((eq x y) t) \\
\text{\quad \quad ((neq (typep x) (typep y)) nil) \\
\text{\quad \quad ((numberp x) (= x y)) \\
\text{\quad \quad ((stringp x) (string-equal x y)) \\
\text{\quad \quad ((listp x) (and (equal (car x) (car y)) \\
\text{\quad \quad \quad (equal (cdr x) (cdr y))))))))})}
\end{align*}
\]

As a consequence of the above definition, it can be seen that **zl:equal** may compute forever when applied to looped list structure. In addition, **eq** always implies **zl:equal**; that is, if (eq a b) then (zl:equal a b). An intuitive definition of
\texttt{zl:equal} (which is not quite correct) is that two objects are \texttt{zl:equal} if they look the same when printed out. For example:

\begin{verbatim}
(setq a '(1 2 3))
(setq b '(1 2 3))
(eq a b) => nil
(zl:equal a b) => t
(zl:equal "Foo" "foo") => t
\end{verbatim}

\texttt{si:equal-hash} \texttt{x}

Function

Computes a hash code of an object, and returns it as an integer. A property of \texttt{si:equal-hash} is that \texttt{(equal x y)} always implies \texttt{(#= (si:equal-hash x) (si:equal-hash y))}. The number returned by \texttt{si:equal-hash} is always a nonnegative integer, possibly a large one. \texttt{si:equal-hash} tries to compute its hash code in such a way that common permutations of an object, such as interchanging two elements of a list or changing one character in a string, always changes the hash code.

\texttt{si:equal-hash} uses \texttt{%pointer} to define the hash key for data types such as arrays, stack groups, or closures. This means that some of the hash keys in \texttt{equal} hash tables are based on a virtual memory address. Hash tables that are at all dependent on memory addresses are rehashed when the garbage collector flips.

\texttt{si:equal-hash} returns a second value (\texttt{t}, \texttt{:dynamic} or \texttt{nil}), if it has used \texttt{%pointer} to define the hash key.

\begin{tabular}{ll}
\textbf{Value} & \textbf{meaning} \\
\texttt{nil} & Returned if the hash does not depend on the virtual address of the object being hashed. \\
\texttt{:dynamic} & Returned if the hash depends on the virtual address, but none of the dependent addresses are ephemeral. That is, if \texttt{:dynamic} is returned, future calls to \texttt{si:equal-hash} for the same object might not return the same number if an intervening dynamic GC occurs. \\
\texttt{t} & Returned if the hash depends on the virtual address \texttt{and} at least one of the virtual addresses is ephemeral. That is, if \texttt{t} is returned, future calls to \texttt{si:equal-hash} for the same object might not return the same number if an intervening ephemeral GC occurs. The value \texttt{t} is the strongest and must be preserved when merging more than one result.
\end{tabular}

For example, if \texttt{running-flag} is the merged flag that will eventually be returned, the following form will efficiently do a hash/merge step:

\begin{verbatim}
(multiple-value-bind (hash flag) (si:equal-hash object)
  ;; t is strongest, :dynamic next, do it fast
  (setq running-flag (or (eq flag 't) running-flag flag))
  hash)
\end{verbatim}
Here is an example of how to use `si:equal-hash` in maintaining hash tables of objects:

```lisp
(defun knownp (x &aux i bkt) ; look up x in the table
  (setq i (remainder (si:equal-hash x) 176)) ; The remainder should be reasonably randomized.
  (setq bkt (aref table i)) ; bkt is thus a list of all those expressions that
                           ; hash into the same number as does x.
  (memq x bkt))
```

To write an "intern" for objects, one could:

```lisp
(defun sintern (x &aux bkt i tem)
  (setq i (remainder (si:equal-hash x) 2n-1)) ; 2n-1 stands for a power of 2 minus one.
  ; This is a good choice to randomize the
  ; result of the remainder operation.
  (setq bkt (aref table i))
  (cond ((setq tem (memq x bkt))
        (car tem))
        (t (aset (cons x bkt) table i)
           x)))
```

For a table of related items: See the section "Table Functions".

**si:equal-hash-table**

Creates an old style Zetalisp hash table using the `zl:equal` function for comparison of the hash keys. This flavor is superseeded by `table:basic-table`. It accepts the following init option as well as those described for `eq` hash tables. See the flavor `si:eq-hash-table`.

**:rehash-threshold** Specifies how full the table can be before it must grow. This is typically a flonum. The default is 0.8, which represents 80 percent.

**equal-typep**

Returns `t` if `type1` and `type2` are equivalent and denote the same data type. For the standard type specifiers in Symbolics Common Lisp, see the section "Type Specifier Symbols".

Examples:
(equal-typep 'bit '(unsigned-byte 1)) => T
(equal-typep 'double-float 'long-float) => T
(equal-typep 'bit '(integer 0 1)) => T
(equal-typep 'short-float 'single-float) => T
(equal-typep 'pathname 'complex) => NIL

**equalp x y**

Two objects are **equalp** if they are **equal**. Objects that have components are **equalp** if they are of the same type and corresponding components are **equalp**.

**equalp** differs from **equal** when it compares characters, strings and arrays. **equalp** returns t for character objects when they satisfy **char-equal**. **char-equal** ignores case, as well as font information. For example:

```
(equalp #\A #\a) => T
(equalp #\A #\A) => T
(equalp #\c-A #\A) => NIL
```

**equalp** returns t for arrays when they have the same dimensions, the dimensions match, and the corresponding elements are **equalp**. A string and a general array that happens to contain some characters will be **equalp** even though it is not **equal**. If either argument has a fill pointer, the fill pointer limits the number of elements examined by **equalp**. Because **equalp** performs element-by-element comparisons of strings and ignores the alphabetic case of characters, case distinctions are also ignored when **equalp** compares strings. For example:

```
(setq string "Any Random String") => "Any Random String"
(setq array (make-array 17 :initial-contents "any random string"))
  => #<ART-Q-17 40102625>
(equalp string array) => T
(equalp 3 3.0) => T
(equalp "Abc" "abc") => t
```

**error format-string &rest format-args**

Signals conditions that are not proceedable.

**error** takes three possible argument lists, as follows:

```lisp
(error (format-string &rest format-args))
```

or

```lisp
(error (condition &rest init-options))
```

or

```lisp
(error (condition-object))
```

Case 1:
When `error` is called with `format-string` and `format-args`, under Genera it signals a `zl:ferror` condition. Under CLOE Runtime system, it signals `simple-error` created by the following code:

```
(MAKE-CONDITION 'SIMPLE-ERROR
  :FORMAT-STRING datum
  :FORMAT-ARGUMENTS arguments)
```

`format-string` is given as a control string to `format` along with `format-args` to construct an error message string.

Case 2:

When called with the arguments `condition` and `init-options`, a condition of type `condition` with init options as specified by `init-options` is created and is signalled. `condition` is the name of a condition flavor.

`init-options` are the init options specified when the error object is created; they are passed in the :init message.

Used this way, `error` is similar to `signal` but restricted as follows:

- `error` sets the proceed types of the error object to `nil` so that it cannot be proceeded.

- If no handler exists, the Debugger assumes control, whether or not the object is an error object.

- `error` never returns to its caller.

**Compatibility Note**: The arguments `condition` and `init-options` are Symbolics extensions to Common Lisp.

Case 3:

In the third and more advanced form of `error`, `condition-object` can be a condition object that has been created with `make-condition` but not yet signalled. In this case, `init-options` is ignored.

Note: The argument `condition-object` is a Symbols extension to Common Lisp.

For compatibility with the old Maclisp `error` function, `error` tries to determine that it has been called with Maclisp-style arguments and turns into an `zl:fsignal` or `zl:ferror` as appropriate. If `condition` is a string or a symbol that is not the name of a flavor, and `error` has no more than three arguments, `error` assumes it was called with Maclisp-style arguments.

Note that in CLOE, if `typep` condition `cloe::*break-on-signals*` is true, then the debugger will be entered prior to beginning the signalling process. The signalling process can be continued using the `continue` restart. This is true also for all other functions and macros which signal errors, such as `cerrror`, `assert`, and `check-type`.

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".
### *error-message-hook*

**Variable**

This variable lets you customize the error message printed by the Debugger.

You can bind *error-message-hook* to a one-argument function. Before printing an error message the Debugger checks the value of *error-message-hook*; if this variable is bound to a non-nil value, the Debugger evaluates it and displays the result at the end of the Debugger message.

**Examples:**

```lisp
(defun my-error-hook ()
  (format t "This is the error hook"))
(setq dbg:*error-message-hook* 'dbg:my-error-hook)
```

```lisp
(defun get-plists (list-of-objects)
  (let ((dbg:*error-message-hook*
         (lambda ()
           (format t "While getting properties of ~S" list-of-objects))))
    (symbol-plist list-of-objects))) => GET-PLISTS

(get-plists '(a b c))
```

**Trap:** The argument given to the SYS:PROPERTY-CELL-LOCATION instruction, (A B C), was not a symbol.

**While getting properties of (A B C)**

**SYMBOL-PLIST:**

Arg 0 (SYMBOL): (A B C)

s-A, <RESUME>: Supply replacement argument

s-B: Return a value from the PROPERTY-CELL-LOCATION instruction

s-C: Retry the PROPERTY-CELL-LOCATION instruction

s-D: <ABORT>: Return to Lisp Top Level in Dynamic Lisp Listener 1

→ Resume Proceed

Supply replacement argument

**Form to evaluate and use as replacement argument:**

```lisp
'integer
(ZWEI:ZMACS-BUFFERS (:SAGE-TYPE-SPECIFIER-RECORD #<SECTION-NODE Sage Type
  Specifier Record INTEGER 254116776>))
```

```
```

### *error-output*

**Variable**

The value is a stream to which error messages should be sent. Normally, this is the same as *standard-output*, but *standard-output* might be bound to a file and *error-output* left going to the terminal or a separate file of error messages.
(with-open-stream (outstream "myfile" :direction :output)
  (let ((*standard-output* outstream)
        (*error-output* outstream)) ;redirects *error-output* to myfile.lisp
    (fun-likely-to-signal-an-error));capture any error messages in file
    ;end of let restores *error-output*, etc.
    ... ;more forms
  );end of with-open-file closes file

zl:error-output

Variable

In your new programs, we recommend that you use the variable *error-output* which is the Common Lisp equivalent of zl:error-output. See *error-output*.

error-restart (flavors description &rest args) &body body

Special Form

This form establishes a restart handler for flavors and then evaluates body. If the handler is not invoked, error-restart returns the values produced by the last form in body and the restart handler disappears. When the restart handler is invoked, control is thrown back to the dynamic environment inside the error-restart form and execution of body starts all over again. The format is:

(error-restart (flavors description)
   form-1
   form-2
   ...
)

flavors is either a condition or a list of conditions that can be handled. description is a list of arguments to be passed to format to construct a meaningful description of what would happen if the user were to invoke the handler. args are evaluated when the handler is bound. The Debugger uses these values to create a message explaining the intent of the restart handler.

For a table of related items: See the section "Restart Functions".

eroerror-restart-loop (flavors description &rest args) &body body

Special Form

Establishes a restart handler for flavors and then evaluates the body. If the handler is not invoked, error-restart-loop evaluates the body again and again, in an infinite loop. Use the return function to leave the loop. This mechanism is useful for interactive top levels.

If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment inside the error-restart-loop form and execution of the body is started all over again. The format is:
(error-restart-loop (flavors description)
    form-1
    form-2
    ...)

flavors is either a condition or a list of conditions that can be handled. description is a list of arguments to be passed to format to construct a meaningful description of what would happen if the user were to invoke the handler. The Debugger uses these values to create a message explaining the intent of the restart handler.

For a table of related items: See the section "Restart Functions".

errorp thing

Function

Determines if thing is an error object; returns t if it is, and nil otherwise.

(errorp x) <=> (typep x 'error)

For a table of related items, see the section "Condition-Checking and Signalling Functions and Variables".

etypecase object &body body

Special Form

The name of this function stands for "exhaustive type case" or "error-checking type case". etypecase is similar to typecase, except that it does not allow an explicit otherwise or t clause, and it signals a non-continuable error instead of returning nil if no clause is satisfied.

etypecase is a conditional that chooses one of its clauses by examining the type of an object. Its form is as follows:

(etypecase form
    (types consequent consequent ...)
    (types consequent consequent ...)
    ...
)

First etypecase evaluates form, producing an object. etypecase then examines each clause in sequence. types in each clause is a type specifier in either symbol or list form, or a list of type specifiers. The type specifier is not evaluated. If the ob-
ject is of that type, or of one of those types, then the consequents are evaluated and
the result of the last one is returned (or nil if there are no consequents in
that clause). Otherwise, `etypecase` moves on to the next clause.

If no clause is satisfied, `etypecase` signals an error with a message constructed
from the clauses. It is not permissible to continue from this error. To supply your
own error message, use `typecase` with an `otherwise` clause containing a call to
`error`.

For an object to be of a given type means that if `typep` is applied to the object
and the type, it returns `t`. That is, a type is something meaningful as a second argu-
ment to `typep`.

See the section “Data Types and Type Specifiers”.

It is permissible for more than one clause to specify a given type, particularly if
one is a subtype of another; the earliest applicable clause is chosen. Thus, for
`etypecase`, the order of the clauses can affect the behavior of the construct.

Examples:

```lisp
(defun tell-about-car (x)
  (etypecase (car x)
    (string "string") => TELL-ABOUT-CAR
    (tell-about-car '(("word" "more")) => "string"
    (tell-about-car '(a 1)) => non-proceedable error is signalled
  )
)
```

```lisp
(defun tell-about-car (x)
  (etypecase (car x)
    (fixnum "The car is a number.")
    ((or string symbol) "symbol or string")
    (otherwise "I don’t know.") => TELL-ABOUT-CAR
    (tell-about-car '(1 a)) => "The car is a number."
    (tell-about-car '(a 1)) => "symbol or string"
    (tell-about-car '(("word" "more")) => "symbol or string"
    (tell-about-car '(1.0)) => "I don’t know."
  )
)
```

For a table of related items: See the section “Conditional Functions”.

For a table of related items: See the section “Condition-Checking and Signalling
Functions and Variables”.

**eval form &optional env**

Evaluates `form`, and returns the result. Example:

```lisp
(setq x 43 foo 'bar)
(eval (list 'cons x 'foo))
=> (43 . bar)
```

It is unusual to explicitly call `eval`, since usually evaluation is done implicitly. If
you are writing a simple Lisp program and explicitly calling `eval`, you are probably
doing something wrong. `eval` is primarily useful in programs that deal with Lisp
itself.
Also, if you are only interested in getting at the value of a symbol (that is, the contents of the symbol’s value cell), then you should use the primitive function **symbol-value**.

The actual name of the compiled code for **eval** is "si:*eval" because use of the **evalhook** feature binds the function cell of **eval**.

**Compatibility Note**: The optional argument **env**, which defaults to the null lexical environment, is a Symbolics extension to Common Lisp. You cannot use **Env** in most other implementations of Common Lisp including CLOE Runtime. See the section "Some Functions and Special Forms".

**sys:eval-in-instance instance form**  
*Function*

Evaluates form in the lexical environment of instance. The following form returns the sum of the instance variables x and y of the instance **this-box-with-cell**:

```lisp
(sys:eval-in-instance this-box-with-cell '(+ x y))
```

=> 6

You can use **setq** to modify an instance variable; this is often useful in debugging. If you need to evaluate more than one form in the lexical environment of the instance, you can use **sys:debug-instance**: See the function **sys:debug-instance**.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**eval-when times-list &body forms**  
*Function*

Allows you to tell the compiler exactly when the body forms should be evaluated. times-list can contain one or more of the symbols **load**, **compile**, or **eval**, or can be **nil**.

The interpreter evaluates the body forms only if the times-list contains the symbol **eval**; otherwise **eval-when** has no effect in the interpreter.

**If symbol is present**  
**Then forms are**

**load**  
Written into the compiled code file to be evaluated when the compiled code file is loaded, with the exception that **defun** forms put the compiled definition into the compiled code file.

**compile**  
Evaluated in the compiler.

**eval**  
Ignored by the compiler, but evaluated when read into the interpreter (because **eval-when** is defined as a special form there).

Example 1: Normally, top-level special forms such as **defprop** are evaluated at load time. If some macro expansion depends on the existence of some property, for example, **constant-value**, the definition of that property must be wrapped inside an
(eval-when (compile) \ldots) so that the property is available at compile (macro expansion) time.

   (eval-when (compile load eval)
     (defprop three 3 constant-value))

Example 2: eval-when should be used around defconstants of complex expressions. This is because the compiler does not maintain an environment acceptable to eval containing defconstants

   (eval-when (compile load eval)
     (defconstant name expr))

In other words, if you are sure that (1) evaluating the expr in the global environment gives the correct results, and (2) that no harm is done by changing the current environment to have the (possibly new) value of name, then you can use the global environment as a substitute for the compilation environment.

\textbf{evenp integer} \hfill Function

\textbf{Returns t} if integer is even, otherwise \textbf{nil}. If integer is not an integer, evenp signals an error.

   (evenp 1) => nil
   (evenp 0) => t
   (evenp (* 2 (random n))) => t

See the section "Numeric Property-checking Predicates".

For a table of related items, see the section "Numeric Property-checking Predicates".

\textbf{every predicate sequence &rest more-sequences} \hfill Function

\textbf{Returns nil} as soon as any invocation of predicate returns nil. predicate must take as many arguments as there are sequences provided. predicate is first applied to the elements of the sequences with an index of 0, then with an index of 1, and so on, until a termination criterion is reached or the end of the shortest of the sequences is reached. If the end of a sequence is reached, every returns a non-nil value. Thus considered as a predicate, it is true if every invocation of predicate is true.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:

   (every #'oddp '(1 3 5)) => T

   (every #'equal '(1 2 3) '(3 2 1)) => NIL

   (setq limit-value 1024 sequence (vector 16 64 512 128 32))
If *predicate* has side effects, it can count on being called first on all those elements with an index of 0, then all those with an index of 1, and so on.

For a table of related items: See the section "Predicates that Operate on Lists".

For a table of related items: See the section "Predicates that Operate on Sequences".

**zl:every** *list* *pred* &optional (*step* '#cdr)

Function

Returns *t* if *pred* returns non-nil when applied to every element of *list*, or *nil* if *pred* returns *nil* for some element. If *step-function* is specified, it replaces '#cdr' as the function used to get to the next element of the list; '#cddr' is a typical function to use here. For example:

(zl:every '(1 3 5) #'oddp) => T

(zl:every '(1 2 3 4 5) #'oddp) => NIL

(zl:every '(1 2 3 4 5) #'oddp '#cddr) => T

For a table of related items: See the section "Predicates that Operate on Lists".

For a table of related items: See the section "Predicates that Operate on Sequences".

**exp** *number*

Function

Returns *e* raised to the *number*th power, where *e* is the base of natural logarithms. If *number* is an integer or a single-float, the result is converted to a single-float; if it is a double-float, the result is double-float.

Examples:

(exp 1) => 2.7182817
(exp #c(0 -3)) => #C(-0.9899925 -0.14112002)
(exp 0.88) => 1.083
(exp 2) => 7.389

For a table of related items: See the section "Powers of *e* and Log Functions".

**zl:explode** *x*

Function

Returns a list of characters represented by symbols that are the characters that would be typed out by (prin1 *x*) (that is, the slashified printed representation of *x*).

Example:

(zl:explode '(+ /12 3)) => (|(| + | | /| |1| |2| /| | | |3| |)|)

(Note that there are slashified spaces in the above list.)
**zl:explodec** \(x\)  
*Function*

Returns a list of characters represented by symbols that are the characters that would be typed out by \((\text{princ} \ x)\) (that is, the unslashified printed representation of \(x\)). Example:

\[
(\text{zl:explodec} \ '(+ /12 3)) \Rightarrow (\|(| + | | |1| |2| | | |3| |)|)
\]

**zl:exploden** \(x\)  
*Function*

Returns a list of characters (as integers) that are the characters that would be typed out by \((\text{princ} \ x)\) (that is, the unslashified printed representation of \(x\)). Example:

\[
(\text{zl:exploden} \ '(+ /12 3)) \Rightarrow (#/( #/+ #/Space #/1 #/2 #/Space #/3 #/))
\]

**export** symbols &optional package  
*Function*

The `symbols` argument should be a list of symbols or a single symbol. If `symbols` is `nil`, it is treated like an empty list. These symbols become available as external symbols in `package`. `package` can be a package object or the name of a package (a symbol or a string). If unspecified, `package` defaults to the value of `*package*`. Returns `t`. The `:export` option to `defpackage` and `make-package` is equivalent.

The following bit of code uses `intern` with `multiple-value-bind` to create a new symbol or determine the `status` of an old one. If the `status` of the interned symbol is `:internal`, then the symbols is exported.

\[
=> (\text{multiple-value-bind} \ (\text{symbol} \ \text{status}) \ (\text{intern} \ \text{"new-symbol")})
\]

\[
(\text{when} \ (\text{or} \ (\text{null} \ \text{status}) \ (\text{eq} \ \text{status} \ ':\text{internal})))
\]

\[
(\text{export} \ \text{symbol}))
\]

\[
=> \text{T}
\]

If "new-symbol" is truly a new symbol, then `intern` would have made it an internal symbol. If we now execute the following code on "new-symbol", we will see that it is now an external symbol, since it has been `exported`.

\[
=> (\text{multiple-value-bind} \ (\text{symbol} \ \text{status}) \ (\text{find-symbol} \ \text{"new-symbol")})
\]

\[
\text{status})
\]

\[
=> :\text{EXTERNAL}
\]

**expt base-number power-number**  
*Function*

Computes and returns `base-number` raised to the power `power-number`. If the `base-number` is of type rational and the `power-number` is an integer, the calculation is exact (using the rule of rational canonicalization where applicable), and the result is of type rational; otherwise, a floating-point approximation may result.

If `power-number` is zero of type integer, the result is the value one in the type of `base-number`. This is true even if `base-number` is zero of any type. If `power-number` is a zero of any other data type, the result is the value one, in the type of the ar-
guments after the application of the coercion rules, except as follows. An error results if the base-number is zero and the power-number is a zero not of type integer.

If base-number is negative and power-number is not an integer, the result of expt can be complex, even though neither argument is complex. expt always returns the principal complex value.

Complex canonicalization is applied to complex results.

Examples:

\[
\text{(expt 2 3)} \Rightarrow 8 \\
\text{(expt .5 3)} \Rightarrow 0.125 \\
\text{(expt -49 1/2)} \Rightarrow \#c(0 7) \quad \text{;the principal value} \\
\text{(expt 1/2 -2)} \Rightarrow 4 \\
\text{(expt 2. 0)} \Rightarrow 1 \\
\text{(expt 0 56)} \Rightarrow 0 \\
\text{(expt 0 3/2)} \Rightarrow 0 \\
\text{(expt 0.0 5)} \Rightarrow 0.0 \\
\text{(expt 0.0 \#c(3 4))} \Rightarrow 0.0 \\
\text{(expt \#c(0 7) 2)} \Rightarrow -49
\]

For a table of related items, see the section "Arithmetic Functions".

\text{zl:expt num expt} \quad \text{Function}

Returns \textit{num} raised to the \textit{expt}th power. The result is an integer if both arguments are integers (even if \textit{expt} is negative!) and floating-point if either \textit{num} or \textit{expt} or both is floating-point. If the exponent is an integer a repeated-squaring algorithm is used, while if the exponent is floating the result is (\text{zl:exp (* expt (log num)))}.

\[
\text{(expt 3/5 2)} \Rightarrow 9/25 \\
\text{(expt 4 3)} \Rightarrow 64 \\
\text{(expt (exp 1) 2)} \Rightarrow 7.389
\]

The following functions are synonyms of \text{zl:expt}:

\text{zl:^} \\
\text{zl:^$}

For a table of related items: See the section "Arithmetic Functions" and see CLtL 203.

\text{sys:external-symbol-not-found} \quad \text{Flavor}

A ";" qualified name referenced a name that had not been exported from the specified package.
The :string message returns the name being referenced (no symbol by this name exists yet). The :package message returns the package.

The :export proceed type exports a symbol by that name and uses it.

\textbf{false \&rest ignore} \quad \textbf{Function}

Takes no arguments and returns nil. See the section "Functions and Special Forms for Constant Values".

\textbf{fboundp} \texttt{symbol} \quad \textbf{Function}

Returns t if symbol's function cell contains a function definition, or if symbol names a special form or a macro. Otherwise it returns nil. Since \textbf{fboundp} returns t for special forms and macros, if you want to check for these cases use \textbf{special-form-p} or \textbf{macro-function}.

\begin{verbatim}
(fboundp alarm-handler) => nil
(defun alarm-handler ()
  (setq *alarms* 0))

(fboundp 'alarm-handler) => t
\end{verbatim}

See the section "Functions Relating to the Function Cell of a Symbol".

\textbf{fceiling} \texttt{number \&optional (divisor 1)} \quad \textbf{Function}

Like ceiling, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If number is a floating-point number and divisor is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as number.

Returns the floating point equivalent of the least integer greater than or equal to number; or, in the case of a supplied second argument, returns the floating point equivalent of the least integer greater than or equal to number divided by divisor. A second value, the remainder, is also returned. The remainder returned is the same as that returned by ceiling applied to the same arguments.

Examples:

\begin{verbatim}
(fceiling 5) => 5.0 and 0
(fceiling -5) => -5.0 and 0
(fceiling 5.2) => 6.0 and -0.8000002
(fceiling -5.2) => -5.0 and -0.19999981
(fceiling 5 3) => 2.0 and -1
(fceiling -5 3) => -1.0 and -2
(fceiling 5.2 4) => 2.0 and -2.8000002
\end{verbatim}
(fceiling -5.2 4) => -1.0 and -1.1999998
(fceiling 4.2d0) => 5.0d0 and -0.7999999999999998d0
(fceiling -4.2d0) => -4.0d0 and -0.20000000000000018d0

For a table of related items: See the section "Functions that Divide and Return Quotient as Floating-point Number".

**fdefine** function-spec definition &optional carefully-flag no-query-flag Function

The primitive that **defun** and everything else in the system use to change the definition of a function spec. If **carefully** is non-**nil**, which it usually should be, only the basic definition is changed, the previous basic definition is saved if possible (see **undefun**), and any encapsulations of the function such as tracing and advice are carried over from the old definition to the new definition. **carefully** also causes the user to be queried if the function spec is being redefined by a file different from the one that defined it originally. However, this warning is suppressed if either the argument **no-query** is non-**nil**, or if the global variable **sys:inhibit-fdefine-warnings** is t.

If **fdefine** is called while a file is being loaded, it records what file the function definition came from so that the editor can find the source code.

If **function-spec** was already defined as a function, and **carefully** is non-**nil**, the function-spec's :**previous-definition** property is used to save the previous definition. If the previous definition is an interpreted function, it is also saved on the :**previous-expr-definition** property. These properties are used by the **undefun** function, which restores the previous definition, and the **uncompile** function, which restores the previous interpreted definition. The properties for different kinds of function specs are stored in different places; when a function spec is a symbol its properties are stored on the symbol’s property list.

**defun** and the other function-defining special forms all supply t for **carefully** and **nil** or nothing for **no-query**. Operations that construct encapsulations, such as **trace**, are the only ones that use **nil** for **carefully**.

**sys:fdefine-file-pathname** Variable

While loading a file, this is the generic pathname for the file. The rest of the time it is **nil**. **fdefine** uses this to remember what file defines each function.

**fdefinedp** function-spec Function

This returns t if **function-spec** has a definition, or **nil** if it does not.

**fdefinition** function-spec Function

Returns **function-spec**'s definition. If it has none, an error occurs. You can use **setf** with **fdefinition**.
sys:fdefinition-location function-spec &optional for-compiler | Function

Returns a locative pointing at the cell that contains function-spec's definition. For some kinds of function specs, though not for symbols, this can cause data structure to be created to hold a definition. For example, if function-spec is of the :property kind, then an entry might have to be added to the property list if it isn't already there. In practice, you should write (locf (fdefinition function-spec)) instead of calling this function explicitly.

*features* | Variable

Returns a list of symbols indicating features of the Lisp environment. The default list for Genera is:

```lisp
(:DEFSTORAGE :DEBUG-SCHEDULER-QUEUES :NEW-SCHEDULER :LOOP
 :ROMAN :TRACE :GRINDEF :GRIND)
```

The value of this list is kept up to date as features are added or removed from the Genera system. Most important is the symbol machine-type; this is either 3600 or :imach and indicates on which type of Symbolics machine the program is running. The order of this list should not be depended on, and might not be the same as shown above.

Features SYMBOLICS and CLOE are present in both the CLOE Developer and the CLOE Application Generator. Feature CLOE-DEVELOPER is present only in the CLOE Developer, and feature CLOE-RUNTIME is present only in the Application Generator.

*features* =>

```lisp
 :S SYMBOLICS)
```

zl:ferror format-string &rest format-args | Function

Signals when you do not care what the condition is. zl:ferror signals the condition zl:ferror. (See the flavor zl:ferror.) The arguments are passed as the :format-string and :format-args init keywords to the error object.

The old (zl:ferror nil ...) syntax continues to be accepted for compatibility reasons indefinitely; the nil is ignored. An error is signalled if the first argument is a symbol other than nil; the first argument must be nil or a string.

Note: zl:ferror is an obsolete function. Use error instead in your new programs.

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

ffloor number &optional (divisor 1) | Function
Like \texttt{floor}, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If \textit{number} is a floating-point number and \textit{divisor} is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as \textit{number}.

Examples:

\begin{verbatim}
(ffloor 5) => 5.0 and 0
(ffloor -5) => -5.0 and 0
(ffloor 5.2) => 5.0 and 0.19999981
(ffloor -5.2) => -6.0 and 0.8000002
(ffloor 5 3) => 1.0 and 2
(ffloor -5 3) => -2.0 and 1
(ffloor 5.2 4) => 1.0 and 1.1999998
(ffloor -5.2 4) => -2.0 and 2.8000002
(ffloor 4.2d0) => 4.0d0 and 0.20000000000000018d0
(ffloor -4.2d0) => -5.0d0 and 0.7999999999999998d0
\end{verbatim}

For a table of related items: See the section "Functions that Divide and Return Quotient as Floating-point Number".

\textbf{fifth list} \hspace{2cm} \textit{Function}

Returns the fifth element of the list \textit{list}. \textbf{fifth} is equivalent to:

\begin{verbatim}
(nth 4 list)
\end{verbatim}

For example:

\begin{verbatim}
(setq letters '(a b c d e f g i j)) =>
(A B C D E F G I J)

(fifth letters) => E
\end{verbatim}

For a table of related items: See the section "Functions for Extracting from Lists".

\textbf{file-position stream \&optional position} \hspace{2cm} \textit{Function}

Returns or sets the current position in a random-access file. When only \textit{stream} is specified, returns a non-negative integer that indicates the current position within \textit{stream}, or \texttt{nil} if this cannot be determined. (The file position at the start of a file is zero.) Ordinarily, the value returned by \textbf{file-position} increases by one each time an input or output operation is performed; however, performing a single \textbf{read-char} or \textbf{write-char} operation on a character file might increment the file position by more than one because of character-set translations. For a binary file, each \textbf{read-byte} or \textbf{write-byte} operation increases the file position by one.

\textit{position} sets the position in \textit{stream} to \textit{position}. \textit{position} can be an integer, \texttt{:start} for the beginning of the stream, or \texttt{:end} for the end of the stream. An error is signalled if the integer is too large for the file. (An integer returned by \textbf{(file-position stream)} should be usable as a value of \textit{position}.) When \textit{position} is specified, \textbf{file-position} returns \texttt{t} if the repositioning was successful, \texttt{nil} if it was not.
Function

Destructively modifies sequence by replacing each element of the subsequence specified by the :start (which defaults to zero) and :end (which defaults to the length of the sequence) arguments with item.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

item can be any be any Lisp object, but must be a suitable element for sequence. Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence, up to but not including the one specified by the :end index (defaults to length of sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

```
(setq a-vector (vector 'a 'b 'c 'd 'e)) => #(A B C D E)

(fill a-vector 'z :start 1 :end 3) => #(A Z Z D E)

a-vector => #(A Z Z D E)

(fill a-vector 'rah) => #(RAH RAH RAH RAH RAH)

a-vector => #(RAH RAH RAH RAH RAH)
```

For a table of related items: See the section "Sequence Modification".

Function

The opposite of math:list-2d-array. list should be a list of lists, with each element being a list corresponding to a row. array's elements are stored from the list. Unlike zl:fillarray, if list is not long enough, math:fill-2d-array "wraps around", starting over at the beginning. The lists that are elements of list also work this way.
**fill-pointer** array

Returns the value of the fill pointer. array must have a fill pointer. `setf` can be used on a fill-pointer form to set the value of the fill pointer.

Under CLOE, if the new value of fill pointer in a `setf` command is greater than the array-total-size, a continuable error signals.

Some other functions, notably `vector-push` and `vector-pop`, alter the value of the fill pointer. The value of the fill pointer can be set at the time the array is created by specifying a non-negative integer as the value of the keyword argument :fill-pointer.

```lisp
(setq astring (make-array 12 :element-type 'string-char :fill-pointer 0))

(fill-pointer astring) => 0
(vector-push #\a astring) => 0
astring => "a"
(fill-pointer astring) => 1

(setf (fill-pointer astring) 0)
astring => ""
(aref astring 0) => #\a

(vector-push #\b astring) => 0
astring => "b"
(aref astring 0) => #\b
(fill-pointer astring) => 1
```

**zl:fillarray** array source

Fills up array with the elements of source. array can be any type of array or a symbol whose function cell contains an array. Two forms of this function exist, depending on whether the type of source is a list or an array.

If source is a list, then `zl:fillarray` fills up array with the elements of list. If source is too short to fill up all of array, then the last element of source is used to fill the remaining elements of array. If source is too long, the extra elements are ignored. If source is nil (the empty list), array is filled with the default initial value for its array type (nil or 0).

If source is an array (or a symbol whose function cell contains an array), the elements of array are filled up from the elements of source. If source is too small, then the extra elements of array are not affected. `zl:fillarray` returns array.

If array is multidimensional, the elements are accessed in row-major order: the last subscript varies the most quickly. The same is true of source if it is an array.

`:filled-elements`
Returns the number of entries in the hash table that have an associated value. This message is obsolete; use `hash-table-count` instead.

**finally keyword for loop**

**finally expression**

Puts `expression` into the *epilogue* of the loop, which is evaluated when the iteration terminates (other than by an explicit `return`). For stylistic reasons, then, this clause should appear last in the loop body. Note that certain clauses can generate code that terminates the iteration without running the epilogue code; this behavior is noted with those clauses. See the section "Aggregated Boolean Tests for loop". This clause can be used to cause the loop to return values in a nonstandard way:

```
(loop for n in l ; l is a list
   sum n into the-sum
   count t into the-count
   finally (return (quotient the-sum the-count)))
```

```
(defun sum-series (limit)
  (loop for num from 0 to limit
     with sum-of-series = 0
     initially (print "The sum of this series is ":)
    do
       (setq sum-of-series (+ sum-of-series num))
   finally (prin1 sum-of-series))) => SUM-SERIES

(sum-series 9)  =>
"The sum of this series is ": 45
NIL

(defun over-the-top (num)
  (loop for i from 1 to 10
     when (= i num) return i
     finally (print "Finally triggered"))) => OVER-THE-TOP

(over-the-top 5) => 5
(over-the-top 20) =>
"Finally triggered" NIL
```

See the macro `loop`.

**find item sequence &key (test #eql) :test-not (:key #identity) :from-end (:start 0) :end**

*Function*

If `sequence` contains an element satisfying the predicate specified by the `:test` keyword argument, returns the leftmost, otherwise returns `nil`.
*item* is matched against the elements specified by the *test* keyword. The *item* can be any Symbolics Common Lisp object.

*sequence* can be either a list or a vector (one-dimensional array). Note that *nil* is considered to be a sequence, of length zero.

**:test** specifies the test to be performed. An element of *sequence* satisfies the test if (funcall *testfun* *item* (*keyfn* *x*)) is true. Where *testfun* is the test function specified by **test**, *keyfn* is the function specified by **key** and *x* is an element of the sequence. The default test is eql.

**:test-not** is similar to **test**, except that the sense of the test is inverted. An element of *sequence* satisfies the test if (funcall *testfun* *item* (*keyfn* *x*)) is false.

For example:

```
(find 'a '(a b c d) :test-not #'eql) => B
```

The value of the keyword argument **key**, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```
(find 'a '((a b) (a d) (b c)) :key #'car) => (A B)
(find 'a #((a b) (a d) (b a)) :key #'cadr) => (B A)
```

If the value of the **from-end** keyword is non-nil, the result is the rightmost element satisfying the test.

For example:

```
(find 3 '((right 3) (west 2) (south 3)) :key #'cadr :from-end t) => (SOUTH 3)
```

You can delimit the portion of the sequence to be operated on by the keyword arguments **start** and **end**.

**:start** and **end** must be non-negative integer indices into the sequence. **start** must be less than or equal to **end**, else an error is signalled. It defaults to zero (the start of the sequence).

**:start** indicates the start position for the operation within the sequence. **end** indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both **start** and **end** are omitted, the entire sequence is processed by default.

For example:

```
(find 'A '(b c a)) => A
(find 'a '(a b b) :start 1 :end 3) => NIL
(find 'a '(a b b) :start 0 :end 3) => A
(find 1 #(2 3 4 1) :end 4) => 1
(find 1 #(2 3 4 1) :end 3) => NIL
```

For a table of related items: See the section "Searching for Sequence Items".
find-all-symbols string

Function

Searches all packages for symbols named string and returns a list of them. Duplicates are removed from the list; if a symbol is present in more than one package, it only appears once in the list. The global package is searched first, and so global symbols appear earlier in the list than symbols that shadow them. In general packages are searched in the order that they were created.

string can be a symbol, in which case its name is used. This is primarily for user convenience when calling find-all-symbols directly from the read-eval-print loop.

Under Genera, invisible packages are not searched.

The where-is function under Genera is a more user-oriented version of find-all-symbols; it returns information about string, rather than just a list. For example:

```
0 => (make-symbol 'foo)
#:FOO
=> (make-symbol 'foo)
#:FOO
=> (setq x (make-symbol 'foo))
#:FOO
=> (setq foo-list (find-all-symbols x)
#:FOO #:FOO #:FOO)
=> (list-length foo-list)
3
```

Note that find-all-symbols is not in CLOE Runtime.

For more information: See the section "Mapping Names to Symbols".

(flavor:method :find-by-item si:heap) item &optional (equal-predicate #'=) Method

Finds the first item that satisfies equal-predicate and returns the item and key if it was found; otherwise it signals si:heap-item-not-found. equal-predicate should be a function that takes two arguments. The first argument to equal-predicate is the current item from the heap and the second argument is item.

For a table of related items: See the section "Heap Functions and Methods".

(flavor:method :find-by-key si:heap) key &optional (equal-predicate #'=) Method

Finds the first item whose key satisfies equal-predicate and returns the item and key if it was found; otherwise it signals si:heap-item-not-found. equal-predicate should be a function that takes two arguments. The first argument to equal-predicate is the current key from the heap and the second argument is key.

For a table of related items: See the section "Heap Functions and Methods".

clos:find-class class-name &optional errorp environment Function
Returns the class object named by class-name in the given environment. You can use setf with clos:find-class to change the class associated with the symbol class-name.

class-name A symbol to be the name of the class, or nil to remove the association between a class name and a symbol.

tfp A boolean value indicating what to do if there is no class object named class-name. A value of t causes an error to be signaled; this is the default. A value of nil causes nil to be returned.

environment The same as the &environment argument to macro expansion functions. It is typically used to distinguish between compile-time and run-time environments.

flavor:find-flavor flavor-name &optional (error-p t)

Determines whether a flavor is defined in the world. Returns non-nil if the flavor is defined.

If the flavor is not defined and error-p is non-nil (or not supplied), flavor:find-flavor returns nil. However, if the flavor is not defined and error-p is nil, flavor:find-flavor signals an error.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

clos:find-method generic-function method-qualifiers specializers &optional errorp

Returns the method object that is identified by generic-function, method-qualifiers, and specializers.

generic-function A generic function object.

method-qualifiers A list of the method's qualifiers. The order of method-qualifiers is significant.

specializers A list of the method's parameter specializers. This list must contain an element for each required argument to the generic function or else an error is signaled. The parameter specializer for any unspecialized parameter is the class named t.

Note that CLOS distinguishes between a parameter specializer name (these appear in the clos:defmethod lambda-list) and the corresponding parameter specializer object. The specializers argument consists of parameter specializer objects. There are two cases: the parameter specializer name is either a class name or a list such as (eql form). When the parameter specializer name is a class name, the corresponding object is the class object of
that name. When the parameter specializer name is a list such as (eql form), the corresponding object is the list (eql object), where object is the result of evaluating form.

errorp A boolean value indicating what to do if there is no method. A value of t causes an error to be signaled; this is the default. A value of nil causes nil to be returned.

find-if predicate sequence &key :key :from-end (:start 0) :end

Function

If sequence contains an element satisfying predicate, the leftmost such element is returned; otherwise nil is returned.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```lisp
(find-if #'atom '((a (b)) ((a) b) (nil nil)) :key #'second)
=> ((A) B)
```

If the value of the :from-end keyword is non-nil, the result is the rightmost element satisfying the test.

For example:

```lisp
(find-if #'numberp '(1 1 2 2) :from-end t)  => 2
(find-if #'numberp '(1 1 2 2) :from-end nil) => 1
```

You can delimit the portion of the sequence to be operated on by the keyword arguments :start and :end.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

```lisp
(find-if #'oddp '(1 2 1 2))  => 1
(find-if #'oddp '(1 1 2 2) :start 3 :end 4) => NIL
```

For a table of related items: See the section "Searching for Sequence Items".
find-if-not  predicate  sequence  &key  :key  :from-end  (start  0)  :end  

Function

If sequence contains an element that does not satisfy predicate, the leftmost such element is returned; otherwise nil is returned.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

(find-if-not #'atom '((a (b)) ((a) b) (nil nil)) :key #'second)  
=> (A (B))

If the value of the :from-end keyword is non-nil, the result is the rightmost element satisfying the test.

For example:

(find-if-not #'evenp '(3 2 1) :from-end t)  => 1
(find-if-not #'evenp '(3 2 1) :from-end nil)  => 3

For the sake of efficiency, you can delimit the portion of the sequence to be operated on by the keyword arguments :start and :end.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(find-if-not #'oddp '(3 5 4 3 5))  => 4
(find-if-not #'oddp '(3 5 4 3 5) :start 3 :end 4)  => NIL
(find-if-not #'evenp '(3 5 4 3 5) :start 3 :end 4)  => 3
(find-if-not #'oddp a :start 1 :key #'car)  => (4 3)
(setq text "It was the height, of folly; Was it not?"

(find-if-not #'(lambda(x)(or (alpha-char-p x)(char= x #\Space))) text)  
=> #\,

For a table of related items: See the section "Searching for Sequence Items".
**find-package name**

Returns the package whose string name is `name` or the print name of `name`, if `name` is a symbol. Case is considered, and if no matching package exists, `nil` is returned. This allows you to locate the actual package object for use with those functions that take a package (not the name of the package) as an argument, such as `package-name` and `package-nicknames`.

```lisp
(find-package 'common-lisp-user) =>
  #<Package USER (really COMMON-LISP-USER) 71733245>
(package-nicknames *) => ("CL-USER")
```

In the following example, the current package is set to the package named `turbine-controller` if there is such a package. If no such package exists, a file which presumably contains its definition is loaded, and then the current package is set to that package.

```lisp
(setq *package*
  (or (find-package 'turbine-controller)
      (progn (load "turbcont.lsp")
             (find-package 'turbine-controller))
      (error "Couldn't find package TURBINE-CONTROLLER.")
))
```

For more information, see the section "Mapping Between Names and Packages".

**zl:find-position-in-list item list**

Looks down `list` for an element that is `eq` to `item`, like `zl:memq`. However, it returns the position (numeric index) in the list at which it found the first occurrence of `item`, or `nil` if it did not find it at all. This function is sort of the complement of `nth`; like `nth`, it is zero-based. See the function `nth`. Examples:

```lisp
(zl:find-position-in-list 'a '(a b c)) => 0
(zl:find-position-in-list 'c '(a b c)) => 2
(zl:find-position-in-list 'e '(a b c)) => nil
```

For a table of related items: See the section "Functions for Finding Information About Lists and Conses".

**zl:find-position-in-list-equal item list**

Looks down `list` for an element that is `eql` to `item`. However, it returns the position (numeric index) in the list at which it found the first occurrence of `item`, or `nil` if it did not find it at all. This function is sort of the complement of `nth`; like `nth`, it is zero-based.

For a table of related items: See the section "Functions for Finding Information About Lists and Conses".

**find-symbol string &optional (pkg *package*)**

Function
Searches *pkg* for the symbol *string*. It behaves like *intern* except that it never creates a new symbol. If it finds a symbol named *string*, it returns that symbol as its first value. The second value is one of the following:

*:internal*  The symbol is present in *pkg* as an internal symbol.

*:external*  The symbol is present in *pkg* as an external symbol.

*:inherited*  The symbol is an internal symbol in *pkg* inherited by way of *use-package*.

If it is unable to find a symbol named *string* in the specified packages, it returns nil nil.

In the following example, *find-symbol* is used to determine the status of a prospective internal symbol. If a symbol with the specified print name already exists, it is *uninterned* unless it is inherited from another package. A new symbol with the specified print name is then *interned*.

```lisp
(multiple-value-bind (symbol status) (find-symbol new-symbol)
  (if symbol
      (unless (eq status ':inherited)
        (unintern symbol)
        (intern new-symbol))
      (intern new-symbol)))
```

**:finish**  

Message

Does a :force-output to a buffered asynchronous device, such as the Chaosnet, then waits until the currently pending I/O operation has been completed. If the stream does not handle this, the default handler ignores it.

For file output streams, :finish finalizes file content. It ensures that all data have actually been written to the file, and sets the byte count. It converts non-direct output openings into append openings. It allows other users to access the data that have been written before the :finish message was sent.

**finish-output**  

&optional  *output-stream*

Function

Some streams are implemented in an asynchronous, or buffered, manner. *finish-output* attempts to ensure that all output sent to *output-stream* has reached its destination, and only then returns nil. *Output-stream* if unspecified or nil, defaults to *standard-output*, and if t, is *terminal-io*.

**first**  *list*

Function

Returns the first element of the list *list*. *first* is equivalent to *car*. This function is provided because it makes more sense when you are thinking of the argument as a list rather than just as a cons. For example:
(setq letters '(a b c d)) => (A B C D)

(first letters) => A

For a table of related items: See the section "Functions for Extracting from Lists".

**zl:firstn n list**

*Function*

Returns the list of length n, whose elements are the first n elements of list. If list is fewer than n elements long, zl:firstn fills out the list with elements of the value nil. Example:

(zl:firstn 2 '(a b c d)) => (a b)
(zl:firstn 0 '(a b c d)) => nil
(zl:firstn 6 '(a b c d)) => (a b c d nil nil)

For a table of related items: See the section "Functions for Extracting from Lists".

**zl:fix number**

*Function*

Converts number from a floating-point or rational number to an integer, truncating towards negative infinity. If number is already an integer, it is returned unchanged.

zl:fix is similar to floor, except that it returns only the first value of floor.

See the section "Functions that Divide and Convert Quotient to Integer".

For a table of related items: See the section "Functions that Divide and Convert Quotient to Integer".

**fixnum**

*Type Specifier*

fixnum is the type specifier symbol for the predefined primitive Lisp object of that name.

The types fixnum and bignum are an exhaustive partition of the type integer, since integer = (or bignum fixnum). These are internal representations of integers used by the system for efficiency depending on integer size; in general, fixnums and bignums are transparent to the programmer.

Examples:

(typep 4 'fixnum) => T
(zl:typep '1 ) => :FIXNUM
(subtypep 'fixnum 'number) => T and T ; subtype and certain
(commonp most-positive-fixnum) => T
(zl:fixnump 90) => T
(type-of 8654) => FIXNUM

See the section "Data Types and Type Specifiers". See the section "Numbers".
sys:fixnump object

Function

Returns t if its argument is a fixnum, otherwise nil.

For a table of related items, see the section "Numeric Type-checking Predicates".

zl:fixp object

Function

In your new programs, we recommend that you use the function integerp which is the Common Lisp equivalent of the function zl:fixp.

zl:fixp returns t if its argument is an integer, otherwise nil.

For a table of related items, see the section "Numeric Type-checking Predicates".

zl:fixr x

Function

Converts x from a floating-point number to an integer, rounding to the nearest integer. zl:fixr is similar to round, except when x is exactly halfway between two integers. In this case, zl:fixr rounds up (towards positive infinity), while round rounds to an even integer.

zl:fixr could have been defined by:

\[
(\text{defun zl:fixr } (x)
  (\text{if (zl:fixp x) } x \text{ (zl:fix (+ x 0.5))})
\]

For a table of related items: See the section "Functions that Divide and Convert Quotient to Integer".

sys:flatc x

Function

Returns the number of characters in the unslashified printed representation of x.

Example:

\[
(\text{flatsize '(+ /12 3)}) \Rightarrow 10
\]

sys:flatsize x

Function

Returns the number of characters in the slashified printed representation of x.

Example:

\[
(\text{flatsize '(+ /12 3)}) \Rightarrow 12
\]

flavor:flavor-allowed-init-keywords flavor-name

Function

Returns an alphabetically sorted list of all symbols that are valid init options for the flavor named flavor-name. Valid init options are allowed keyword arguments to make-instance.
This function is primarily useful for people, rather than programs, to call to get information. You can use this to help remember the name of an init option or to help write documentation about a particular flavor.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**flavor-allows-init-keyword-p**  
*flavor*-name *keyword*  
**Function**

Returns non-nil if the *keyword* is a valid init option for the flavor named *flavor-name*, or nil if it does not. Valid init options are allowed keyword arguments to `make-instance`. The non-nil value is the name of the component flavor that contributes the support of that keyword.

This function is primarily useful for people, rather than programs, to call to get information.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**flavor:*flavor-compile-trace-list***  
**Variable**

Value is a list of structures, each of which describes the compilation of a combined method into the run-time (not the compile-time) environment, in newest-first order. The function `flavor:print-flavor-compile-trace` lets you selectively access the information saved in this variable. See the function `flavor:print-flavor-compile-trace`.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**flavor:*flavor-default-init-get***  
*flavor* *property*  
**Function**

Similar to `get`, except that its first argument is either a flavor structure or the name of a flavor. It retrieves the property from the default init-plist of the specified flavor. You can use `setf` with it:

```lisp
(setf (flavor:flavor-default-init-get f p) x)
```

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**flavor:*flavor-default-init-putprop***  
*flavor* *value* *property*  
**Function**

Like `zl:putprop`, except that its first argument is either a flavor structure or the name of a flavor. It puts the property on the default-init-plist of the specified flavor.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".
flavor:flavor-default-init-remprop flavor property

Function

Similar to remprop, except that its first argument is either a flavor structure or the name of a flavor. It removes the property from the default init-plist of the specified flavor.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

flet functions &body body

Special Form

Defines named internal functions. flet (function let) defines a lexical scope, body, in which these names can be used to refer to these functions. functions is a list of clauses, each of which defines one function. Each clause of the flet is identical to the cdr of a defun special form; it is a function name to be defined, followed by an argument list, possibly declarations, and function body forms. flet is a mechanism for defining internal subroutines whose names are known only within some local scope.

Functions defined by the clauses of a single flet are defined "in parallel", similar to let. The names of the functions being defined are not defined and not accessible from the bodies of the functions being defined. The labels special form is used to meet those requirements. See the special form labels.

Here is an example of the use of flet:

(defun triangle-perimeter (p1 p2 p3)
  (flet ((squared (x) (* x x)))
    (flet ((distance (point1 point2)
              (sqrt (+ (squared (- (point-x point1)
                               (point-x point2)))))
              (squared (- (point-y point1)
                            (point-y point2))))))
      (+ (distance p1 p2)
          (distance p2 p3)
          (distance p1 p3))))

flet is used twice here, first to define a subroutine squared of triangle-perimeter, and then to define another subroutine, distance. Note that since distance is defined within the scope of the first flet, it can use squared. distance is then called three times in the body of the second flet. The names squared and distance are not meaningful as function names except within the bodies of these flets.

Note that functions defined by flet are internal, lexical functions of their containing environment. They have the same properties with respect to lexical scoping and references as internal lambdas. They can make free lexical references to variables of that environment and they can be passed as funargs to other procedures. Functions defined by flet, when passed as funargs, generate closures. The allocation of these closures, that is, whether they appear on the stack or in the heap, is controlled in the same way as for internal lambdas. See the section "Funargs and Lexical Closure Allocation".
Here is an example of the use, as a funarg, of a closure of a function defined by \texttt{flet}.

\begin{verbatim}
(defun sort-by-closeness-to-goal (list goal)
  (flet ((closer-to-goal (x y)
           (< (abs (- x goal)) (abs (- y goal))))
         (sort list #'closer-to-goal)))

This function sorts a list, where the sort predicate of the (numeric) elements of the list is their absolute distance from the value of the parameter \texttt{goal}. That predicate is defined locally by \texttt{flet}, and passed to \texttt{sort} as a funarg.

Note that \texttt{flet} (as well as \texttt{labels}) defines the use of a name as a function, not as a variable. Function values are accessed by using a name as the car of a form or by use of the \texttt{function} special form (usually expressed by the reader macro \('#').

Within its lexical scope, \texttt{flet} can be used to redefine names that refer to globally defined functions, such as \texttt{sort} or \texttt{cdar}, though this is not recommended for stylistic reasons. This feature does, however, allow you to bind names with \texttt{flet} in an unrestricted fashion, without binding the name of some other function that you might not know about (such as \texttt{number-into-array}), and thereby causing other functions to malfunction. This occurs because \texttt{flet} always creates a lexical binding, not a dynamic binding. Contrast this with \texttt{let}, which usually creates a lexical binding, unless the variable being bound is declared special, in which case it creates a dynamic binding.

\texttt{flet} can also be used to redefine function names defined by enclosing uses of \texttt{flet} or \texttt{labels}.

In the following example, \texttt{eql} is redefined to a more liberal treatment for characters. Note that the global definition of \texttt{eql} is used in the local definition (this would not be possible with \texttt{labels}). Note also that \texttt{member} uses the global definition of \texttt{eql}.

\begin{verbatim}
(flet ((eql (x y)
         (if (characterp x)
             (equalp x y)
             (eql x y))))
    (if (member foo bar-list)              ;uses global eql
        (adjoin 'baz bar-list :test #'eql)   ;uses flet'd eql
        (eql foo (car bar-list))))
\end{verbatim}

\texttt{float} \&optional \texttt{(low '*)} \texttt{(high '*)} Type Specifier

\texttt{float} is the type specifier symbol for the predefined Lisp floating-point number type.

The types \texttt{float}, \texttt{rational}, and \texttt{complex} are \textit{pairwise disjoint subtypes} of \texttt{number}.

The \texttt{float} data type is a \textit{supertype} of the types:
short-float
single-float
long-float
double-float

This type specifier can be used in either symbol or list form. Used in list form, float allows the declaration and creation of specialized floating-point numbers, whose range is restricted to low and high.

low and high must each be a floating-point number, a list of floating-point number, or unspecified; in floating-point number form the limits are inclusive; in list form they are exclusive, and * means that a limit does not exist and so effectively denotes minus or plus infinity, respectively.

Examples:

(typep 20.4e-2 'float) => T
(typep (/ (float 14) (float 4)) 'float) => T
;note the use of float the function and float the type
(subtypep 'float 'number) => T and T ;subtype and certain
(subtypep 'single-float 'float) => T and T
(commonp (float 3)) => T
(floatp 989.e-3) => T

See the section "Data Types and Type Specifiers".
See the section "Numbers".

float number &optional other

Function

Converts any noncomplex number to a floating-point number. With no second argument, if number is already a floating-point, number is returned. If number is not of floating-point type, a single-float is produced and returned.

If the second argument other is provided, it must be of floating-point type, and number is converted to the same format as other.

Examples:

(float 3) => 3.0
(float 3 1.0d0) => 3.0d0

For a table of related items, see the section "Functions that Convert Numbers to Floating-point Numbers".

zl:float x

Function

Converts any noncomplex number to a single-precision floating-point number. Note that zl:float reduces a double-precision argument to single precision.

Examples:

(zl:float 3) => 3.0
(zl:float 6.02d23) => 6.02e23
See the section "Functions that Convert Numbers to Floating-point Numbers".

For a table of related items: See the section "Functions that Convert Numbers to Floating-point Numbers".

**float-digits float**  
*Function*

Returns, as a non-negative integer, the number of binary digits used in the binary representation of its floating-point argument (including the implicit "hidden bit" used in IEEE standard floating-point representation).

Genera examples:

- `(float-digits 0.0) => 24`
- `(float-digits 3.0s5) => 24`
- `(float-digits pi) => 53` ;pi is a long float when using Genera
- `(float-digits 1.0s-40) => 24`

In CLOE, returns a non-negative integer that provides the number of digits in the radix of `float` (two in CLOE implementations) used to represent `float`. For normalized floats, this function will produce the same result as `float-precision`.

- `(float-digits 5.06s2) => 22`

For a table of related items, see the section "Functions that Decompose and Construct Floating-point Numbers".

**float-precision float**  
*Function*

Returns, as a non-negative integer, the number of significant binary digits present in the binary representation of the floating-point argument. Note that if the argument is (a floating-point) zero, the result is an (integer) zero. For normalized floating-point numbers, `float-digits` and `float-precision` return identical results. For a denormalized or zero number, the precision is smaller than the number of representation digits (that is, `float-precision` returns a smaller number).

Examples:

- `(float-precision 0.0) => 0`
- `(float-precision 1.6s-19) => 24`
- `(float-precision 1.61-19) => 53`
- `(float-precision 1.0s-40) => 17`

Under CLOE, returns a non-negative integer that provides the number of significant digits in the radix of `float` used in the representation of `float`. For floating point zeroes, this function returns zero. For normalized floats, this function produces the same result as `float-digits`.

- `(float-precision 4.5) => 22`

For a table of related items, see the section "Functions that Decompose and Construct Floating-point Numbers".
\textbf{float-radix} \textit{float} \hspace{1cm} \textit{Function}

Returns the integer 2 denoting the radix of the internal IEEE floating-point representation in Symbolics Common Lisp under Genera.

In CLOE implementations, \textbf{float-radix} returns the constant 2, but Common Lisp permits implementations to have an alternate float radix, or even different radices for different floats.

Examples:

\begin{verbatim}
(float-radix pi) => 2
(float-radix 5.0l0) => 2
\end{verbatim}

For a table of related items, see the section "Functions that Decompose and Construct Floating-point Numbers".

\textbf{float-sign} \textit{float1} \&optional \textit{float2} \hspace{1cm} \textit{Function}

Returns a floating-point number, \( z \), which has the same sign as \textit{float1} and the same absolute value and format as \textit{float2}. The second argument defaults to the value of \((\textit{float} \ \textit{float1})\), that is, it is a floating-point 1 of the same type as \textit{float1}. Both arguments must be floating-point numbers.

Examples:

\begin{verbatim}
(float-sign 3.0) => 1.0
(float-sign -7.9) => -1.0
(float-sign -2.0 pi) => -3.141592653589793d0
\end{verbatim}

For a table of related items, see the section "Functions that Decompose and Construct Floating-point Numbers".

\textbf{floatp} \textit{object} \hspace{1cm} \textit{Function}

Returns \texttt{t} if its argument is a (single- or double-precision) floating-point number. Otherwise it returns \texttt{nil}. The following code tests whether \texttt{a} and \texttt{b} are numbers. If numbers, they are added. Otherwise, we attempt to extract floats that are then tested by \textbf{floatp}.

\begin{verbatim}
(if (and (numberp a) (numberp b))
  (+ a b)
  (if (and (consp a)
           (floatp (car a))
           (consp b)
           (floatp (car b)))
       (+ (car a) (car b))
       (error "couldn't extract floats from ~a and ~a" a b)))
\end{verbatim}

For a table of related items, see the section "Numeric Type-checking Predicates".

\textbf{floatp} \textit{object} \hspace{1cm} \textit{Function}
Returns \texttt{t} if its argument is a (single- or double-precision) floating-point number. Otherwise it returns \texttt{nil}. The following code tests whether \texttt{a} and \texttt{b} are numbers. If numbers, they are added. Otherwise, we attempt to extract floats that are then tested by \texttt{floatp}.

\begin{verbatim}
(if (and (numberp a) (numberp b))
  (+ a b)
  (if (and (consp a)
  (floatp (car a)))
     (consp b)
    (floatp (car b)))
  (+ (car a) (car b))
  (error "couldn't extract floats from ~a and ~a" a b)))
\end{verbatim}

For a table of related items, see the section "Numeric Type-checking Predicates".

\textbf{zl:flonump \textit{object}}

\textbf{Function}

Returns \texttt{t} if \textit{object} is a single-precision floating-point number, otherwise it returns \texttt{nil}.

The following function is a synonym of \texttt{zl:flonump}:

\textbf{sys:single-float-p}

For a table of related items, see the section "Numeric Type-checking Predicates".

\textbf{floor \textit{number} \&optional (\textit{divisor} 1)}

\textbf{Function}

Divides \textit{number} by \textit{divisor}, and truncates the result toward negative infinity. The truncated result and the remainder are the returned values.

\textit{number} and \textit{divisor} must each be a noncomplex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are \texttt{Q} and \texttt{R}, then \((+ (* \texttt{Q} \texttt{divisor}) \texttt{R})\) equals \texttt{number}. If \texttt{divisor} is 1, then \texttt{Q} and \texttt{R} add up to \texttt{number}. If \texttt{divisor} is 1 and \texttt{number} is an integer, then the returned values are \texttt{number} and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, then the second returned value is always a number of the same type as the argument.

\textbf{Examples:}

\begin{verbatim}
(floor 5) => 5 and 0
(floor -5) => -5 and 0
(floor 5.2) => 5 and 0.19999981
(floor -5.2) => -6 and 0.8000002
(floor 5.8) => 5 and 0.8000002
\end{verbatim}
Using `floor` with one argument is the same as the `zl:fix` function, except that `zl:fix` returns only the first value of `floor`.

See the section "Comparison of `floor`, `ceiling`, `truncate` and `round".

For a table of related items: See the section "Functions that Divide and Convert Quotient to Integer".

`fmakunbound` symbol

Causes `symbol` to be undefined, that is, its function cell to be empty. It returns `symbol`.

Because `symbol` no longer has a function definition, function invocation results in an error after applying `fmakunbound`, unless later redefined.

```
(fboundp 'alarm-handler) => nil

(defun alarm-handler ()
  (setq *alarms* 0))

(fboundp 'alarm-handler) => t

(fmakunbound 'alarm-handler)

(fboundp 'alarm-handler) => nil
```

See the section "Functions Relating to the Function Cell of a Symbol".

`future-common-lisp:fmakunbound` function-name

Removes the definition of `function-name` and returns `function-name`.

Note that `future-common-lisp:fmakunbound` is just like `fundefine`.

If the function is encapsulated, `future-common-lisp:fmakunbound` removes both the basic definition and the encapsulations. Some types of function specs (`:location` for example) do not implement `future-common-lisp:fmakunbound`. Using `future-
common-lisp:fmakunbound on a :within function spec removes the replacement of function-to-affect, putting the definition of within-function back to its normal state. Using future-common-lisp:fmakunbound on a method's function spec removes the method completely, so that future messages or generic functions will be handled by some other method.

for keyword for loop

for is one of the iteration driving clauses for loop. As described below, there are numerous variants for this keyword.

The optional argument, data-type is reserved for data type declarations. It is currently ignored.

for var {data-type} from expr1 to expr2 {by expr3}

To iterate upward. Performs numeric iteration.

var is initialized to expr1, and on each succeeding iteration is incremented by expr3 (default 1). If the to phrase is given, the iteration terminates when var becomes greater than expr2. Each of the expressions is evaluated only once, and the to and by phrases can be written in either order.

Note that the to variant appropriate for the direction of stepping must be used for the endtest to be formed correctly; that is, the code does not work if expr3 is negative or 0.

data-type defaults to fixnum. The keyword as is equivalent to the keyword for.

Examples:

(defun loop1 ()
  (loop for i from 1 to 10
        collect i)) => LOOP1
(loop1) => (1 2 3 4 5 6 7 8 9 10)

(defun loop2 ()
  (loop for i from 0 to 5 by 1
        do
        (princ i)) => LOOP2
(loop2) => 012345NIL

(defun loop3 (inc)
  (loop as x from 0 by inc to (+ inc 4)
        do
        (princ x)
        (setq x (+ x 1))) => LOOP3
(loop3 1) => 024NIL

for var {data-type} from expr1 downto expr2 {by expr3}
To iterate *downward*. Performs numeric iteration. var is initialized to expr1, and on each succeeding iteration is decremented by expr3, and the endtest is adjusted accordingly.

Examples:

```lisp
(defun loop3 ()
  (loop for my-number from 7 by 2 downto -2
do
        (princ my-number)(princ " ")) => LOOP3
(loop3) => 7 5 3 1 -1 NIL
```

*for* var {data-type} from *expr1* {below *expr2*} {by *expr3*}

Loop will terminate when the variable of iteration, *expr1*, is *greater than or equal* to some terminal value, *expr2*.

Examples:

```lisp
(defun loop1 ()
  (loop for i from 0 below 10
do
        (princ i))) => LOOP1
(loop1) => 0123456789NIL
```

```lisp
(defun loop2 ()
  (loop for my-number from 7.5 by .5 below 12
do
        (princ my-number)(princ " ")) => LOOP2
(loop2) => 7.5 8.0 8.5 9.0 9.5 10.0 10.5 11.0 11.5 NIL
```

*for* var {data-type} from *expr1* {above *expr2*} {by *expr3*}

Loop will terminate when the variable of iteration is *less than or equal* to some terminal value.

Examples:
(defun loop1 ()
    (loop for my-number from 12 by .5 above 7.5
do
        (print my-number))) => LOOP1
(loop1) =>
12
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0 NIL

for var {data-type} downfrom expr1 {by expr2}
Used to iterate downward with no limit.
Examples:

(defun loop-downfrom (num)
    (loop for x downfrom 8 by num
do
        (print x))) => LOOP-DOWNFROM
(loop-downfrom 1)
8
7
6
5... ;infinite

for var {data-type} upfrom expr1 {by expr2}
Used to iterate upward with no limit.
Examples:

(defun loop-upfrom ()
    (loop for x upfrom -2 by 2
do
        (print x))) => LOOP-UPFROM
(loop-upfrom)
-2
0
2
4... ;infinite

for var {data-type} in expr1 {by expr2}
Iterates over each of the elements in the list \emph{expr1}. If the \texttt{by} subclause is present, \emph{expr2} is evaluated once on entry to the loop to supply the function to be used to fetch successive sublists, instead of \texttt{cdr}.

Examples:

\begin{verbatim}
(defun loop1 (input-list)
  (loop for x in input-list
        for i from 0
        do
          (princ (list i x))) => LOOP1
  (loop1 '(a b (c d) e)) => (0 A)(1 B)(2 (C D))(3 E)NIL

for var \{data-type\} on expr \{by expr2\}
\end{verbatim}

Like the previous \texttt{for} format, except that \emph{var} is set to successive sublists of the list instead of successive elements. Note that since \emph{var} is always a list, it is not meaningful to specify a \texttt{data-type} unless \emph{var} is a \texttt{destructuring pattern}, as described in the section on \texttt{destructuring}. Note also that \texttt{loop} uses a \texttt{null} rather than an \texttt{atom} test to implement both this and the preceding clause.

Example:

\begin{verbatim}
(defun loop1 (input-list)
  (loop for sub1 on input-list
        do
          (print sub1))) => LOOP1
  (loop1 '(a b c (k c) d)) =>
    (A B C (K C) D)
    (B C (K C) D)
    (C (K C) D)
    ((K C) D)
    (D) NIL

In contrast to what \texttt{in} would do
\end{verbatim}

\begin{verbatim}
(defun loop1 (input-list)
  (loop for sub1 in input-list
        do
          (print sub1))) => LOOP1
  (loop1 '(a b c (k c) d)) =>
    A
    B
    C
    (K C)
    D NIL

for var \{data-type\} = expr
\end{verbatim}

On each iteration, \emph{expr} is evaluated and \emph{var} is set to the result.

\begin{verbatim}
for var \{data-type\} = expr1 then expr2
\end{verbatim}
var is bound to expr1 when the loop is entered, and set to expr2 (reevaluated) at all but the first iteration. Since expr1 is evaluated during the binding phase, it cannot reference other iteration variables set before it; for that, use the following:

Examples:

```lisp
(defun loop1 (x)
  (loop for stepper = x then (* stepper x)
        do
          (print stepper))) => LOOP1
(loop1 3)
  3
  9
  27
  81... ; infinite loop
```

```lisp
for var {data-type} first expr1 then expr2
```

Sets var to expr1 on the first iteration, and to expr2 (reevaluated) on each succeeding iteration. The evaluation of both expressions is performed inside of the loop binding environment, before the loop body. This allows the first value of var to come from the first value of some other iteration variable, allowing such constructs as:

```lisp
(loop for term in poly
      for ans first (car term) then (gcd ans (car term))
      finally (return ans))
```

```lisp
for var {data-type} being expr and its path ...
```

```lisp
for var {data-type} being {each | the} path ...
```

This provides a user-definable iteration facility. path names the manner in which the iteration is to be performed. The ellipsis indicates where various path-dependent preposition/expression pairs can appear.

See the section "Iteration Paths for loop".

Examples:

```lisp
(define-loop-sequence-path ascii-char
  (lambda (string i)
    (length) => NIL
    (ascii-code (aref string i)))
  (ascii-char of "ABC"
    doing
    (print x)) =>
  65
  66
  67 NIL ; 65 is the ascii equivalent of "A"
```
(loop for a being the array-elements of q using (index ai)
collecting (lambda (x)
    when (> x a)
    (aset x q ai)))))

See the section "Iteration-Driving Clauses".

**force-output** &optional output-stream

Function

Some streams are implemented in an asynchronous, or buffered, manner. **force-output** initiates the emptying of any internal buffers, but returns **nil** without waiting for completion or acknowledgment. **Output-stream** if unspecified or **nil**, defaults to ***standard-output***, and if **t**, is ***terminal-io***.

**:force-output**

Message

Causes any buffered output to be sent to a buffered asynchronous device, such as the Chaosnet. It does not wait for it to complete; use **:finish** for that. If a stream supports **:force-output**, then **:tyo**, **:string-out**, and **:line-out** might have no visible effect until a **:force-output** is done. If the stream does not handle this, the default handler ignores it.

**fourth** list

Function

Returns the fourth element of the list. **fourth** is equivalent to:

(nth 3 list)

(setq letters '(a b c d e f)) => (A B C D E F)

(fourth letters) => D

For a table of related items: See the section "Functions for Extracting from Lists".

**dbg:frame-active-p** frame

Function

Indicates whether **frame** is an active frame.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Frame is not active</td>
</tr>
<tr>
<td>not nil</td>
<td>Frame is active</td>
</tr>
</tbody>
</table>

Caution: Use this function only within the context of the **dbg:with-erring-frame** macro.

For a table of related items: See the section "Functions for Examining Stack Frames".
**Function**

(*debug-frame-arg-value frame arg-name-or-number &optional callee-context no-error-p*)

Returns the value of the *n*th argument to *frame*. Returns a second value, which is a locative pointer to the word in the stack that holds the argument. If *n* is out of range, it takes action based on *no-error-p*: if *no-error-p* is *nil*, it signals an error, otherwise it returns *nil*. *n* can also be the name of the argument (a symbol, but it need not be in the right package). Each argument passed for an &rest parameter counts as a separate argument when *n* is a number. *debug-frame-arg-value* controls whether you get the caller or callee copy of the argument (original or possibly modified.)

**Caution:** Use this function only within the context of the *debug-with-erring-frame* macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

---

**Function**

(*debug-frame-local-value frame local-name-or-number &optional no-error-p*)

Returns the value of the *n*th local variable in *frame*. *n* can also be the name of the local variable (a symbol, but it need not be in the right package). It returns a second value, which is a locative pointer to the word in the stack that holds the local variable. If *n* is out of range, then the action is based on *no-error-p*: if *no-error-p* is *nil*, it signals an error, otherwise it returns *nil*.

**Caution:** Use this function only within the context of the *debug-with-erring-frame* macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

---

**Function**

(*debug-frame-next-active-frame frame*)

Returns a frame pointer to the next active frame following *frame*. If *frame* is the last active frame on the stack, it returns *nil*.

"Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack).

**Caution:** Use this function only within the context of the *debug-with-erring-frame* macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

---

**Function**

(*debug-frame-next-interesting-active-frame frame*)

Returns a frame pointer to the next interesting active frame following *frame*. If *frame* is the last active frame on the stack, it returns *nil*.

"Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack).

**Caution:** Use this function only within the context of the *debug-with-erring-frame* macro.

For a table of related items: See the section "Functions for Examining Stack Frames".
Returns a frame pointer to the next interesting active frame following \textit{frame}. If \textit{frame} is the last interesting active frame on the stack, it returns \texttt{nil}.

"Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack).

"Interesting active frames" include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as the frames for \texttt{eval}, \texttt{zl:apply}, \texttt{funcall}, \texttt{let}, and other basic Lisp special forms. The list of such functions is the value of the system constant, \texttt{dbg:*uninteresting-functions*}. 

\textbf{Caution:} Use this function only within the context of the \texttt{dbg:with-erring-frame} macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

\begin{verbatim}
dbg:frame-next-nth-active-frame\frame &optional (count 1) skip-invisible
\end{verbatim}

\textbf{Function}

Goes up the stack by \texttt{count} active frames from \texttt{frame} and returns a frame pointer to that frame. It returns a second value that is not \texttt{nil}. When \texttt{count} is positive, this is like calling \texttt{dbg:frame-next-active-frame count} times; \texttt{count} can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and \texttt{nil}.

"Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack).

\textbf{Caution:} Use this function only within the context of the \texttt{dbg:with-erring-frame} macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

\begin{verbatim}
dbg:frame-next-nth-interesting-active-frame\frame &optional (count 1) skip-invisible
\end{verbatim}

\textbf{Function}

Goes up the stack by \texttt{count} interesting active frames from \texttt{frame} and returns a frame pointer to that frame. It returns a second value that is not \texttt{nil}. When \texttt{count} is positive, this is like calling \texttt{dbg:frame-next-interesting-active-frame count} times; \texttt{count} can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and \texttt{nil}.

"Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack).

"Interesting active frames" include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as the frames for \texttt{eval}, \texttt{zl:apply}, \texttt{funcall}, \texttt{let}, and other basic Lisp special forms. The list of such functions is the value of the system constant, \texttt{dbg:*uninteresting-functions*}. 


Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

**dbg:frame-next-nth-open-frame**  
frame &optional (count 1) skip-invisible

Function

Goes up the stack by count open frames from frame and returns a frame pointer to that frame. It returns a second value that is not `nil`. When count is positive, this is like calling `dbg:frame-next-open-frame` count times; count can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and `nil`.

"Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack).

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

**dbg:frame-next-open-frame**  
frame

Function

Returns a frame pointer to the next open frame following frame-pointer. If frame is the last open frame on the stack, it returns `nil`.

"Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack).

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

**dbg:frame-number-of-locals**  
frame

Function

Returns the number of local variables allocated for frame.

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

**dbg:frame-number-of-spread-args**  
frame &optional (type :supplied)

Function
Returns the number of "spread" arguments that were passed in frame. (These are the arguments that are not part of a &rest parameter.) Sending a message to an instance results in two implicit arguments being passed internally along with the other arguments. These implicit arguments are included in the count.

_type_ requests more specific definition of the number:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:supplied</td>
<td>Returns the number of arguments that were actually passed by the caller, except for arguments that were bound to a &amp;rest parameter. This is the default.</td>
</tr>
<tr>
<td>:expected</td>
<td>Returns the number of arguments that were expected by the function being called.</td>
</tr>
<tr>
<td>:allocated</td>
<td>Returns the number of arguments for which stack locations have been allocated. In the absence of a &amp;rest parameter, this is the same as :expected for compiled functions, and the same as :supplied for interpreted functions. If stack locations were allocated for arguments that were bound to a &amp;rest parameter, they are included in the returned count.</td>
</tr>
</tbody>
</table>

These values would all be the same except in cases where a wrong-number-of-arguments error occurred, or where there are optional arguments (expected but not supplied).

Caution: Use this function only within the context of the dbg:with-erring-frame macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

dbg:frame-out-to-interesting-active-frame frame  Function

Returns either frame (if it points to an interesting active frame) or the previous interesting active frame before frame-pointer. (This is what the :Previous Frame command c-m-U in the Debugger does.)

"Interesting active frames" include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as the frames for eval, zl:apply, funcall, let, and other basic Lisp special forms. The list of such functions is the value of the system constant, dbg:*uninteresting-functions*.

Caution: Use this function only within the context of the dbg:with-erring-frame macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

dbg:frame-previous-active-frame frame  Function

Returns a frame pointer to the previous active frame before frame. If frame is the first active frame on the stack, it returns nil.
"Previous" means the frame of a procedure that was invoked less recently (the caller of this frame; towards the base of the stack).

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

`dbg:frame-previous-interesting-active-frame` frame

Function

Returns a frame pointer to the previous interesting active frame before `frame`. If `frame` is the first interesting active frame on the stack, it returns `nil`.

"Previous" means the frame of a procedure that was invoked less recently (the caller of this frame; towards the base of the stack).

"Interesting active frames" include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as the frames for `eval`, `zl:apply`, `funcall`, `let`, and other basic Lisp special forms. The list of such functions is the value of the system constant, `dbg:*uninteresting-functions*`.

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

`dbg:frame-previous-open-frame` frame

Function

Returns a frame pointer to the previous open frame before `frame`. If `frame` is the first open frame on the stack, it returns `nil`.

"Previous" means the frame of a procedure that was invoked less recently (the caller of this frame; towards the base of the stack).

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

`dbg:frame-real-function` frame

Function

Returns either the function object associated with `frame` or the value of self when the frame was the result of sending a message to an instance.

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".
**dbg:frame-real-value-disposition** frame

Function

Returns a symbol indicating how the calling function is going to handle the values to be returned by this frame. If the calling function just returns the values to its caller, then the symbol indicates how the final recipient of the values is going to handle them.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:ignore</td>
<td>The values would be ignored; the function was called for effect.</td>
</tr>
<tr>
<td>:single</td>
<td>The first value would be received and the rest would not; the function was called for value.</td>
</tr>
<tr>
<td>:multiple</td>
<td>All the values would be received; the function was called for multiple values. It returns a second value indicating the number of values expected. nil indicates an indeterminate number and is always returned.</td>
</tr>
</tbody>
</table>

**Caution:** Use this function only within the context of the **dbg:with-erring-frame** macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

**dbg:frame-self-value** frame &optional instance-frame-only

Function

Returns the value of self in frame, or nil if self does not have a value. If instance-frame-only is not nil then it returns nil unless this frame is actually a message-sending frame created by send.

**Caution:** Use this function only within the context of the **dbg:with-erring-frame** macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

**fresh-line** &optional output-stream

Function

Outputs a newline only if the stream is not already at the start of a line. If for any reason this cannot be determined, then a newline is output anyway. This guarantees that the stream will be on a fresh line while consuming as little vertical space as possible. fresh-line returns t if it output a newline, otherwise it returns nil. output-stream, which, if unspecified or nil, defaults to *standard-output*, and if t, is *terminal-io*.

```lisp
(progn (princ 'foo) (terpri) (princ 'bar) (fresh-line) (princ 'baz) nil)
  FOO
  BAR
  BAZ
  => NIL
```
(progn (princ 'foo) (terpri) (fresh-line)
  (princ 'bar) (fresh-line) (terpri)
  (princ 'baz) nil)
FOO
BAR

BAZ
=> NIL

:fresh-line

Message
Tells the stream to position itself at the beginning of a new line. If the stream is already at the beginning of a fresh line it does nothing; otherwise it outputs a carriage return. For streams that do not support this, the default handler always outputs a carriage return.

fround number &optional (divisor 1)

Function
Like round, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If number is a floating-point number and divisor is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as number.

Round returns the floating point equivalent of the integer nearest to number, or nearest to the quotient of number divided by divisor. If number is exactly 0.5 greater than an integer, the even floating point equivalent of the two integers closest to number, or closest to the quotient of number divided by divisor is returned. A second value, the remainder, is also returned. The remainder returned is the same as that returned by round applied to the same arguments.

Examples:

  (fround 5) => 5.0 and 0
  (fround -5) => -5.0 and 0
  (fround 5.2) => 5.0 and 0.19999981
  (fround -5.2) => -5.0 and -0.19999981
  (fround 5 3) => 2.0 and -1
  (fround -5 3) => -2.0 and 1
  (fround 5.2 4) => 1.0 and 1.1999998
  (fround -5.2 4) => -1.0 and -1.1999998
  (fround 4.2d0) => 4.0d0 and 0.2000000000000018d0
  (fround -4.2d0) => -4.0d0 and -0.2000000000000018d0

For a table of related items: See the section "Functions that Divide and Return Quotient as Floating-point Number".
**zl:fset**  
*sym definition*  
*Function*  
Stores *definition*, which can be any Lisp object, into *sym*’s function cell. It returns *definition*.

See the section “Functions Relating to the Function Cell of a Symbol”.

**zl:fset-carefully**  
*function-spec definition* &optional *no-query-flag*  
*Function*  
This function is obsolete. It is equivalent to:

```
(fdefine symbol definition t force-flag)
```

**zl:fsignal**  
*format-string* &rest *format-args*  
*Function*  
This is a simple function for signalling when you do not care to use a particular condition.  
*zl:fsignal* signals *dbg:proceedable-ferror*. (See the flavor *dbg:proceedable-ferror*.)  
The arguments are passed as the :format-string and :format-args init keywords to the error object.

**Note:**  
*zl:fsignal* is now obsolete. Use *cerror* in your new programs instead.

For a table of related items: See the section “Condition-Checking and Signalling Functions and Variables”.

**zl:fsymeval**  
*symbol*  
*Function*  
In your new programs, we recommend that you use the function *symbol-function*,  
which is the Common Lisp equivalent of the function *zl:fsymeval*.

Returns *symbol*’s definition, the contents of its function cell. If the function cell is empty,  
*zl:fsymeval* signals an error.

See the section “Functions Relating to the Function Cell of a Symbol”.

**ftruncate**  
*number* &optional (*divisor 1*)  
*Function*  
Like *truncate*, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If *number* is a floating-point number and *divisor* is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as *number*.

Returns the floating point equivalent of the integer nearer to zero of the two integers closest to *number*, or closest to the quotient of *number* divided by *divisor*. A second value, the remainder, is also returned. The remainder returned is the same as that returned by *truncate* applied to the same arguments.

**Examples:**
```
(ftruncate 5) => 5.0 and 0
```
(ftruncate -5) => -5.0 and 0
(ftruncate 5.2) => 5.0 and 0.19999981
(ftruncate -5.2) => -5.0 and -0.19999981
(ftruncate 5 3) => 1.0 and 2
(ftruncate -5 3) => -1.0 and -2
(ftruncate 5.2 4) => 1.0 and 1.1999998
(ftruncate -5.2 4) => -1.0 and -1.1999998
(ftruncate 4.2d0) => 4.0d0 and 0.20000000000000018d0
(ftruncate -4.2d0) => -4.0d0 and -0.20000000000000018d0

For a table of related items: See the section "Functions that Divide and Return Quotient as Floating-point Number".

**funcall fn &rest args**  
(funcall fn a1 a2 ... an) applies the function fn to the arguments a1, a2, ..., an. fn cannot be a special form nor a macro; this would not be meaningful. Example:

- (cons 1 2) => (1 . 2)
- (setq cons '+) => +
- (funcall cons 1 2) => 3
- (cons 1 2) => (1 . 2)

This shows that the use of the symbol cons as the name of a variable and the use of that symbol as the name of a function do not interact. The funcall form evaluates the variable and gets the symbol +, which is the name of a different function. The cons form invokes the function named cons.

Note: The Maclisp functions subrcall, lsubrcall, and zl:arraycall are not needed in Symbolics Common Lisp; funcall is just as efficient. zl:arraycall is provided for compatibility; it ignores its first subform (the Maclisp array type) and is otherwise identical to aref. subrcall and lsubrcall are not provided.

- (setq + subfn (symbol-function '-))
- (defun subfn(x y) (+ x y))
- (subfn 2 1) => 3
- (funcall subfn 2 1) => 1

(defun size-of-form (form print-function)  
"print-function should be princ-to-string or prin1-to-string"
  (length (funcall print-function form)))

In the previous example, the print length of a form is determined by using funcall on one of two print functions.

See the section "Functions for Function Invocation".

**function name arglist result-type1 result-type2 ...**  
Declaration
Equivalent to \textbf{ftype type function-name-1 function-name-2}, but might be more convenient.

\textbf{function function} \hfill \textit{Special Form}

Means different things, depending on whether \textit{function} is a function or the name of a function. (Note that in neither case is \textit{function} evaluated.) The name of a function is a symbol or a function-spec list. See the section "Function Specs". A function is typically a list whose car is the symbol \texttt{lambda}; however, there are several other kinds of functions available. See the section "Kinds of Functions".

If you want to pass an anonymous function as an argument to a function, you could just use \texttt{quote}. For example:

\begin{verbatim}
(mapc (quote (lambda (x) (car x))) some-list)
\end{verbatim}

The compiler and interpreter cannot tell that the first argument is going to be used as a function; for all they know, \texttt{mapc} treats its first argument as a piece of list structure, asking for its \texttt{car} and \texttt{cdr} and so forth. The compiler cannot compile the function; it must pass the lambda-expression unmodified. This means that the function does not get compiled, which makes it execute more slowly than it might otherwise. The interpreter cannot make references to free lexical variables work by making a lexical closure; it must pass the lambda-expression unmodified.

The \texttt{function} special form is the way to say that a lambda-expression represents a function rather than a piece of list structure. You just use the symbol \texttt{function} instead of \texttt{quote}:

\begin{verbatim}
(mapc (function (lambda (x) (car x))) some-list)
\end{verbatim}

To ease typing, the reader converts \texttt{#'thing} into \texttt{(function thing)}. So \texttt{"} is similar to \texttt{'} except that it produces a \texttt{function} form instead of a \texttt{quote} form. So the above form could be written as:

\begin{verbatim}
(mapc #'(lambda (x) (car x)) some-list)
\end{verbatim}

If \texttt{function} is not a function but the name of a function (typically a symbol, but in general any kind of function spec), \texttt{function} returns the definition of \textit{function}; it is like \texttt{fdefinition} except that it is a special form instead of a function, and so

\begin{verbatim}
(function fred)
\end{verbatim}

is like

\begin{verbatim}
(fdefinition 'fred)
\end{verbatim}

which is like

\begin{verbatim}
(fsymeval 'fred)
\end{verbatim}

since \texttt{fred} is a symbol. Note that you cannot use \texttt{fsymeval} in CLOE.

If \texttt{function} is the name of a local function defined with \texttt{flet} or \texttt{labels}, then \texttt{(function function)} produces a lexical closure of \textit{function}, just like \texttt{(function (lambda...))}.

Another way of explaining \texttt{function} is that it causes \textit{function} to be treated the same way as it would as the car of a form. Evaluating the form \texttt{(function argl}
arg2...) uses the function definition of function if it is a symbol, and otherwise expects function to be a list that is a lambda-expression. Note that the car of a form cannot be a nonsymbol function spec, to avoid difficult-to-read code. This can be written as:

(funcall (function spec) args...)

You should be careful about whether you use #' or '. Suppose you have a program with a variable x whose value is assumed to contain a function that gets called on some arguments. If you want that variable to be the car function, there are two things you could say:

(setq x 'car)

or

(setq x #'car)

The former causes the value of x to be the symbol car, whereas the latter causes the value of x to be the function object found in the function cell of car. When the time comes to call the function (the program does (funcall x ...)), either of these two work because if you use a symbol as a function, the contents of the symbol’s function cell are used as the function. The former case is a bit slower, because the function call has to indirect through the symbol, but it allows the function to be redefined, traced, or advised. (See the special form trace. See the special form advise.) The latter case, while faster, picks up the function definition out of the symbol car and does not see any later changes to it.

**function** (( arg1-type arg2-type ... ) value-type )

Type Specifier

function is the type specifier for the predefined Lisp object of that name.

The list syntax is for declaration. Every element of this type is a function that accepts arguments at least of the types specified by the argj-type forms, and returns a value that is a member of the types specified by the value-type form.

Examples:

(defun fun-example (num) (+ num num)) => FUN-EXAMPLE
(typep ‘fun-example ‘function) => T
(sys:type-arglist ‘function) => NIL and T
(functionp ‘fun-example) => T

See the section "Data Types and Type Specifiers".

See the section "Functions".

**sys:function-cell-location** sym

Function

Returns a locative pointer to sym’s function cell. See the section "Cells and Locatives". It is preferable to write:

(locf (zl:fsymeval sym))

rather than calling this function explicitly. See the section "Functions Relating to the Function Cell of a Symbol".
si:function-encapsulated-p function-spec  

Function

Looks at the debugging info alist to check whether function-spec is an encapsulation.

clos:function-keywords method  

Generic Function

Returns a list of the keywords for method as its first value, and a boolean indicating whether &allow-other-keys was specified as its second value.

method  
A method object.

sys:function-parent function-spec &optional definition-type  

Function

When a symbol's definition is produced as the result of macro expansion of a source definition, so that the symbol's definition does not appear textually in the source, the editor cannot find it. The accessor, constructor, and alterant macros produced by a defstruct are an example of this. The sys:function-parent declaration can be inserted in the source definition to record the name of the outer definition of which it is a part.

The declaration consists of the following:

(sys:function-parent name type)

name is the name of the outer definition. type is its type, which defaults to defun. See the section "How Programs Manipulate Definitions". Declarations are explained in another section. See the section "Declarations".

sys:function-parent  

is a function related to the declaration. It takes a function spec and returns nil or another function spec. The first function spec's definition is contained inside the second function spec's definition. The second value is the type of definition.

Two examples:

(defsubst foo (x y)
  (declare (sys:function-parent bar))
  ...
)

(defmacro defxxx (name ...)
  '(zl:local-declare ((sys:function-parent ,name defxxx))
    (defmacro ...)
    (defmacro ...
      ))

si:function-spec-get function-spec indicator  

Function
Returns the value of the indicator property of function-spec, or nil if it doesn’t have such a property.

si:function-spec-putprop function-spec value indicator

Gives function-spec an indicator property whose value is value.

functionp x &optional allow-special-forms

Returns t if its argument x is a function (essentially, something that is acceptable as the first argument to apply), otherwise it returns nil. Under Genera, in addition to interpreted, compiled, and built-in functions, functionp is true of closures, select-methods, and symbols whose function definition is functionp. See the section "Other Kinds of Functions". functionp is not true of objects that can be called as functions but are not normally thought of as functions: arrays, stack groups, entities, and instances.

Compatibility Note: Symbolics Common Lisp (but not CLOE) provides the optional argument allow-special-forms. If allow-special-forms is specified and non-nil, then functionp is true of macros and special-form functions (those with quoted arguments). Normally functionp returns nil for these since they do not behave like functions. allow-special-forms might not work in other implementations of Common Lisp. functionp returns nil when it is called on a symbol that does not have a function definition, although Common Lisp specifies that functionp of a symbol is always t.

As a special case, functionp of a symbol whose function definition is an array returns t, because in this case the array is being used as a function rather than as an object.

Under CLOE, closures (results of the function special form), lambda expressions, and names of functions are all considered functions by functionp. When applied to symbols, functionp always returns true, regardless of whether or not they are currently fboundp.

(functiomp #':eql) => t
(functiomp ':eql) => t
(functiomp '(:lambda (x) (+ 5 x))) => t

fundefine function-spec

Removes the definition of function-spec and returns function-spec. For symbols this is equivalent to fmakunbound. If the function is encapsulated, fundefine removes both the basic definition and the encapsulations. Some types of function specs (:location for example) do not implement fundefine. Using fundefine on a :within function spec removes the replacement of function-to-affect, putting the definition of within-function back to its normal state. Using fundefine on a method’s func-
tion spec removes the method completely, so that future messages or generic functions will be handled by some other method.

\textbf{g-l-p array}  

\textit{Function}

If \textit{array} has a fill pointer, \textbf{g-l-p} returns a list that stops at the fill pointer, if you never modify the fill-pointer except with \texttt{zl:array-push}, \texttt{zl:array-pop} and so on. \textit{array} must be a general (\texttt{sys:art-q-list}) array. Example:

\begin{verbatim}
(setq a (zl:make-array 4 :type 'art-q-list))
(aref a 0) => nil
(setq b (g-l-p a)) => (nil nil nil nil)
(setf (car b) t)
b => (t nil nil nil)
(setf (aref a 2) 30)
b => (t nil 30 nil)
\end{verbatim}

\textbf{gcd &rest integers}  

\textit{Function}

If one argument is given, the absolute value is returned. If there are no arguments, the returned value is 0.

Examples:

\begin{verbatim}
(gcd) => 0
(gcd -9) => 9
(gcd 36 48) => 12
(gcd 16 72 40 24) => 8
\end{verbatim}

For a table of related items, see the section "Arithmetic Functions".

\textbf{zl:gcd integer1 integer2 &rest more-integers}  

\textit{Function}

Returns the greatest common divisor of all its arguments. The arguments must be integers. With one argument \texttt{integer}, it returns the absolute value of \texttt{integer}, and with no arguments, it returns 0. The result returned is always returns a non-negative integer.

\begin{verbatim}
(gcd -15 105) \rightarrow 15
(gcd 15 12 9) \rightarrow 3
(gcd 5 7 11 18) \rightarrow 1
(gcd) \rightarrow 0
\end{verbatim}

The following function is a synonym of \texttt{zl:gcd}:

\texttt{zl:}
For a table of related items: See the section "Arithmetic Functions".

**flavor:generic** *generic-function-name*  
*Special Form*

Evaluates to the generic function object for *generic-function-name* (which is not evaluated). This is used when there is a prologue function so that the function definition of *generic-function-name* is not itself the generic function. This is used in conjunction with the :function option to defgeneric. For example:

```
(apply (flavor:generic make-instance) new-instance init-options)
```

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**clos:generic-flet**  
*Special Form*

Symbolics CLOS does not support **clos:generic-flet**.

**clos:generic-function**  
*Macro*

Symbolics CLOS does not support **clos:generic-function**.

**sys:generic-function**  
*Type Specifier*

**clos:generic-labels**  
*Special Form*

Symbolics CLOS does not support **clos:generic-labels**.

**gensym** &optional* arg*  
*Function*

Invents a print-name, and creates a new symbol with that print-name. It returns the new, uninterned symbol.

The invented print-name is a character prefix (the value of *gensym-prefix*) followed by the decimal representation of a number (the value of *gensym-counter*), for example, "G0001". The number is increased by 1 every time gensym is called.

If the argument *arg* is present and is a fixnum, then *gensym-counter* is set to *arg*. If *arg* is a string or a symbol, then *gensym-prefix* is set to the string or the symbol’s print-name. After handling the argument, gensym creates a symbol as it would with no argument. Examples:

```
if (gensym) => #:G3310
then (gensym "foo") => #:f003311
(gensym 32) => #:lfoo332
(gensym) => #:lfoo331
```

**gensym** is usually used to create a symbol that should not normally be seen by the user, and whose print-name is unimportant, except to allow easy distinction by eye
between two such symbols. The optional argument is rarely supplied because it changes the default prefix for future calls to **gensym**. To create a symbol with a particular prefix when using Genera, use **sys:gensymbol**. See the function **sys:gensymbol**.

The name **gensym** comes from "generate symbol", and the symbols produced by it are often called "gensyms". This function is also useful for obtaining anonymous, locally bound variables created by macros at compile time. In the following example, macro **do-vector** is created by using **gensym**. This form is similar to **dolist** because **var** is successively bound to vector elements.

```lisp
(defmacro do-vector((var vector &optional result) &body forms)
  (let ((genvar1 (gensym))
        (genvar2 (gensym)))
    `(do ((,genvar1 (length ,vector))
          (,genvar2 0 (+ 1 ,genvar2)))
         ((>= ,genvar2 ,genvar1) ,result)
          (let ((,var (elt ,vector ,genvar2)))
            ,@forms))))
```

```lisp
(do-vector (element '#(foo bar baz)) (print element))
FOO
BAR
BAZ
```

For a list of related functions: See the section "Functions for Creating Symbols".

**zl:gensym** &optional **x**

Invents a print-name, and creates a new symbol with that print-name. It returns
the new, uninterned symbol.

If the argument **x** is present and is a fixnum, then **future-common-lisp:*gensym-counter*" is set to **x** and incremented. If **x** is a string or a symbol, then **cli::*gensym-prefix*" is set to the first character of the string or of the symbol's print-name. After handling the argument, **gensym** creates a symbol as it would with no argument. Examples:

```lisp
if (zl:gensym) => #:G3310
then (zl:gensym "foo") => #:F3311
(zl:gensym 32) => #:F0033
(zl:gensym) => #:F0034
```

Note that the number is in decimal and always has four digits, and the prefix is always one character.

See the function **gensym**.

See the section "Functions for Creating Symbols".

**sys:gensymbol** &optional (prefix "G") **count**

Function
Like `gensym`, invents a print-name and creates a new symbol with that print-name. It returns the new, uninterned symbol. Unlike `gensym`, however, if a `prefix` is given, it does not become the default for future calls to `sys:gensymbol`. For example:

```lisp
if (sys:gensymbol) => #:G0035
then (sys:gensymbol "foo") => #:|foo36|
(sys:gensymbol) => #:G0037
```

Contrasted with:

```lisp
if (gensym) => #:G0038
then (gensym "foo") => #:|foo39|
(gensym) => #:|foo40|
```

`sys:gensymbol` is the recommended way to get symbols with a specific prefix.

For a list of related functions: See the section "Functions for Creating Symbols".

---

### `gentemp` &optional `(prefix "T")` `package`

Creates and returns a new symbol as `gensym` does, but `gentemp` interns the symbol in `package`. `Package` defaults to the current package, that is, the value of `*package*`. `gentemp` guarantees that the generated symbol is a new one not already existing in `package`. There is no provision for resetting the `gentemp` counter and the prefix is not remembered from one call to the next. If `prefix` is omitted, "T" is used.

```lisp
(gentemp) => T1
(defparameter T2 42)
(gentemp) => T3
(gentemp "FOO") => FOO4
```

See the section "Functions for Creating Symbols".

---

### `get symbol indicator` &optional `default`

Searches the property list of `symbol` for an indicator that is `eq` to `indicator`. (See the section "Property Lists".) The first argument must be a symbol. If a matching indicator is found, the corresponding value is returned; otherwise `default` is returned. If `default` is not specified, `nil` is used. Note that there is no way to distinguish an absent property from one whose value is `default`.

To give a symbol a property, use:

```lisp
(setf (get symbol indicator) value)
```

Suppose that the property list of `eagle` is

```lisp
(color (brown white) food snakes seed-eater nil)
```

Then, for example:
(get 'eagle 'color) => (BROWN WHITE)
(get 'eagle 'food) => SNAKES
(get 'eagle 'seed-eater) => NIL
(get 'eagle 'beak "No such indicator") => "No such indicator"

**setf** can be used with **get** to create a new property-value pair, possibly replacing an old pair with the same name. For example:

(setf (get 'eagle 'food) '(mice snakes)) => (MICE SNAKES)

For a table of related items: See the section "Functions That Operate on Property Lists".

---

**zl:get symbol indicator**

Looks up symbol’s indicator property. (See the section "Property Lists".) If it finds such a property, it returns the value; otherwise, it returns nil. **zl:get** uses the symbol’s associated property list. For example, if the property list of foo is (baz 3), then:

(zl:get 'foo 'baz) => 3
(zl:get 'foo 'zoo) => nil

For a table of related items: See the section "Functions That Operate on Property Lists".

---

**flavor:get-all-flavor-components flavor-name &optional env**

Returns a list of the components of the flavor flavor-name, in the sorted ordering of flavor components. Any duplicate flavors are eliminated from this list by the flavor ordering mechanism. See the section "Ordering Flavor Components".

For example:

(flavor:get-all-flavor-components 'tv:minimum-window) =>

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

---

**get-dispatch-macro-character disp-char sub-char &optional (a-readtable *readtable*)**

Returns the macro-character function for sub-char under disp-char, or nil if there is no function associated with sub-char. If sub-char is one of the ten decimal digits, **get-dispatch-macro-character** always returns nil. If sub-char is a lowercase character, its uppercase equivalent is always used instead.

An error is signalled if the specified disp-char is not a dispatch character in the specified readtable.
(get-dispatch-macro-character #\# #\') =>
#<LEXICAL-CLOSURE (:INTERNAL GET-DISPATCH-MACRO-CHARACTER 0) 36057616>

(get-dispatch-macro-character #\# #\1) => NIL

Note that because get-dispatch-macro-character returns a lexical closure, subsequent calls will not necessarily return the same object. This may be changed in a future release.

(let ((*readtable* (copy-readtable nil)))
  (get-dispatch-macro-character #\# #\)
  (get-dispatch-macro-character #\# #\Q))
=> NIL

**zl:get-flavor-handler-for**  
**flavor-name**  
**operation**  

Given a flavor-name and an operation (a function spec that names a generic function or a message), zl:get-flavor-handler-for returns the flavor's method for the operation or nil if it has none.

For example:

(lzl:get-flavor-handler-for 'box-with-cell 'find-neighbors)
=>#<DTP-COMPILED-FUNCTION
  (FLAVOR:METHOD FIND-NEIGHBORS CELL) 20740320>

(lzl:get-flavor-handler-for 'cell ':print-self)
=>#<DTP-COMPILED-FUNCTION
  (FLAVOR:METHOD SYS:PRINT-SELF FLAVOR:VANILLA DEFAULT) 42456350>

Although operation is usually a symbol (naming a generic function) or a keyword (naming a message), it is occasionally a list. For example, names of some generic functions are lists, such as (setf function).

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section “Summary of Flavor Functions and Variables”.

**si:get-font**  
**device**  
**character-set**  
**style**  
&optional (error-p t) inquiry-only  

Given a device, character-set and style, returns a font object that would be used to display characters from that character set in that style on the device. This is useful for determining whether there is such font mapping for a given device/set/style combination.

A font object may be various things, depending on the device.

If error-p is non-nil, this function signals an error if no mapping to a font is found. If error-p is nil and no font mapping is found, si:get-font returns nil.
If inquiry-only is provided, the returned value is not a font object, but some other representation of a font, such as a symbol in the fonts package (for screen fonts) or a string (for printer fonts).

```lisp
(si:get-font si:*b&w-screen* si:*standard-character-set*
 '(:jess :roman :normal))
```

=> #<FONT JESS13 154102066>

```lisp
(si:get-font lgp:*lgp2-printer* si:*standard-character-set*
 '(:swiss :roman :normal) nil t)
```

=> "Helvetica10"

For related information: See the section "Mapping a Character Style to a Font".

**dbg:get-frame-function-and-args frame**

*Function*

Returns a list containing the name of the function for frame-pointer and the values of the arguments.

**Caution:** Use this function only within the context of the **dbg:with-erring-frame** macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

**get-handler-for object operation**

*Function*

Given an object and an operation (a function spec that names a generic function or a message), returns that object's method for the operation, or nil if it has none. When object is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method. If a combined method is returned, you can use the Zmacs command List Combined Methods (\(m-X\)) to find out what it does.

For example:

```lisp
(get-handler-for this-box-with-cell 'count-live-neighbors)
```

=>#<DTP-COMPILED-FUNCTION

(FLAVOR:METHOD 'COUNT-LIVE-NEIGHBORS CELL) 42456350>

```lisp
(get-handler-for this-box-with-cell ':print-self)
```

=>#<DTP-COMPILED-FUNCTION

(FLAVOR:METHOD SYS:PRINT-SELF FLAVOR:VANILLA DEFAULT) 42456350>

Because it is a generic function, you can define methods for **get-handler-for**. The syntax of this is:

```lisp
(defun (get-handler-for (flavor) (operation))
  body)
```
In most cases you should use :or method combination (by supplying the :method-combination option for defflavor) so your method need not know what the flavor:v vanilla method does.

Although operation is usually a symbol (naming a generic function) or a keyword (naming a message), it is occasionally a list. For example, names of some generic functions are lists, such as (setf function).

Note that get-handler-for does not work on named-structures or non-instance streams. You might consider using :operation-handled-p instead.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section “Summary of Flavor Functions and Variables”.

:get-hash key

Message

Find the entry in the hash table whose key is key, and return three values. The first returned value is the associated value of key, or nil if there is no such entry. The second value is t if an entry was found or nil if there is no entry for key in this table. The third value is key, or nil if there was no such key.

This message is obsolete; use zl:gethash instead.

get-macro-character char &optional (a-readtable *readtable*)

Function

Returns two values: the function associated with char, and the value of the non-terminating-p flag. It returns just the symbol nil if char does not have macro-character syntax. For example:

(get-macro-character #\') =>
#<LEXICAL-CLOSURE (INTERNAL GET-MACRO-CHARACTER 0) 16433170>
NIL

(get-macro-character #\-) => NIL

Note that because get-macro-character returns a lexical closure, subsequent calls will not necessarily return the same object. This may be changed in a future release.

(let ((*readtable* (copy-readtable nil)))
  (get-macro-character #\_))
=> NIL

:get-output-buffer

Message

Returns an array and starting and ending indices.

get-output-stream-string stream

Function
Returns a string containing all of the characters output to stream so far. Works in conjunction with make-string-output-stream. stream is reset after each call, thus each call to get-output-stream-string gets only the characters that have been output to the stream since the last such call (or the creation of stream, if no such previous call has been made).

```lisp
(setq s (make-string-output-stream))
=> #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 10602460>

(write-string "Hello" s) => "Hello"

(get-output-stream-string s) => "Hello"

(write-string "Goodbye" s) => "Goodbye"

(get-output-stream-string s) => "Goodbye"

(defun *heading* '("Name " "Rank " "Serial-number "))

(let ((my-stream (make-string-output-stream))
      (list-of-strings *heading*)
      (dolist (str list-of-strings)
        (princ str my-stream))
      (get-output-stream-string my-stream))

(dotimes (i *number-of-names*
            (print (+ i 1) my-stream))
      (get-output-stream-string my-stream))

=>
  
  1.
  2.
  3.
```

**zl:get-pname** symbol

Returns the print-name of the symbol symbol. Example:

```lisp
(zl:get-pname 'xyz) => "xyz"
```

In your new programs, we recommend that you use the function symbol-name which is the Common Lisp equivalent of the function zl:get-pname. See the section "Functions Relating to the Print Name of a Symbol".

**get-properties** plist indicator-list

Searches the property list stored in plist for any of the indicators in indicator-list.
get-properties returns three values. If none of the indicators is found, all three values are \texttt{nil}. If the search is successful, the first two values are the property found and its value and the third value is the tail of the property list whose \texttt{car} is the property found. Thus the third value serves to indicate success or failure and also allows you to restart the search after the property found, if you so desire.

In the following example, note that although \texttt{COLOR} does not precede \texttt{SPEED} in the \texttt{indicator-list}, it does precede \texttt{SPEED} in the property list. Therefore, \texttt{COLOR} is located before \texttt{SPEED}.

\begin{verbatim}
(defvar '*some-symbol*
  (list 'COLOR 'RED 'SPEED 'MYSTICAL 'HIT-POINTS '60))

(get-properties *some-symbol* '(magic speed color)) =>
COLOR
RED
(COLOR RED SPEED MYSTICAL HIT-POINTS 60)
\end{verbatim}

See the section "Functions Relating to the Property List of a Symbol".

\textbf{get-setf-method} \texttt{reference \&optional for-effect} \hspace{1cm} \textit{Function}

In this context, the word "method" has nothing to do with flavors.

Returns five values constituting the \texttt{setf} method for \texttt{reference}, which is a generalized-variable reference. (The five values are described in detail at the end of this discussion.) \texttt{get-setf-method} takes care of error-checking and macro expansion and guarantees to return exactly one store-variable.

\textbf{Compatibility Note:} The optional argument \texttt{for-effect} is a Symbolics extension to Common Lisp. If \texttt{for-effect} is \texttt{t}, you are indicating that you don't care about the evaluation of \texttt{store-forms} (one of the five values), which allows the possibility of more efficient code. In other words, \texttt{for-effect} is an optimization. \texttt{for-effect} might not work in other implementations of Common Lisp, in particular, it is not implemented for CLOE.

As an example, an extremely simplified version of \texttt{setf}, allowing no more and no fewer than two subforms, containing no optimization to remove unnecessary variables, and not allowing storing of multiple values, could be defined by:

\begin{verbatim}
(defmacro setf (reference value)
  (multiple-value-bind (vars vals stores store-form access-form)
      (get-setf-method reference)
       (declare (ignore access-form))
      '(let* ,(mapcar #'list
                    (append vars stores)
                    (append vals (list value)))
             ,store form)))
\end{verbatim}

For more information, see the macro \texttt{define-setf-method}. 

get-setf-method-multiple-value reference &optional for-effect  Function

Returns five values constituting the setf method for reference, which is a generalized-variable reference. (The five values are described in detail at the end of this discussion.) This is the same as get-setf-method, except that it does not check the number of store-variables (one of the five values). Use get-setf-method-multiple-value in cases that allow storing multiple values into a generalized variable. This is not a common need.

Compatibility Note: The optional argument for-effect is a Symbolics extension to Common Lisp, which might not work in other implementations of Common Lisp.

Here are the five values that express a setf method for a given access form.

- A list of temporary variables.
- A list of value forms (subforms of the given form) to whose values the temporary variables are to be bound.
- A second list of temporary variable, called store variables.
- A storing form.
- An accessing form.

The temporary variables are bound to the value forms as if by let*; that is, the value forms are evaluated in the order given and may refer to the values of earlier value forms by using the corresponding variable.

The store variables are to be bound to the values of the newvalue form, that is, the values to be stored into the generalized variable. In almost all cases, only a single value is stored, and there is only one store variable.

The storing form and the accessing form may contain references to the temporary variables (and also, in the case of the storing form, to the store variables). The accessing form returns the value of the generalized variable. The storing form modifies the value of the generalized variable and guarantees to return the values of the store variables as its values. These are the correct values for setf to return. (Again, in most cases there is a single store variable and thus a single value to be returned.) The value returned by the accessing form is, of course, affected by execution of the storing form, but either of these forms may be evaluated any number of times, and therefore should be free of side effects (other than the storing action of the storing form).

The temporary variables and the store variables are generated names, as if by gensym or gentemp, so that there is never any problem of name clashes among them, or between them and other variables in the program. This is necessary to make the special forms that do more than one setf in parallel work properly. These are psetf, shiftf and rotatef.

Here are some examples of setf methods for particular forms:
• For a variable x:

( )
( )
(g0001)
(setq x g0001)

x

• For (car exp):

(g0002)
(exp)
(g0003)
(progn (rplaca g0002 g0003) g0003)
(car g0002)

• For (supseq seq s e):

(g0004 g0005 g0006)
(seq s e)
(g0007)
(progn (replace g0004 g0007 :start1 g0005 :end1 g0006) g0007)
(subseq g0004 g0005 g0006)

zl:getchar s i

Function
Returns the i (indexth) character of s (string) as a symbol. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (however, aref does not coerce symbols into strings).

Examples:

(zl:getchar "string" 1) => l
(zl:getchar 'symbol 2) => Y
(zl:getchar "STRING" 1) => S
(zl:getchar "ORANGE" 0) => NIL ;1-origin indexing is used

zl:getcharn s i

Function
Returns the i (indexth) character of s (string) as a character. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (however, aref does not coerce symbols or numbers into strings).

Examples:
getf plist indicator &optional default

Stores the property indicator on plist. If indicator is found, the corresponding value is returned. If getf cannot find indicator, default is returned. If default is not specified, nil is used. Note that there is no way to distinguish between a property whose value is default and a missing property.

This function differs from function get in that it takes a place rather than a symbol as its first argument.

(getf (symbol-plist 'some-symbol) 'color) => RED

(getf (symbol-plist 'some-symbol) 'size 'moderate) => MODERATE

(defvar *my-plist* '())
(setf (getf *my-plist* 'mode) 'auto-fill)
*my-plist* => (MODE AUTO-FILL)

(getf *my-plist* 'mode) => AUTO-FILL

See the section "Functions Relating to the Property List of a Symbol".

gethash key table &optional default

Finds the entry in table whose key is key and returns the associated value. If there is no such entry, gethash returns default, which is nil if not specified. It returns three values; the value associated with key, whether or not the key was found (t or nil), and the found key if one exists, or nil if not.

setf is used with gethash to make new entries in the table. If an entry with the specified key exists, it is removed before the new entry is added.

Compatibility Note: Under Genera, gethash is extended to return an extra value: it returns the value of the found key. The reason for this extension is that the :test function, in general, matches non-eq keys with the key stored in the table. In some situations, you might want to know the actual stored key. CLOE returns two values, as specified in CLtL.

(setf (gethash a-key my-table) a-value)

The default argument to gethash can be specified in a very useful way with related functions like incf.

(incf (gethash b-key my-table 0) b-value)
is a shorthand for
(setf (gethash b-key my-table) (+ (gethash b-key my-table) b-value))
For a table of related items: See the section "Table Functions".

**zl:gethash** *key hash-table*  
Function

Finds the entry in *table* whose key is *key* and returns the associated value. This function is obsolete; use **gethash** instead.

**zl:gethash-equal** *key hash-table*  
Function

Finds the entry in *table* whose key is *key* and returns the associated value. This function is obsolete; use **gethash** instead.

**zl:getl** *symbol indicator-list*  
Function

Searches down *symbol* for any of the indicators in *indicator-list* until it finds a property whose indicator is one of the elements of *indicator-list*. **zl:getl** uses the symbol's associated property list. (See the section "Property Lists") **zl:getl** returns the portion of the list inside *symbol* that begin with the first such property it finds. So the car of the returned list is an indicator, and the cdr is the property value. If none of the indicators on *indicator-list* are on the property list, **zl:getl** returns **nil**. For example, if the property list of *foo* were:

```
(bar (1 2 3) baz (3 2 1) color blue height six-two)
```
then:

```
(zl:getl 'foo '(baz height))
=> (baz (3 2 1) color blue height six-two)
```

When more than one of the indicators in *indicator-list* is present in *indicator-list*, which one **zl:getl** returns depends on the order of the properties. This is the only thing that depends on that order.

For a table of related items: See the section "Functions That Operate on Property Lists".

**globalize** *name* &optional *package*  
Function

Establishes a symbol named *name* in *package* and exports it. If this causes any name conflicts with symbols with the same name in packages that use *package*, instead of signalling an error **globalize** makes an attempt to resolve the name conflict automatically and prints an explanation of what is being done on zl:error-output.

**globalize** is useful for patching up an existing package structure. For example, if a new function is added to the Lisp language **globalize** can be used to add its name to the **global** package and hence make it accessible to all packages. Symbols with the desired name might already exist, either by coincidence or because the function was already defined or already called. **globalize** makes all such symbols have the new function as their definition.
package can be a package object or the name of a package, as a symbol or a string. It defaults to the global package. globalize is the only function that does not care whether package is locked.

name can be a symbol or a string. If package already contains a symbol by that name, that symbol is chosen. Otherwise, if name is a symbol, it is chosen. If name is a string and any of the packages that use package contains a nonshadowing symbol by that name, one such symbol is chosen. Otherwise, a new symbol named name is created. Whichever symbol is chosen this way is made present in package and exported from it. If the home package of the chosen symbol is a package that uses package, then the home package is set to package; in other words, the symbol is "promoted" to a "higher" package. If the home package of the chosen symbol is some other package, it is not changed. This case typically occurs when the chosen symbol is inherited by package from some package it uses.

The above rules for choosing a symbol to export ensure that no name conflict occurs if at all possible. If any nonshadowing symbols exist named name but that are distinct from the chosen symbol present in the packages that use package, then a name conflict occurs. globalize does its best to resolve the name conflict by merging together the values, function definitions, and properties of all the symbols involved. After merging, all the symbols have the same value, the same function definition, and the same properties. The value cells, function cells, and property list cells of all the symbols are forwarded to the corresponding cells of the chosen symbol, using sys:dtp-one-q-forward. This ensures that any future change to one of the symbols is reflected by all of the symbols.

The merging operation consists simply of making sure that there are no conflicts. If more than one of the symbols has a value (is boundp), all the values must be eql or an error is signalled. Similarly, all the function definitions of symbols that are fboundp must be eql and all the properties with any particular indicator must be eql. If an error occurs, you must manually resolve it by removing the unwanted value, definition, or property (using makunbound, fmakunbound, or zl:remprop) then try again.

Note that if name is a symbol, globalize attempts to use that symbol, but there is no guarantee that it will not use some other symbol. If name is in a package that does not use package, and globalize does not use name as the symbol (because another symbol by that name already exists in package or in some package that uses package), name is not merged with the chosen symbol. It is generally more predictable to use a string, rather than a symbol, for name.

Of course, globalize cannot cause two distinct symbols to become eq. Its conflict resolution techniques are useful only for symbols that are used as names for things like functions and variables, not for symbols that are used for their own sake. You can sometimes get the desired effect by using one of the conflicting symbols as the first argument to globalize, rather than using a string.

For example, suppose a program in the color package deals with colors by symbolic names, perhaps using zl:selectq to test for such symbols as red, green, and yellow. Suppose there is also a function named red in the math package and someone decides that this function is generally useful and should be made global.
Doing `(globalize 'color:red)` ensures that the exported symbol is the one that the color program is looking for; this means that every package except the `math` package sees the right symbol to use if it wants to call the color program. Programs that call the `red` function do not care which of the two symbols they use as the name of the function, since both symbols have the same definition. Usually the situation described in this example would not arise, because standard programming style dictates that the color program should have been using keywords for this application.

`globalize` returns two values. The first is the chosen symbol and the second is a (possibly empty) list of all the symbols whose value, function, and property cells were forwarded to the cells of the chosen symbol.

To disable the messages printed by `globalize`, bind `zl:error-output` to a null stream (one that throws away all output). For example:

```
(let ((zl:error-output 'si:null-stream))
  (globalize 'rumpelstiltskin))
```

### go tag

Transfers control within a `tagbody` form or a construct like `do` or `prog` that uses an implicit `tagbody`.

The `tag` can be a symbol or an integer. It is not evaluated. `go` transfers control to the tag in the body of the `tagbody` that is `eql` to the `tag` in the `go` form. If the body has no such tag, the bodies of any lexically containing `tagbody` forms are examined as well. If no tag is found, an error is signalled.

The scope of `tag` is lexical. That is, the `go` form must be inside the `tagbody` construct itself (or inside a `tagbody` form that that `tagbody` lexically contains), not inside a function called from the `tagbody`, but defined outside the `tagbody`.

Examples:

```
(tagbody
  (let ((z 5))
    (unwind-protect
      (if (= 5 z) (go out))
      (print z)))
  out
  (princ "4 3 and then there were none") (terpri)) =>
NIL
```
(prog (x y z)
  (setq x some frob)
loop
  do something
  (if some predicate (go endtag))
  do something more
  (if (minusp x) (go loop))
endtag
  (return z))

(let ((i 0)
       (result t))
  (tagbody loop
    (when (and (< i 20) result)
      (unless (= (aref *data-vector* i) i)
        (setq result nil))
      (go loop))))

For a table of related items: See the section "Transfer of Control Functions".

**graphic-char-p** char

Returns t if char does not have any control bits set and is not a format effector.

- (graphic-char-p #\A) => T
- (graphic-char-p #\c-A) => NIL
- (graphic-char-p #\Space) => T

For a table of related items, see the section "Character Predicates".

**zl:greaterp** number &rest more-numbers

In your new programs, we recommend that you use the function >, which is the Common Lisp equivalent of zl:greaterp.

zl:greaterp compares its arguments from left to right. If any argument is not greater than the next, zl:greaterp returns nil. But if the arguments are monotonically strictly decreasing, the result is t. Examples:
- (zl:greaterp 4 3) => t
- (zl:greaterp 4 3 2 1 0) => t
- (zl:greaterp 4 3 1 2 0) => nil

**zl:grind-top-level** exp &optional si:grind-width (si:grind-real-io zl:standard-output)

si:grind-untyo-p (si:grind-displaced 'si:displaced) (terpri-p t) si:grind-notify-fun (loc (ncons exp))

Pretty-prints exp on stream, inserting up to si:grind-width characters per line. This is the primitive interface to the pretty-printer. Note that it does not support vari-
able-width fonts. If the \texttt{si:grind-width} argument is supplied, it is how many charac-
ters wide the output is to be. If \texttt{si:grind-width} is unsupplied or \texttt{nil}, \texttt{zl:grind-top-
level} tries to determine the "natural width" of the stream by sending a \texttt{:size-in-
characters} message to the stream and using the first returned value. If the stream does not handle that message, a width of 95 characters is used instead.

The remaining optional arguments activate various features and usually should not be supplied. These options are for internal use by the system, and are documented here only for completeness. If \texttt{untyo-p} is \texttt{t}, the \texttt{:untyo} and \texttt{:untyo-mark} operations are used on \texttt{stream}, speeding up the algorithm somewhat. \texttt{displaced} controls the checking for displacing macros; it is the symbol that flags a place that has been displaced, or \texttt{nil} to disable the feature. If \texttt{terpri-p} is \texttt{nil}, \texttt{zl:grind-top-level} does not advance to a fresh line before printing.

If \texttt{si:grind-notify-fun} is non-\texttt{nil}, it is a function of three arguments and is called for each "token" in the pretty-printed output. Tokens can be atoms, open and close parentheses, and reader macro characters such as \texttt{'}\texttt{ }. The arguments to \texttt{si:grind-notify-fun} are the token, its "location" (see next paragraph), and \texttt{t} if it is an atom or \texttt{nil} if it is a character.

\texttt{loc} is the "location" (typically a cons) whose \texttt{car} is \texttt{exp}. As the grinder recursively descends through the structure being printed, it keeps track of the location where each thing came from, for the benefit of the \texttt{si:grind-notify-fun}, if any. This makes it possible for a program to correlate the printed output with the list structure. The "location" of a close parenthesis is \texttt{t}, because close parentheses have no asso-
ciated location.

\begin{verbatim}
grindef &rest fns

Prints the definitions of one or more functions, with indentation to make the code readable. Certain other "pretty-printing" transformations are performed:
\begin{itemize}
\item The \texttt{quote} special form is represented with the \texttt{'} character.
\item Displacing macros are printed as the original code rather than the result of macro expansion.
\item The code resulting from the backquote (\texttt{'}) reader macro is represented in terms of \texttt{'}.
\end{itemize}

The subforms to \texttt{grindef} are the function specs whose definitions are to be print-
ed; ordinarily, \texttt{grindef} is used with a form such as \texttt{(grindef foo)} to print the defi-
nition of \texttt{foo}. When one of these subforms is a symbol, if the symbol has a value its value is prettily printed also. Definitions are printed as \texttt{defun} special forms, and values are printed as \texttt{setq} special forms.

If a function is compiled, \texttt{grindef} says so and tries to find its previous interpreted definition by looking on an associated property list. See the function \texttt{uncompile}. This works only if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a binary file, \texttt{grindef} does not find the interpreted definition and cannot do anything useful.
\end{verbatim}
With no subforms, `grind` assumes the same arguments as when it was last called.

**zl:haipart x n**  
*Function*  
Returns the high \( n \) bits of the binary representation of \(|x|\), or the low \(-n\) bits if \( n \) is negative. \( x \) must be an integer; its sign is ignored. `zl:haipart` could have been defined by:

```lisp
(defun zl:haipart (x n)
  (setq x (abs x))
  (if (minusp n)
      (logand x (1- (ash 1 (- n))))
    (ash x (min (- n (zl:haulong x)) 0))))
```

For a table of related items: See the section "Functions Returning Components or Characteristics of Argument".

**:handle-condition cond ignore**  
*Message*  
An interactive handler message to instances of `dbg:basic-handler`. `cond` is a condition object. You should handle this condition, ignoring the second argument. `:handle-condition` can return values or throw in the same way that `condition-bind` handlers can. See the message `:handle-condition-p`.

**:handle-condition-p cond**  
*Message*  
An interactive handler message to Restart Handlers instances of `dbg:basic-handler`. This message examines `cond` which is a condition object. It returns `nil` if declines to handle the condition and something other than `nil` when it is prepared to handle the condition. See the message `:handle-condition`.

**hash-table**  
*Type Specifier*  
`hash-table` is the type specifier symbol for the predefined Lisp data structure of that name.

The types `hash-table`, `readtable`, `package`, `pathname`, `stream` and `random-state` are *pairwise disjoint*.

Examples:
(setq a-hash-table (make-hash-table))
=> #<EQL-BLOCK-ARRAY-PROCESS-LOCKING-DUMMY
- GC-LOCKING-ASSOCIATION-MUTATING-TABLE 16126776>
(setf (gethash 'color a-hash-table) 'red) => RED
(setf (gethash 'name a-hash-table) 'Ron) => RON
(typep 'hash-table 'common) => T
(subtypep 'hash-table 't) => T and T
(sys:type-arglist 'hash-table) => NIL and T
(hash-table-p a-hash-table) => T

See the section "Data Types and Type Specifiers". See the section "Table Management".

**hash-table-count** *table*  \hspace{1cm} Function

Returns the number of entries in *table*. When a table is first created or has been cleared, the number of entries is zero.

\[
\text{(hash-table-count (setq new-hash-table (make-hash-table :size 5000)))}
\]
\[
=> 0
\]

For a table of related items: See the section "Table Functions".

**hash-table-p** *object*  \hspace{1cm} Function

Returns true if and only if *object* is a hash table. Under Genera, **hash-table-p** returns t for old Zetalisp hash tables also.

\[
\text{(hash-table-p (make-hash-table))} => T
\]

For a table of related items: See the section "Table Functions".

**zl:haulong** \hspace{1cm} Function

Returns the number of significant bits in |*x|*. *x* must be an integer. Its sign is ignored. The result is the least integer strictly greater than the base-2 logarithm of |*x|.

**zl:haulong** is similar to **integer-length**.

Examples:

\[
\text{(zl:haulong 0)} => 0
\]
\[
\text{(zl:haulong 3)} => 2
\]
\[
\text{(zl:haulong -7)} => 3
\]

For a table of related items: See the section "Functions Returning Components or Characteristics of Argument".

**:home-cursorpos**  \hspace{1cm} Message
This operation is supported by the same streams that support :read-cursorpos. It sets the position of the cursor. It puts the cursor back at the beginning of the stream. For window streams, it puts the cursor at the upper left edge of the window.

**zl:hostat &rest hosts**

*Function*

Asks each of the hosts for its status, and prints the results. If no hosts are specified, asks all hosts on the Chaosnet. Hosts can be specified by either name or octal number.

For each host, a line is displayed that either says that the host is not responding or gives metering information for the host’s network attachments. If a host is not responding, probably it is down or there is no such host at that address. A Symbolics host can fail to respond if it is looping inside `without-interrupts` or paging extremely heavily, such that it is simply unable to respond within a reasonable amount of time.

See the section “Show Hosts Command”.

To abort the host status report produced by *zl:hostat* or FUNCTION H, press `c-`ABORT`.

**zl:ibase**

*Variable*

In your new programs, we recommend that you use the variable `*read-base*`, which is the Common Lisp equivalent of `zl:ibase`.

The value of `zl:ibase` is a number that is the radix in which integers and ratios are read. The initial value of `zl:ibase` is 10. `zl:ibase` should not be greater than 36.

When `zl:ibase` is set to a value greater than ten, the reader interprets the token as a symbol, unless control variable `si:*read-extended-ibase-signed-number*` or `si:*read-extended-ibase-unsigned-number*` is set to `t`.

**identity** object

*Function*

Always returns `object` as its value. Sometimes functions require a second function as an argument, and `identity` is useful in those situations.

**if condition true &rest false**

*Special Form*

The simplest conditional form. The “if-then” form looks like:

```
(if predicate-form then-form)
```

`predicate-form` is evaluated, and if the result is `non-nil`, the `then-form` is evaluated and its result is returned. Otherwise, `nil` is returned.
Examples:

(if (numberp 'a) "never reaches this point") => NIL

(if (not nil) "A Word") => "A Word"

(if 'not-nil "reaches this point") => "reaches this point"

In the "if-then-else" form, it looks like:

(if predicate-form then-form else-form)

_predicate-form_ is evaluated, and if the result is non-nil, the _then-form_ is evaluated and its result is returned. Otherwise, the _else-form_ is evaluated and its result is returned.

Examples:

(if (equal 'boy 'girl) "same" "different") => "different"

(if (not nil) 'A 'B) => A

(if 'word "reaches this point" "never reaches this point")
  => "reaches this point"

(defun make-even (integer)
  (if (oddp integer) (+ integer 1) integer))

(make-even 5) => 6
(make-even 2) => 2

**Common Lisp Compatibility Note:** The Symbolics Common Lisp version of _if_ is extended to allow you to supply more than three arguments; the _CLtL_ version requires two or three arguments, and signals an error if additional arguments are supplied.

**Zetalisp Note:** Zetalisp supports multiple _else_ clauses: if there are more than three subforms, _if_ assumes you want more than one _else-form_; these are evaluated sequentially and the result of the last one is returned, if the predicate returns _nil_.

**CLOE Note:** In CLOE, _if_ signals a warning if you use multiple else forms. Multiple _else_ clauses are incompatible with the language specification presented in Guy Steele’s _Common Lisp: the Language_.

For a table of related items: See the section "Conditional Functions".

**if keyword for loop**

if _expr_

If _expr_ evaluates to _nil_, the following clause is skipped, otherwise not.

Examples
(defun print-odd (list-of-nums)
  (loop for num in list-of-nums
    if (oddp num)
      collect num and do (print num)) => PRINT-ODD
  (print-odd '(2 3 49 2 3 4)) =>
  3
  49
  3 (3 49 3)

if-then-else conditionals can be written using the \texttt{else} keyword, as in:

(defun print-odd-else (list-of-nums)
  (loop for num in list-of-nums
    if (oddp num)
      collect num and do (print num)
    else
      do (print "An even number !")) => PRINT-ODD-ELSE
  (print-odd-else '(4 3 2 9 7)) =>
  "An even number !"
  3
  "An even number !"
  9
  7 (3 9 7)

Multiple clauses can appear in an \texttt{else}-phrase using \texttt{and} to join them.

In the typical format of a conditionalized clause such as

\begin{verbatim}
  when expr1 keyword expr2
\end{verbatim}

\texttt{expr2} can be the keyword \texttt{it}. If that is the case, a variable is generated to hold the value of \texttt{expr1}, and that variable gets substituted for \texttt{expr2}. Thus, the composition:

\begin{verbatim}
  when expr return it
\end{verbatim}

is equivalent to the clause:

\begin{verbatim}
  thereis expr
\end{verbatim}

and you can collect all non-null values in an iteration by saying:

\begin{verbatim}
  when expression collect it
\end{verbatim}

If multiple clauses are joined with \texttt{and}, the \texttt{it} keyword can only be used in the first. If multiple \texttt{whens}, \texttt{unless}s, and/or \texttt{ifs} occur in sequence, the value substituted for \texttt{it} is that of the last test performed. The \texttt{it} keyword is not recognized in an \texttt{else}-phrase.

Conditionals can be nested.

See the section "\texttt{loop} Conditionalization".

\texttt{ignore} var1 var2 ...

\texttt{Declaration}
Specifies that bindings of the \textit{vars} are never used.
See the section "Declaration Specifiers".

\textbf{ignore \&rest ignore \textit{Function}}

Takes any number of arguments and returns \texttt{nil}. This is often useful as a "dummy" function; if you are calling a function that takes a function as an argument, and you want to pass one that does not do anything and does not mind being called with any argument pattern, use this.

\texttt{ignore} is also used to suppress compiler warnings for ignored arguments. For example:

\begin{verbatim}
(defun foo (x y)
  (ignore y)
  (sin x))
\end{verbatim}

See the section "Functions and Special Forms for Constant Values".

\textbf{ignore-errors \&body body \textit{Special Form}}

Sets up a very simple handler on the bound handlers list that handles all error conditions. Normally, it executes \texttt{body} and returns the first value of the last form in \texttt{body} as its first value and \texttt{nil} as its second value. If an error signal occurs while \texttt{body} is executing, \texttt{ignore-errors} immediately returns with \texttt{nil} as its first value and something not \texttt{nil} as its second value.

\texttt{ignore-errors} replaces \texttt{zl:errset} and \texttt{catch-error}.

For a table of related items, see the section "Basic Forms for Bound Handlers".

\textbf{imagpart number \textit{Function}}

If \texttt{number} is a complex number, \texttt{imagpart} returns the imaginary part of \texttt{number}.
If \texttt{number} is a noncomplex number, \texttt{imagpart} returns a zero of the same type as \texttt{number}.

Examples:

\begin{verbatim}
(imagpart #c(3 4)) => 4
(imagpart 4) => 0
\end{verbatim}

Related Functions:

\texttt{complex}
\texttt{realpart}

For a table of related items: See the section "Functions that Decompose and Construct Complex Numbers".

\textbf{zl:implode x \textit{Function}}
Similar to \texttt{zl:maknam}, except that the returned symbol is interned in the current package. This function is provided mainly for Maclisp compatibility.

Example:

\begin{verbatim}
(zl:implode '(a #\b "C" #\4 5)) => |AbC4-
\end{verbatim}

\textbf{import} \texttt{symbols} \&optional \texttt{package} \hspace{1cm} \textbf{Function}

\textit{symbols} should be a list of symbols or a single symbol. If \textit{symbols} is \texttt{nil}, it is treated like an empty list. These symbols become internal symbols in \textit{package}, and can therefore be referred to without a colon qualifier. \textit{import} signals a correctable error if any of the imported symbols has the same name as some distinct symbol already available in the package.

\begin{verbatim}
=> *package*
TURBINE-PACKAGE
=> (export valve-pressure)
T
=> (import generator:valve-pressure)
ERROR: GENERATOR:VALVE-PRESSURE WILL SHADOW VALVE-PRESSURE
\end{verbatim}

\textit{package} can be a package object or the name of a package (a symbol or a string). If unspecified, \textit{package} defaults to the value of \texttt{*package*}. Returns \texttt{t}.

The following code makes all the external symbols of the turbine-package accessible in the generator-package.

\begin{verbatim}
(do-external-symbols (symbol 'turbine-package)
  (import symbol 'generator-package))
\end{verbatim}

Of course, the following call to \texttt{use-package} inside of \texttt{generator-package} would accomplish the same thing:

\begin{verbatim}
(use-package 'turbine-package)
\end{verbatim}

\textbf{in-package} \texttt{package-name} \&\textbf{rest} \texttt{make-package-keywords} \hspace{1cm} \textbf{Function}

Intended to be placed at the start of a file containing a subsystem that is to be loaded into some package other than \texttt{user}. If there is not already a package named \textit{package-name}, this function acts like \texttt{make-package}, except that after the new package is created, \texttt{*package*} is set to it. This binding remains until changed by the user, or until the \texttt{*package*} variable reverts to its old value at the completion of a \texttt{load} operation.

If there is a package named \textit{package-name}, the assumption is that the user is reloading a file after making some changes. The existing package is augmented to reflect any new nicknames or new packages in the :\texttt{use} list, and \texttt{*package*} is then set to this package.

\textbf{incf} \texttt{access-form} \&optional \texttt{amount} \hspace{1cm} \textbf{Macro}
Increments the value of a generalized variable. \texttt{(incf ref)} increments the value of \texttt{ref} by 1. \texttt{(incf ref amount)} adds \texttt{amount} to \texttt{ref} and stores the sum back into \texttt{ref}. It returns the new value of \texttt{ref}.

\textit{access-form} can be any form acceptable to \texttt{setf}.

\begin{verbatim}
  (incf (car (mumble)))
\end{verbatim}

is almost equivalent to

\begin{verbatim}
  (setf (car (mumble)) (1+ (car (mumble))))
\end{verbatim}

except that while the latter would evaluate \texttt{mumble} twice, \texttt{incf} actually expands into a \texttt{let} and \texttt{mumble} is evaluated only once.

\begin{verbatim}
  (setq arr (make-array (4) :element-type 'integer
          :initial-element 5))
\end{verbatim}

\begin{verbatim}
  (incf (aref arr 3) 4) => #(5 5 5 9)
\end{verbatim}

See the section "Generalized Variables".

\texttt{:increment-cursorpos x y &optional (units ':pixel)} \hspace{1cm} \textit{Message}

This operation is supported by the same streams that support \texttt{:read-cursorpos}. It sets the position of the cursor. \texttt{x} and \texttt{y} are the amounts to increment the current \texttt{x} and \texttt{y} coordinates. \texttt{units} is the same as the \texttt{units} argument to \texttt{:read-cursorpos}.

\texttt{:info} \hspace{1cm} \textit{Message}

Returns a cons of the truename and creation date of the file. The creation date is a number that is a universal time. This can be used to tell if the file has been modified between two \texttt{open}s. For an output stream the info is not meaningful until after the stream has been closed, at least on an ITS file server.

\texttt{sys:inhibit-fdefine-warnings} \hspace{1cm} \textit{Variable}

Controls printing of warnings when functions are redefined. This variable is normally \texttt{nil}. Setting it to \texttt{t} prevents \texttt{fdefine} from warning you and asking about questionable function definitions such as a function being redefined by a different file than defined it originally, or a symbol that belongs to one package being defined by a file that belongs to a different package. Setting it to \texttt{:just-warn} allows the warnings to be printed out, but prevents the queries from happening; it assumes that your answer is "yes", that is, that it is all right to redefine the function.

\textbf{Note}: The preferred way of associating the definition of a function with its source file is by using \texttt{record-source-file-name}: See the function \texttt{record-source-file-name}.

\texttt{clos:initialize-instance instance &rest initargs} \hspace{1cm} \textit{Generic Function}
Calls `clos:shared-initialize` to initialize the instance, and returns the initialized instance. This generic function is intended to be specialized by programmers, but not to be called directly. It is called by `clos:make-instance`.

`instance` The instance to initialize.

`initargs` Alternating initialization argument names and values.

The default primary method for `clos:initialize-instance` calls the `clos:shared-initialize` generic function with the instance, `t`, and the initialization arguments provided to `clos:initialize-instance`.

Note that the usual way for users to customize the initialization behavior is to specialize `clos:initialize-instance` by writing after-methods. Any applicable after-methods for `clos:initialize-instance` are called after the primary method for `clos:initialize-instance`. A user-defined primary method would override the default method, and thus could prevent the usual slot-filling behavior.

### `si:initial-readtable` Variable

The value of `si:initial-readtable` is the initial standard readtable. You should never change the contents of either this readtable or `si:initial-readtable`; only examine it, by using it as the `from-readtable` argument to `zl:copy-readtable` or `zl:set-syntax-from-char`. Change `zl:readtable` instead.

### `dbg:initialize-special-commands` condition Generic Function

The Debugger calls `dbg:initialize-special-commands` after it prints the error message. The methods are combined with `:progn` combination, so that each one can do some initialization. In particular, the methods for this generic function can remove items from the list `dbg:special-commands` in order to decide not to offer these special commands.

The compatible message for `dbg:initialize-special-commands` is:

`:initialize-special-commands`

For a table of related items: See the section "Debugger Special Command Functions".

### `initially` keyword for loop

`initially expression`

Puts `expression` into the prologue of the iteration. It is evaluated before any other initialization code other than the initial bindings. For the sake of good style, the `initially` clause should therefore be placed after any `with` clauses but before the main body of the loop.

Examples
(defun sum-it (limit)
  (loop with sum-of-series = 0
    initially (print "The sum of this series is :")
    for num from 0 to limit
  do
    (setq sum-of-series (+ sum-of-series num))
  finally (prin1 sum-of-series))) => SUM-IT

(sum-it 9) =>
"The sum of this series is :" 45
NIL

See the macro loop.

inline

(inline function1 function2 ... ) specifies that it is desirable for the compiler to
open-code calls to the specified functions; that is, the code for a specified function
should be integrated into the calling routine, appearing "in line" in place of a pro-
cEDURE call. This may achieve extra speed at the expense of debuggability (calls to
functions compiled in-line cannot be traced, for example). This declaration is perv-
asive, that is it affects all code in the body of the form. The compiler is free to
ignore this declaration.

Note that rules of lexical scoping are observed; if one of the functions mentioned
has a lexically apparent local definition (as made by flet or labels), the declaration
applies to that local definition and not to the global function definition.

See the section "Declaration Specifiers".

:input-editor function &rest arguments

This is supported by interactive streams such as windows. It is described in its
own section (see the section "The Input Editor Program Interface").

Most programs should not send this message directly. See the function with-input-
editing.

input-stream-p stream

Returns t if stream can handle input operations, otherwise returns nil.

(streamp *standard-input*) => T
(setq file-stream
  (open "foo" :direction :output :element-type 'character))

(input-stream-p file-stream) => NIL

:input-wait &optional whostate function &rest arguments

Message
This message to an input stream causes the stream to **process-wait** with *whostate* until either of the following conditions is met:

- Applying *function* to arguments returns non-*nil*.
- The stream enters a state in which sending it a :**tyi** message would immediately return a value or signal an error.

When either of these conditions is met, **:input-wait** returns. If the stream enters a state in which sending it a :**tyi** message would signal an error, **:input-wait** returns instead of signalling the error. The returned value is not defined.

*whostate* is what to display in the status line while process-waiting. It can be a string or *nil*. A value of *nil* means to use the normal whostate for this stream, such as "Tyi", "Net In", or "Serial In". For interactive streams, the default whostate is "Tyi".

*function* can be a function or *nil*. A value of *nil* means that the stream just waits until sending it a :**tyi** message would immediately return a value or signal an error.

This message is intended for programs that need to wait until either input is available from some interactive stream or some other condition, such as the arrival of a notification, occurs. Any stream that can become the value of **zl:terminal-io** must support **:input-wait**.

Following is a simple example of the use of **:input-wait** to wait for input or a notification to an interactive stream. The function just displays notifications and prints representations of characters or blips received as input.

```lisp
(defun my-top-level (stream)
  (error-restart-loop ((error sys:abort) "My top level")
    (send stream :input-wait nil
      #'(lambda (note-cell)
          (not (null (location-contents note-cell)))
          (send stream :notification-cell))
      (let ((note (send stream :receive-notification)))
        (if note
            (sys:display-notification stream note :stream)
            (let ((char (send stream :any-tyi-no-hang)))
              (cond ((null char))
                  ((characterp char)
                    (format stream "\~&Character: \"C\" char))
                  ((listp char)
                    (format stream "\~&Blip: \"S\" char))
                  (t (format stream "\~&Unknown object: \"S\" char))))))))
```

**Method**

**Inserts method :insert si:heap item key**

Inserts *item* into the heap based on *key*, and returns *item* and *key*. 

---

(continued)
For a table of related items: See the section "Heap Functions and Methods".

**inspect** &optional object

Function

A window-oriented version of **describe**.

Note: While the Symbolics Common Lisp version of this function does not require the argument **object**, the function as specified in *Common LISP: The Language* does. See the section "How the Inspector Works".

**instance** &optional (flavor '*)

Type Specifier

Denotes flavor instances. When a new flavor is defined with **defflavor**, the name of the flavor becomes a valid type symbol, and individual instances of that flavor become valid types of **instance** that can be tested with **typep**.

**instance** is a subtype of **t**.

Examples:

```lisp
(defflavor ship
  (name x-velocity y-velocity z-velocity mass)
  () ; no component flavors
  :readable-instance-variables
  :writable-instance-variables
  :initial-instance-variables) => SHIP

(setq my-ship
  (make-instance 'ship :name "Enterprise"
    :mass 4534
    :x-velocity 24
    :y-velocity 2
    :z-velocity 45)) => #<SHIP 43100701>

(ship-name my-ship) => "Enterprise"

(typep my-ship 'instance) => T

(typep my-ship '(instance ship)) => T

(zl:typep my-ship) => SHIP

(type-of my-ship) => SHIP

(type-of 'ship) => SYMBOL

(sys:type-arglist 'instance) => (&OPTIONAL (FLAVOR '*)) and T
```

See the section "Data Types and Type Specifiers".
For a discussion of flavors: See the section "Flavors".

**instancep** `object`  
*Function*

Returns `t` if the `object` is a flavor instance, otherwise `nil`.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**int-char** `integer`  
*Function*

Accepts a non-negative integer argument and returns a character if `integer` is in the range of `char-int`. Especially useful for converting an integer returned by a call to `char-int` back into a character.

```
(int-char 65) => #\A
```

```
(defun char-arr (make-array 512))
(setq (elt char-arr (char-int #\a)) 'first)
```

```
(dotimes (i 512)
  (if (eq (elt char-arr i) 'first)
    (return (int-char i))))
```

In the current Unix implementation of CLOE, integer arguments in the range of 1 to 4096 return unique character objects. A larger integer argument returns one of the characters returned by an argument less than 4096. Arguments above 4096 return `#\undefined-lozenge` as defined under Genera.

For information on characters, see the section "The Character Set".

For a table of related items, see the section "Character Conversions".

**integer** &optional (`low` `*`) (`high` `*`)  
*Type Specifier*

`integer` is the type specifier symbol for the predefined Lisp integer number type.

The types `integer` and `ratio` are an exhaustive partition of the type `rational`, since `rational` ≡ `(or integer ratio)`.

This type specifier can be used in either symbol or list form. Used in list form, `integer` allows the declaration and creation of specialized integer numbers, whose range is restricted to `low` and `high`.

`low` and `high` must each be an integer, a list of an integer, or unspecified. If these limits are expressed as integers, they are inclusive; if they are expressed as a list of an integer, they are exclusive; `*` means that a limit does not exist, and so effectively denotes minus or plus infinity, respectively.

The type `fixnum` is simply a name for `(integer smallest largest)` for the values of `most-negative-fixnum` and `most-positive-fixnum`. The type `(integer 0 1)` is so useful that it has the special name `bit`.
Examples:

(typep 4 'integer) => T
(subtypep 'integer 'rational) => T and T ;subtype and certain
(subtypep '(integer *) 'rational) => T and T
(subtypep 'signed-byte 'integer) => T and T
(subtypep 'fixnum 'integer) => T and T
(subtypep 'bignum 'integer) => T and T
(commonp 23.) => T
(integerp 23.) => T
(integerp -3_78) => T
(integerp most-positive-fixnum) => T
(integerp most-negative-fixnum) => T
(integerp -2147483648) => T
(equal-typep 'bit '(integer 0 1)) => T
(equal-typep '(integer -2147483648 2147483647) 'fixnum) => T
(sys:type-arglist 'integer) => (&OPTIONAL (LOW '*)) and T

See the section "Data Types and Type Specifiers".
See the section "Numbers".

integer-decode-float float

Function

Returns three values, representing: the significand (scaled so as to be an integer),
the exponent, and the sign of the floating-point argument, float, as described below. Scaling the significand essentially means interpreting the bit field of the mantissa as an integer.

For an argument f, the first result is an integer which is strictly less than (expt 2 (float-precision f)), but no less than (expt 2 (- (float-precision f) 1)) except that if f is zero, the returned integer value is zero.

The second value returned is an integer e such that the first result (the significand) times 2 raised to the power e is equal to the absolute value of the argument float.

The final value of integer-decode-float represents the sign of float and is 1 or -1.

Examples:

(integer-decode-float 2.0) => 8388608 and -22 and 1
(integer-decode-float -2.0) => 8388608 and -22 and -1
(integer-decode-float 4.0) => 8388608 and -21 and 1
(integer-decode-float 8.0) => 8388608 and -20 and 1
(integer-decode-float 3.0) => 12582912 and -22 and 1

The exact values produced by the following functions serve illustrative purposes, and might vary between CLOE implementations or within an implementation over time.
For a table of related items, see the section "Functions that Decompose and Construct Floating-point Numbers".

**integer-length integer**  
*Function*

Returns the result of the following computation:
\[
(values (ceiling (log (if (minusp integer) (- integer)(1+ integer)) 2))))
\]

If `integer` is non-negative, the result represents the number of significant bits in the unsigned binary representation of `integer`. More generally, regardless of the sign of `integer`, the result denotes the number of significant bits needed to represent `integer` in unsigned binary two's-complement form. (To get the number of bits needed for a signed binary two's complement representation, add 1 bit to the result of `integer-length`).

Examples:

\[
\begin{align*}
(integer\text{-}length\ 0) \Rightarrow 0 & \quad (integer\text{-}length\ -0) \Rightarrow 0 \\
(integer\text{-}length\ 1) \Rightarrow 1 & \quad (integer\text{-}length\ -1) \Rightarrow 0 \\
(integer\text{-}length\ 2) \Rightarrow 2 & \quad (integer\text{-}length\ -2) \Rightarrow 1 \\
(integer\text{-}length\ 8) \Rightarrow 4 & \quad (integer\text{-}length\ -8) \Rightarrow 3 \\
(integer\text{-}length\ 15) \Rightarrow 4 & \quad (integer\text{-}length\ -15) \Rightarrow 4
\end{align*}
\]

;;; A possible use of integer-length
;;; The function trailing-zeros returns the number of
;;; consecutive zeros starting at the least significant
;;; bit of the binary representation of an integer

(defun trailing-zeros (integer)
  (1- (integer\text{-}length (logand integer (- integer))))))

(trailing-zeros 0) \Rightarrow -1
;;; An adequate result since there are an undefined amount
;;; of trailing zeros in 0
(trailing-zeros 1) \Rightarrow 0
(trailing-zeros 4) \Rightarrow 2 ; 4 is \#b100
(trailing-zeros 9) \Rightarrow 0 ; 9 is \#b1001

For a table of related items, see the section "Functions Returning Components or Characteristics of Argument". 

\[
(*\ 0.5625\ (expt\ 2\ 3)) \Rightarrow 4.5
\]
**integerp** object

This predicate returns **t** if its argument is an integer; otherwise it returns **nil**.

Examples:

```lisp
(integerp 7) => T
(integerp 4.0) => NIL
(integerp #c(2 0)) => T ;#c(2 0) is coerced to an integer
(integerp "not a number") => NIL
```

The following code tests whether `a` and `b` are numbers. If they are numbers, they are added. Otherwise, we attempt to extract integers that are then tested by `integerp`:

```lisp
(if (and (numberp a) (numberp b))
  (+ a b)
(if (and (consp a)
  (integerp (car a))
  (consp b)
  (integerp (car b)))
  (+ (car a) (car b))
(error "couldn't extract integers from ~a and ~a" a b))
```

For a table of related items, see the section "Numeric Type-checking Predicates".

**:interactive**

Returns **t** if the stream is interactive and **nil** if it is not. Interactive streams, built on `si:interactive-stream`, are streams designed for interaction with human users. They support input editing. Use the **:interactive** message to find out whether a stream supports the **:input-editor** message.

**intern** string &optional (pkg *package*)

Finds or creates a symbol named `string` in `pkg`. Inherited symbols in `pkg` are included in the search for a symbol named `string`. If a symbol named `string` is found, it is returned. If no such symbol is found, one is created and installed in `pkg` as an internal symbol (if `pkg` is the **keyword** package, the symbol is installed as an external symbol).

`intern` returns two values. The first is the symbol that was found or created. The second value is **nil** for newly created symbols. If the symbol returned is a pre-existing symbol, this second value is one of the following:

- **:internal** The symbol is present in `pkg` as an internal symbol.
- **:external** The symbol is present in `pkg` as an external symbol.
- **:inherited** The symbol is an internal symbol in `pkg` inherited by way of `use-package`.

`intern` is sensitive to case and under Genera, style. If a string contains character
styles, use the function `string-thin` on its arguments. See the function `string-thin`. The following code uses `intern` with `multiple-value-bind` to capture both returned values. If the `status` of the interned symbol is `:internal`, then the symbols is exported.

```
(multiple-value-bind (symbol status) (intern new-symbol)
  (when (eq status ':internal)
    (export symbol)))
```

For more information: See the section "Mapping Names to Symbols".

### intern

`zl:intern sym &optional pkg`

Finds or creates a symbol named `sym` accessible to the package `pkg`, either directly present in `pkg` or inherited from a package it uses.

See the function `intern`.

### intern-local

`intern-local string &optional pkg`

Finds or creates a symbol named `string` directly present in `pkg`. Symbols inherited by `pkg` from packages it uses are not considered, thus `intern-local` can cause a name conflict. `intern-local` is considered to be a low-level primitive, and indiscriminate use of it can cause undetected name conflicts. Use `import`, `shadow`, or `shadowing-import` for normal purposes.

If `string` is not a string but a symbol, and no symbol with that print name is already interned in `pkg`, `intern-local` interns `string` — rather than a newly created symbol — in `pkg` (even if it is also interned in some other package) and returns it.

For more information: See the section "Mapping Names to Symbols".

### intern-local-soft

`intern-local-soft string &optional pkg`

Find a symbol named `string` directly present in `pkg`. Symbols inherited by `pkg` from packages it uses are not considered. If no symbol is found, the two values `nil nil` are returned.

`intern-local-soft` is a good low-level primitive for when you want complete control of what packages to search and when to add new symbols.

For more information: See the section "Mapping Names to Symbols".

### intern-soft

`intern-soft string &optional pkg`

Finds a symbol named `string` accessible to `pkg`, either directly present in `pkg` or inherited from a package it uses. If no symbol is found, the two values `nil nil` are returned.

### intersection

`intersection list1 list2 &key (test #'eql) test-not (key #'identity)`

Function

```
Returns a new list containing everything that is an element of both list1 and list2, as checked by the :test and :test-not keywords. If either list has duplicate entries, the redundant entries may or may not appear in the result. For example:

\[
\text{(intersection '(a b c) '(f a d)) => (A)}
\]

\[
\text{(intersection '(a b c a d) '(f a d)) => (A A D)}
\]

\[
\text{(intersection '(a b c) '(a f a d)) => (A)}
\]

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way.

:test Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

For all possible ordered pairs consisting of one element from list1 and one element from list2, the test is used to determine whether they match. For every matching pair, the element from list1 is put in the result.

In the following example, intersection finds the new tenured professor:

\[
\begin{align*}
\text{(setq professors-with-tenure} &
\text{'}(("Jones" CS101 CS242)("smith" CS202 CS231)} \\
\text{("parks" CS221)("hunter" CS216 CS232)))} \\
\text{(setq new-professors} &
\text{'}(("Able" CS101 CS244)("Cain" CS101 CS331)} \\
\text{("Parks" CS221)("adams" CS215 CS222)))} \\
\text{(intersection professors-with-tenure new-professors} &
\text{:test #'string-equal :key #'car)} \\
\Rightarrow \\
\text{(('parks" CS221))}
\end{align*}
\]

For a table of related items: See the section "Functions for Comparing Lists".

zl:intersection &rest lists Function

Takes any number of lists that represent sets and returns a new list that represents the intersection of all the sets it is given. zl:intersection uses eq for its comparisons. You cannot change the function used for the comparison. If no arguments are supplied, (zl:intersection) returns nil.

For a table of related items: See the section "Functions for Comparing Lists".
clos:invalid-method-error  method  format-string  &rest  args  

Within method combination, signals an error when the method qualifiers of an applicable method are not valid for the method-combination type; it should be called only within the dynamic extent of a method-combination function.

clos:invalid-method-error  is called automatically when a method fails to satisfy any qualifier pattern or predicate in a clos:define-method-combination-type form. A method-combination function that imposes additional restrictions should call clos:invalid-method-error explicitly if it encounters an invalid method.

method  The method object that is invalid.
format-string  A control string that can be given to format.
args  Arguments required by the format-string.

math:invert-matrix  matrix  &optional  into-matrix  

Computes the inverse of matrix. If into-matrix is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. matrix must be two-dimensional and square. The Gauss-Jordan algorithm with partial pivoting is used. Note: If you want to solve a set of simultaneous equations, you should not use this function; use math:decompose and math:solve.

math:invert-matrix does not work on conformally displaced arrays.

dbg:invoke-restart-handlers  condition  &key  (flavors  nil  flavors-specified)  

Searches the list of restart handlers to find a restart handler for condition. The flavors argument controls which restart handlers are examined. flavors is a list of condition names. When flavors is omitted, the function examines every restart handler. When flavors is provided, the function examines only those restart handlers that handle at least one of the conditions on the list.

The first restart handler that it finds to handle the condition is invoked and given condition. It returns nil if no appropriate restart handler is found.

isqrt  integer  

Integer square root. integer must be a non-negative integer; the result is the greatest integer less than or equal to the exact square root of integer.

Examples:

(isqrt 4) => 2
(isqrt 5) => 2
(isqrt 8) => 2
(isqrt 9) => 3
(isqrt 81) => 9
(isqrt 42) => 6
For a table of related items: See the section "Arithmetic Functions".

&key

Lambda List Keyword

If the lambda-list keyword &key is present, all specifiers up to the next lambda-list keyword, or the end of the list, are keyword parameter specifiers. The keyword parameter specifiers can be followed by the lambda-list keyword &allow-other-keys, if desired.

keyword

Type Specifier

keyword is the type specifier symbol for the predefined Lisp object of that name.

Examples:

```lisp
(typedp ':list 'keyword) => T
(subtypedp 'keyword 't) => T and T
(subtypedp 'keyword 'common) => NIL and NIL
(sys:typed-arglist 'keyword) => NIL and T
(keyworpd ':fixnum) => T
```

See the section "Data Types and Type Specifiers".

See the section "Symbols, Keywords, and Variables".

zl:keyword-extract keylist keyvar keywords &optional flags &body otherwise

Special Form

Aids in writing functions that take keyword arguments in the standard fashion. You can also use the &key lambda-list keyword to create functions that take keyword arguments. &key is preferred and is substantially more efficient; zl:keyword-extract is obsolete. See the section "Evaluating a Function Form".

The form:

```
(zl:keyword-extract key-list iteration-var
  keywords flags other-clauses ...)
```

parses the keywords out into local variables of the function. key-list is a form that evaluates to the list of keyword arguments; it is generally the function’s &rest argument. iteration-var is a variable used to iterate over the list; sometimes other-clauses uses the form:

```
(car (setq iteration-var (cdr iteration-var)))
```

to extract the next element of the list. (Note that this is not the same as pop, because it does the car after the cdr, not before.)

keywords defines the symbols that are keywords to be followed by an argument. Each element of keywords is either the name of a local variable that receives the argument and is also the keyword, or a list of the keyword and the variable, for use when they are different or the keyword is not to go in the keyword package.
Thus, if `keywords` is `(a (b c) d)` the keywords recognized are `a`, `b`, and `d`. If `a` is specified, its argument is stored into `a`. If `d` is specified, its argument is stored into `d`. If `b` is specified, its argument is stored into `c`.

Note that `zl:keyword-extract` does not bind these local variables; it assumes you have done that somewhere else in the code that contains the `zl:keyword-extract` form.

`flags` defines the symbols that are keywords not followed by an argument. If a flag is seen its corresponding variable is set to `t`. (You are assumed to have initialized it to `nil` when you bound it with `let` or `&aux`.) As in `keywords`, an element of `flags` can be either a variable from which the keyword is deduced, or a list of the keyword and the variable.

If there are any `other-clauses`, they are `zl:selectq` clauses selecting on the keyword being processed. These clauses are for handling any keywords that are not handled by the `keywords` and `flags` elements. These can be used to do special processing of certain keywords for which simply storing the argument into a variable is not good enough. Unless the `other-clauses` include an `otherwise` (or `t`) clause after them, there is an `otherwise` clause to complain about any unhandled keywords found in `key-list`. If you write your own `otherwise` clause, it is up to you to take care of any unhandled keywords.

For a table of related items, see the section "Iteration Functions".

**keywordp object**

A predicate that is true if `object` is a symbol and its home package is the keyword package, and false otherwise.

```
(keywordp 'key) => NIL
(keywordp ':key) => T
```

See the section "The Package Cell of a Symbol".

**labels functions &body body**

Identical to `flet` in structure and purpose, but has slightly different scoping rules. It, too, defines one or more functions whose names are made available within its body. In `labels`, unlike `flet`, however, the functions being defined can refer to each other mutually, and to themselves, recursively. Any of the functions defined by a single use of `labels` can call itself or any other; there is no order dependence. Although `flet` is analogous to `let` in its parallel binding, `labels` is not analogous to `let*`.

`labels` is in all other ways identical to `flet`. It defines internal functions that can be called, redefined, passed as funargs, and so on.

Functions defined by `labels`, when passed as funargs, generate closures. The allocation of these closures, that is, whether they appear on the stack or in the heap, is controlled in the same way as for internal lambdas. See the section "Funargs and Lexical Closure Allocation".
Here is an example of the use of labels:

(defun combinations (total-things at-a-time)
  ;; This function computes the number of combinations of
  ;; total-things things taken at-a-time at a time.
  ;; There are more efficient ways, but this is illustrative.
  (labels ((factorial (x)
               (permutations x x))
             (permutations (x n) ; x things n at a time
               (if (= n 1)
                x
                (* x (permutations (1- x) (1- n))))))
    (/ (permutations total-things at-a-time)
       (factorial at-a-time))))

In the following example, we use labels to locally define a function that calls itself. If we instead use flet, an error will result because the call to my-adder in the body would refer to an outer (presumably non-existent) my-adder instead of the local one.

(defun example-labels (operand-a operand-b)
  (labels ((my-adder (accumulator counter)
              (if (= counter 0)
               accumulator
               (my-adder (incf accumulator) (decf counter))))
           (my-adder operand-a operand-b)))

(example-labels 6 4) => 10

**lambda** *lambda-list &rest body*

Special Form

Provided as a convenience, to obviate the need for using the function special form when the latter is used to name an anonymous (lambda) function. When lambda is used as a special form, it is treated by the evaluator and compiler identically to the way it would have been treated if it appeared as the operand of a function special form. For example, the following two forms are equivalent:

(my-mapping-function (lambda (x) (+ x 2)) list)

(my-mapping-function (function (lambda (x) (+ x 2))) list)

Note that the form immediately above is usually written as:

(my-mapping-function #'(lambda (x) (+ x 2)) list)

The first form uses lambda as a special form; the latter two do not use the lambda special form, but rather, use lambda to name an anonymous function.

See the section "Functions and Special Forms for Constant Values".

Using lambda as a special form is incompatible with Common Lisp.
**lambda-list-keywords**  
_A constant_  
A list of all of the allowed "&" keywords. Some of these are obsolete and should not be used in new code.

For more information on lambda-list keywords: See the section "Lambda-List Keywords". See the section "Evaluating a Function Form".

---

**&optional**
Declares the following arguments to be optional. See the section "Evaluating a Function Form".

---

**&rest**
Declares the following argument to be a rest argument. There can be only one &rest argument.

Under Genera, it is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is stored in the function-calling stack, and loses its validity when the function returns. If a rest-argument is to be returned or made part of permanent list-structure, it must first be copied, as you must always assume that it is one of these special lists. See the function sys:copy-if-necessary.

The system does not detect the error of omitting to copy a rest-argument; you simply find that you have a value that seems to change behind your back. At other times the rest-args list is an argument that was given to apply; therefore it is not safe to rplaca this list, because you might modify permanent data structure. An attempt to rplacd a rest-args list is unsafe in this case, while in the first case it causes an error, since lists in the stack are impossible to rplacd.

Under CLOE, rest arguments are not typically stack-consed. You can move a rest-arg consed on the stack using the declaration (sys:downward-rest-argument).

---

**&key**
Separates the positional parameters and rest parameter from the keyword parameters. See the section "Evaluating a Function Form".

---

**&allow-other-keys**
In a lambda-list that accepts keyword arguments, says that keywords that are not specifically listed after &key are allowed. They and the corresponding values are ignored, as far as keyword arguments are concerned, but they do become part of the rest argument, if there is one.

---

**&aux**
Separates the arguments of a function from the auxiliary variables. Following &aux you can put entries of the form:

```
(variable initial-value-form)
```
or just variable if you want it initialized to nil or do not care what the initial value is.
&body For macros defined by defmacro or macrolet only. &body is similar to &rest, but declares to grindef and the code-formatting module of the editor that the body forms of a special form follow and should be indented accordingly. See the macro defmacro.

&whole For macros defined by defmacro or macrolet only. &whole is followed by variable, which is bound to the entire macro-call form or subform. variable is the value that the macro-expander function receives as its first argument. &whole is allowed only in the top-level pattern, not in inside patterns. See the macro defmacro.

&environment For macros defined by defmacro or macrolet only. &environment is followed by variable, which is bound to an object representing the lexical environment where the macro call is to be interpreted. This environment might not be the complete lexical environment. It should be used only with the macroexpand function for any local macro definitions that the macrolet construct might have established within that lexical environment. &environment is allowed only in the top-level pattern, not in inside patterns. See the section “Lexical Environment Objects and Arguments”. See the macro defmacro.

zl:&special Declares the following arguments and/or auxiliary variables to be special within the scope of this function. zl:&special can appear anywhere in the lambda-list any number of times. Note that you cannot use this keyword if you are using CLOE.

zl:&local Turns off a preceding zl:&special for the variables that follow. zl:&local can appear anywhere in the lambda-list any number of times. Note that you cannot use this keyword if you are using CLOE.

zl:&quote Using zl:&quote is an obsolete way to define special functions. zl:&quote declares that the following arguments are not to be evaluated. You should implement language extensions as macros rather than through special functions, because macros directly define a Lisp-to-Lisp translation and therefore can be understood by both the interpreter and the compiler.

Special functions, on the other hand, only extend the interpreter. The compiler has to be modified to understand each new special function so that code using it can be compiled. Since all real programs are eventually compiled, writing your own special functions is strongly discouraged. Note that you cannot use this keyword in CLOE.

zl:&eval This is obsolete. Use macros instead to define special functions. zl:&eval
turns off a preceding \texttt{zl:quote} for the arguments which follow. Note that if you are using CLOE, you cannot use this keyword.

\textbf{zl:list-of}

This is not supported. Use \texttt{loop} or \texttt{mapcar} instead of \texttt{zl:list-of}.

\textbf{lambda-macro \texttt{name} \texttt{lambda-list} \&body \texttt{body} \texttt{Function}}

Like \texttt{macro}, defines a lambda macro to be called \texttt{name}. \texttt{lambda-list} should be a list of one variable, which is bound to the function being expanded. The lambda macro must return a function. Example:

\begin{verbatim}
  (lambda-macro ilisp (x)
      '(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
\end{verbatim}

This defines a lambda macro called \texttt{ilisp}. After it has been defined, the following list is a valid Lisp function:

\begin{verbatim}
  (ilisp (x y z) (list x y z))
\end{verbatim}

The above function takes three arguments and returns a list of them, but all of the arguments are optional and any extra arguments are ignored. (This shows how to make functions that imitate Interlisp functions, in which all arguments are always optional and extra arguments are always ignored.) So, for example:

\begin{verbatim}
  (funcall #'(ilisp (x y z) (list x y z)) 1 2)  =>  (1 2 nil)
\end{verbatim}

\textbf{lambda-parameters-limit \texttt{Constant}}

A positive integer that is the upper exclusive bound on the number of distinct parameter names that can appear in a single lambda-list. The value is currently 128 for 3600-series machines and 50 for Ivory-based machines, and CLOE. If you are using CLOE, consider this example:

\begin{verbatim}
  (if (> (length keyword-pair-list) lambda-parameter-limit)
      (handle-too-many-keywords keyword-pair-list))
\end{verbatim}

\textbf{last \texttt{x} \texttt{l}, \texttt{x} \texttt{O}, \texttt{x} \texttt{n} \texttt{Function}}

Using \texttt{last} with the arguments \texttt{x l} returns the last cons of list \texttt{x}. If \texttt{x l} is \texttt{nil}, it returns \texttt{nil}. Note that \texttt{last} is not analogous to \texttt{first} (\texttt{first} returns the first element of a list, but \texttt{last} does not return the last element of a list); this is a historical artifact. Example:

\begin{verbatim}
  (setq x '(a b c d))
  (last x) => (d)
  (rplacd (last x) '(e f))
  x => '(a b c d e f)
\end{verbatim}

Using \texttt{last} with the arguments \texttt{x o} returns the cdr of the last cons of the list. Using \texttt{last} with the arguments \texttt{x n} returns the list of the last \texttt{n} conses of the list.
last could have been defined by:

```
(defun last (x)
  (cond ((atom x) x)
        ((atom (cdr x)) x)
        ((last (cdr x)))))
```

```
(setq b '(q r s t)) => (QRST)
(last b) => (T)
```

In the following example, last is used in the body of the do* to locate the cons for operation on by rplacd:

```
(defun my-nconc( &rest lists )
  (setq lists (remove nil lists :test #'eq))
  (do* ((segment1 (first lists) segment2)
       (segment2 (second lists) (first list))
       (result segment1)
       (list (rest (rest lists)) (rest list)))
       ((null segment2) result)
      (rplacd (last segment1) segment2)))
```

For a table of related items: See the section "Functions for Extracting from Lists".

**lcm &rest integers**

Computes and returns the least common multiple of the absolute values of its arguments. All the arguments must be integers, and the result is always a non-negative integer.

For one argument, lcm returns the absolute value of that argument. If one or more of the arguments is zero, lcm returns zero. If there are no arguments, the returned value is 1.

Examples:

```
(lcm) => 1
(lcm -6) => 6 ;absolute value of only one argument
(lcm 6 15) => 30
(lcm 0 6) => 0
(lcm 2 3 4 5) => 60
(lcm -15 105) => 105
(lcm 15 12 9) => 180
(lcm 5 7 11 18) => 6930
```

For a table of related items, see the section "Arithmetic Functions".
ldb bytespec integer

"Load byte."

Returns a byte extracted from integer as specified by bytespec.

bytespec is built using function byte with bit size and position arguments.

ldb extracts from integer size contiguous bits starting at position and returns this value. integer must be an integer.

The result is right-justified: the size bits are the lowest bits in the returned value and the rest of the returned bits are zero. ldb always returns a nonnegative integer. This function has a setf method. However, in order to use zl:setf on an ldb form, the integer argument must suit the zl:setf operation. Examples:

(ldb (byte 1 2) 5) => 1
(ldb (byte 32. 0) -1) => (1- 1_32.) ;; a positive bignum
(ldb (byte 16. 24.) -1_31.) => #0177000
(ldb (byte 6 3) #04567) => #056
(setq eight-x-seven 56)
(setf (ldb (byte 3 3) eight-x-seven) 4) => 4
eight-x-seven => 32
(ldb (byte 7 0) 257) => 1

For a table of related items: See the section "Summary of Byte Manipulation Functions".

ldb-test bytespec integer

Returns t if any of the bits designated by the byte specifier bytespec are 1's in integer. That is, it returns t if the designated field is nonzero. ldb-test could have been defined as follows:

(ldb-test bytespec integer) => (not (zerop (ldb bytespec integer)))

Examples:

(ldb-test (byte 2 1) 6) => T
(ldb-test (byte 2 3) #0542) => NIL

For a table of related items: See the section "Summary of Byte Manipulation Functions".

ldiff list sublist

Returns a new list, whose elements are those elements of list that appear before sublist. list should be a list, and sublist should be eq one of the conses that make up list.

Examples:
\begin{verbatim}
(setq x '(a b c d e))
(setq y (cdddr x)) => (d e)
(lendiff x y) => (a b c)
(lendiff '(a b c d) '(c d)) => (a b c d)
\end{verbatim}

For a table of related items: See the section "Functions for Comparing Lists"

\textbf{least-negative-double-float} \hspace{1cm} \textit{Constant}

The negative floating-point number in double-float format which is closest in value (but not equal to) zero.

\textbf{least-negative-long-float} \hspace{1cm} \textit{Constant}

The negative floating-point number in long-float format closest in value (but not equal to) zero. In Symbolics Common Lisp this constant has the same value as \textbf{least-negative-double-float}.

\textbf{least-negative-normalized-double-float} \hspace{1cm} \textit{Constant}

The normalized negative floating-point number in double-float format which is closest in value (but not equal to) zero. Its value is $-2.2250738585072014d-308$.

\textbf{least-negative-normalized-long-float} \hspace{1cm} \textit{Constant}

The normalized negative floating-point number in long-float format which is closest in value (but not equal to) zero. Its value is the same as \textbf{least-negative-normalized-double-float}, $-2.2250738585072014d-308$.

\textbf{least-negative-normalized-short-float} \hspace{1cm} \textit{Constant}

The normalized negative floating-point number in short-float format which is closest in value (but not equal to) zero. Its value is the same as \textbf{least-negative-normalized-single-float}, $-1.1754944e-38$.

\textbf{least-negative-normalized-single-float} \hspace{1cm} \textit{Constant}

The normalized negative floating-point number in single-float format which is closest in value (but not equal to) zero. Its value is $-1.1754944e-38$.

\textbf{least-negative-short-float} \hspace{1cm} \textit{Constant}

The negative floating-point number in short-float format closest in value (but not equal to) zero. In Symbolics Common Lisp this constant has the same value as \textbf{least-negative-single-float}.
least-negative-single-float

The negative floating-point number in single-float format that is closest in value (but not equal to) zero.

least-positive-double-float

The positive floating-point number in double-float format closest in value (but not equal to) zero.

least-positive-long-float

The positive floating-point number in single-float format closest in value (but not equal to) zero. In Symbolics Common Lisp this constant has the same value as least-positive-double-float.

least-positive-normalized-double-float

The normalized positive floating-point number in double-float format closest in value (but not equal to) zero. Its value is 2.2250738585072014d-308.

least-positive-normalized-long-float

The normalized positive floating-point number in long-float format closest in value (but not equal to) zero. Its value is the same as least-positive-normalized-double-float, 2.2250738585072014d-308.

least-positive-normalized-short-float

The normalized positive floating-point number in short-float format closest in value (but not equal to) zero. Its value is the same as least-positive-normalized-single-float, 1.1754944e-38.

least-positive-normalized-single-float

The normalized positive floating-point number in single-float format closest in value (but not equal to) zero. Its value is 1.1754944e-38.

least-positive-short-float

The positive floating-point number in short-float format closest in value (but not equal to) zero. In Symbolics Common Lisp this constant has the same value as least-positive-single-float.

least-positive-single-float

The negative floating-point number in single-float format that is closest in value (but not equal to) zero.
The positive floating-point number in single-float format closest in value (but not equal to) zero.

**length sequence**

*Function*

Returns the number of elements in `sequence` as a non-negative integer. If the sequence is a vector with a fill pointer, the "active length" as specified by the fill pointer, is returned.

`sequence` can be either a list or a vector (one-dimensional array). Note that `nil` is considered to be a sequence, of length zero.

For example:

```
(length '()) => 0
(length '(a b c)) => 3
(length '(a (b c) d e)) => 4
(length (vector 'a 'b 'c 'd 'e)) => 5
```

The following example defines a simplified replacement function. This function uses `length` to ensure that the end values default to the length of the sequences.

```
(defun my-replace (sequence1 sequence2 &key start1 end1 start2 end2)
  "real replace must do some extra work"
  (unless end1 (setq end1 (length sequence1)))
  (unless end2 (setq end2 (length sequence2)))
  (setf (subseq sequence1 start1 end1)
        (subseq sequence2 start2 end2))
  sequence1)
```

See the section "Array Leaders".

For a table of related items: See the section "Functions for Finding Information About Lists and Conses".

For a table of related items: See the section "Sequence Construction and Access". Also: See the section "Getting Information About an Array".

**:length**

*Message*

Returns the length of the file, in bytes or characters. For text files on PDP-10 file servers, this is the number of PDP-10 characters, not Symbolics characters. The numbers are different because of character-set translation. (See the section "The Character Set".) For an output stream the length is not meaningful until after the stream has been closed, at least on an ITS file server.

**zl:length x**

*Function*
Returns the length of $x$. The length of a list is the number of elements in it. Examples:

- `(zl:length nil) => 0`
- `(zl:length '(a b c d)) => 4`
- `(zl:length '(a (b c) d)) => 3`

`zl:length` could have been defined by:

```lisp
(defun zl:length (x)
  (cond ((atom x) 0)
        ((1+ (zl:length (cdr x))))
)
```

or by:

```lisp
(defun zl:length (x)
  (do ((n 0 (1+ n))
       (y x (cdr y)))
      ((atom y) n))
```

except that it is an error to take `zl:length` of a non-nil atom.

For a table of related items: See the section "Functions for Finding Information About Lists and Conses".

For a table of related items: See the section "Sequence Construction and Access".

`zl:lessp number &rest more-numbers`  

In your new programs, we recommend that you use function `<`, which is the Common Lisp equivalent of `zl:lessp`.

`zl:lessp` compares its arguments from left to right. If any argument is not less than the next, `zl:lessp` returns `nil`. But if the arguments are monotonically strictly increasing, the result is `t`.

Arguments must be noncomplex numbers, but they need not be of the same type.

Examples:

- `(zl:lessp 3 4) => t`
- `(zl:lessp 1 1) => nil`
- `(zl:lessp 0 1 2 3 4) => t`
- `(zl:lessp 0 1 0.5/2 3 2 4) => nil`

`let bindings &body body`  

Binds some variables to some objects and evaluates some forms (the "body") in the context of those bindings. A `let` form looks like this:
When this form is evaluated, first the \textit{vforms} (the values) are evaluated. Then the variables are bound to the values returned by the corresponding \textit{vforms}. Thus the bindings happen in parallel; all the \textit{vforms} are evaluated before any of the variables are bound. Finally, the \textit{bforms} (the body) are evaluated sequentially, the old values of the variables are restored, and the result of the last \textit{bform} is returned.

The body of the \texttt{let} form is an implicit \texttt{progn}.

You can omit the \texttt{vform} from a \texttt{let} clause, in which case it is as if the \texttt{vform} were \texttt{nil}: the variable is bound to \texttt{nil}. Furthermore, you can replace the entire clause (the list of the variable and form) with just the variable, which also means that the variable gets bound to \texttt{nil}. It is customary to write just a variable, rather than a clause, to indicate that the value to which the variable is bound does not matter, because the variable is \texttt{setq}’ed before its first use. Example:

\begin{verbatim}
(let ((a (+ 3 3))
     (b 'foo)
     (c)
     d)
...)
\end{verbatim}

Within the body, \texttt{a} is bound to 6, \texttt{b} is bound to \texttt{foo}, \texttt{c} is bound to \texttt{nil}, and \texttt{d} is bound to \texttt{nil}.

The values of any special variables bound by \texttt{let} are restored upon returned value of \texttt{let}.

\begin{verbatim}
(setq a '(1 2 3) b '(3 4 5))
(let ((one a)
     (two (cdr b)))
     (append one two))
=> (1 2 3 4 5)
\end{verbatim}

The special form \texttt{let} and its companion \texttt{let*} are most useful for providing a context with local variables for temporary storage during a computation. For example:

\begin{verbatim}
(let ((list arg1)
     (ptr (car arg1))
     (rest (cdr arg1)))
(lisp:loop
     (process ptr)
     (unless rest (return))
     (setq list rest)
     (setq ptr (car list))
     (setq rest (cdr rest))))
\end{verbatim}

Nesting of \texttt{let} forms is also possible, for example, to avoid use of \texttt{let*}:
(let ((*print-escape* nil)
       (array (get-my-array)))
  (let ((message (format nil "~A" array)))
    (my-process message)))

See the section "Special Forms for Binding Variables".

**let* bindings &body body**

Binds some variables to some objects, sequentially, and evaluates some forms (the "body") in the context of those bindings. **let*** is the same as **let**, except that the binding is sequential. Each variable is bound to the value if its **vform** before the next **vform** is evaluated. This is useful when the computation of a **vform** depends on the value of a variable bound in an earlier **vform**. Example:

```
(let* ((a (+ 1 2))
       (b (* a a))
       ...)
```

Within the body, a is bound to 3 and b is bound to 6.

The body of the **let*** form is an implicit **progn**. Therefore, the **forms** are evaluated sequentially, and **let*** returns the value of the last **form** evaluated. The values of any special variables bound by **let*** are restored upon the returned value of the **let***.

```
(setq a '(1 2 3) b '(3 4 5))
(let* ((one (append a b))
       (two (remove-duplicates one)))
  two)
=> (1 2 3 4 5)
```

Special forms **let*** and **let** provide a local variable context for temporary storage during a computation. For example:

```
(let* ((list arg1)
       (ptr (car list))
       (rest (cdr list)))
  (tagbody loop
    (process ptr)
    (when rest
      (setq list rest)
      (setq ptr (car list))
      (setq rest (cdr rest))
      (go loop))))
```

See the section "Special Forms for Binding Variables".

**let-and-make-dynamic-closure vars &body body**

Function

When using dynamic closures, it is very common to bind a set of variables with initial values, and then make a closure over those variables. Furthermore, the vari-
ables must be declared as "special". **let-and-make-dynamic-closure** is a special form that does all of this. It is best described by example:

```
(let-and-make-dynamic-closure ((a 5) b (c 'x))
  (function (lambda () ...)))
```

macro-expands into

```
(let ((a 5) b (c 'x))
  (declare (special a b c))
  (make-dynamic-closure '(a b c)
    (function (lambda () ...))))
```

See the section "Dynamic Closure-Manipulating Functions".

**zl:let-closed** vars &body body

When using dynamic closures, it is very common to bind a set of variables with initial values, and then make a closure over those variables. Furthermore, the variables must be declared as "special". **zl:let-closed** is a special form that does all of this. It is best described by example:

```
(zl:let-closed ((a 5) b (c 'x))
  (function (lambda () ...)))
```

macro-expands into

```
(zl:let ((a 5) b (c 'x))
  (declare (special a b c))
  (closure '(a b c)
    (function (lambda () ...))))
```

The Symbolics Common Lisp equivalent of this function is **let-and-make-dynamic-closure**. See the section "Dynamic Closure-Manipulating Functions".

**let-globally** varlist &body body

Similar in form to **letf**. The difference is that **let-globally** does not bind the variables; instead, it saves the old values and sets the variables, and sets up an **unwind-protect** to set them back. The important difference between **let-globally** and **letf** is that when the current stack group calls some other stack group, the old values of the variables are not restored. Thus, **let-globally** makes the new values visible in all stack groups and processes that do not bind the variables themselves, not just the current stack group.

See the section "Special Forms for Binding Variables".
let-globally-if \texttt{cond \ varlist \&body body}

\textit{Special Form}

Similar to \texttt{let-globally}. It takes a \texttt{cond} form as its first argument. It binds the variables only if \texttt{cond} evaluates to something other than \texttt{nil}. \texttt{body} is evaluated in either case.

let-if \texttt{cond \ bindings \&body body}

\textit{Special Form}

A variant of \texttt{let} in which the binding of variables is conditional. The variables must all be special variables. The \texttt{let-if} special form, typically written as:

\begin{verbatim}
(let-if \texttt{cond}
  \texttt{(\((\texttt{var-1 \ val-2) \ (var-1 \ val-2})\ldots\))}
  \texttt{body-form1 \ body-form2 \ldots})
\end{verbatim}

first evaluates the predicate form \texttt{cond}. If the result is non-\texttt{nil}, \texttt{bindings} (in the example above, \texttt{val-1, val-2, and so on}, are evaluated and then the variables \texttt{var-1, var-2, and so on}, are bound to them). If the result is \texttt{nil}, \texttt{bindings} are ignored. Finally the body forms are evaluated.

See the section “Special Forms for Binding Variables”.

letf \texttt{places-and-values \&body body}

\textit{Special Form}

Just like \texttt{let}, except that it can bind any storage cells rather than just variables. The cell to be bound is specified by an access form that must be acceptable to \texttt{locf}. For example, \texttt{letf} can be used to bind slots in a structure. \texttt{letf} does parallel binding.

Given the following structure, \texttt{letf} calls \texttt{do-something-to} with \texttt{ship}’s \texttt{x} position bound to zero.

\begin{verbatim}
(defstruct ship position-x position-y) => SHIP
(setq QE2 (make-ship)) => #S(SHIP :POSITION-X NIL :POSITION-Y NIL)

(letf (((ship-position-x QE2) 0))
  (do-something-to QE2))
\end{verbatim}

It is preferable to use \texttt{letf} instead of the \texttt{sys:%bind-location} and \texttt{sys:%with-binding-stack-level} subprimitives.

See the section “Special Forms for Binding Variables”.

letf* \texttt{places-and-values \&body body}

\textit{Special Form}
Just like let*, except that it can bind any storage cells rather than just variables. The cell to be bound is specified by an access form that must be acceptable to locf. For example, letf* can be used to bind slots in a structure. letf* does sequential binding.

Given the following structure, letf* calls do-something-to with ship's x position bound to 0 and y position bound to 5.

```lisp
(defstruct ship position-x position-y) => SHIP
(setq QE2 (make-ship)) => #S(SHIP :POSITION-X NIL :POSITION-Y NIL)

(letf* (((ship-position-x QE2) 0)
    ((ship-position-y QE2) (+ (ship-position-x QE2) 5)))
  (do-something-to QE2))
```

It is preferable to use letf* instead of the zl:bind subprimitive.
See the section "Special Forms for Binding Variables".

**sys:lexical-closure**

sys:lexical-closure is the type specifier symbol for the predefined Lisp object of that name.

Examples:

```lisp
(typep *standard-output* 'sys:lexical-closure) => T
(zl:typep *standard-output*) => :LEXICAL-CLOSURE
(sys:type-arglist 'sys:lexical-closure) => NIL and T
```

See the section "Data Types and Type Specifiers".
See the section "Scoping".

**lexpr-continue-whopper** &rest args

Calls the methods for the generic function that was intercepted by the whopper in the same way that continue-whopper does, but the last element of args is a list of arguments to be passed. This is useful when the arguments to the intercepted generic function include an &rest argument. Returns the values returned by the combined method.

For more information on whoppers, including examples: See the section "Wrappers and Whoppers".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**lexpr-send** object message-name &rest arguments

Sends the message named message-name to the object. arguments are the arguments passed, except that the last element of arguments should be a list, and all the elements of that list are passed as arguments. For example:
(send some-window :set-edges 10 10 40 40)

does the same thing as these forms do:

(lexpr-send some-window :set-edges 10 '(10 40 40))
(lexpr-send some-window :set-edges 10 10 '(40 40))
(lexpr-send some-window :set-edges 10 10 40 '(40))

lexpr-send is to send as zl:lexpr-funcall is to funcall.

lexpr-send is supported for compatibility with previous versions of the flavor system. When writing new programs, it is good practice to use generic functions instead of message-passing.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**lexpr-send-if-handles**

object message &rest arguments

Function

object performs the operation indicated by message with the given arguments, if it has a method for the operation. If no method for the operation is available, nil is returned.

object is a Lisp object, usually a flavor instance. message is a message name or a generic function object, such as the result of evaluating the form (flavor:generic generic-function-name). arguments are the arguments for the operation.

The difference between lexpr-send-if-handles and send-if-handles is that for lexpr-send-if-handles, the last element of arguments is a list of arguments, all of which are used as arguments to the operation.

lexpr-send-if-handles is to send-if-handles as lexpr-send is to send.

For information on restrictions in using lexpr-send-if-handles with generic functions: See the function send-if-handles.

Note that lexpr-send-if-handles works by sending the :send-if-handles message. You can customize the behavior of lexpr-send-if-handles by defining a method for the :send-if-handles message.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**:line-in** &optional leader

Message

The stream should input one line from the input source and return it as a string with the carriage return character stripped off. Despite its name, this operation is not much like the zl:readline function.

Many streams have a string that is used as a buffer for lines. If this string itself were returned, there would be problems if the caller of the stream attempted to save the string away somewhere, because the contents of the string would change when the next line was read in. To solve this problem, the string must be copied. On the other hand, some streams do not reuse the string, and it would be wasteful
to copy it on every :line-in operation. This problem is solved by using the leader argument to :line-in. If leader is nil (the default), the stream does not copy the string, and the caller should not rely on the contents of that string after the next operation on the stream. If leader is t, the stream makes a copy. If leader is an integer then the stream makes a copy with an array-leader leader elements long. (This is used by the editor, which represents lines of buffers as strings with additional information in their array-leaders, to eliminate an extra copy operation.)

If the stream reaches the end-of-file while reading in characters, it returns the characters it has read in as a string, and returns a second value of t. The caller of the stream should therefore arrange to receive the second value, and check it to see whether the string returned was an whole line or only the trailing characters after the last carriage return in the input source.

The :line-in message can be sent to windows. It interacts correctly with the input editor, including correct handling of activation characters.

:line-out string &optional start end  
Message
Outputs the characters of string, followed by a carriage return character, to the stream. start and end optionally specify a substring, as with :string-out. If the stream does not support :line-out itself, the default handler converts it to :tyos.

lisp-implementation-type  
Function
Returns a string that is the name of the Lisp system running on your machine.

(lisp-implementation-type) => "Symbolics Common Lisp"
or

(lisp-implementation-type) => "Symbolics CLOE"

lisp-implementation-version  
Function
Returns a string that identifies the current version of the system running on your machine, including the patch level and microcode.

(lisp-implementation-version)
  => "System 424.207 3640-MIC microcode 428"

For the CLOE Developer,

(lisp-implementation-version)
  => "1.1, Cloe Developer 318.0"
and for the CLOE Application Generator,

  => (lisp-implementation-version)
    "CLOE Application Generator 1.1"

si:lisp-top-level1 &optional (stream zl:terminal-io)  
Function
This is the actual top-level loop. It reads a form from *standard-input*, evaluates it, prints the result (with slashification) to *standard-output*, and repeats indefinitely. If several values are returned by the form, all of them will be printed. The values of *, +, -, /, ++, **, +++ and *** are maintained.

**list**

*list* is the type specifier symbol for the predefined Lisp data structure of that name.

The types *list* and *vector* are an exhaustive partition of the type *sequence*, since sequence ≡ (or list vector).

Examples:

- (typep `(a b c) 'list) => T
- (zl:typep `(a b (d c) e)) => :LIST
- (subtypep 'list 'sequence) => T and T
- (sys:type-arglist 'list) => NIL and T
- (listp ()) => T
- (listp '(2.0s0 (a 1) #\*)) => T
- (listp '(
  | a | b |)
) => T

See the section "Data Types and Type Specifiers". See the section "Lists".

**list &rest elements**

Constructs and returns a list of its arguments. Example:

- (list 3 4 'a (car '(b . c)) (+ 6 -2)) => (3 4 a b 4)

list could have been defined by:

```lisp
(defun list (&rest args)
  (let ((list (make-list (length args))))
    (do ((l list (cdr l))
         (a args (cdr a)))
        ((null a) list)
      (rplaca l (car a))))
```

Using list helps avoid clumsy nesting callsto cons by providing a clean constructor for lists (as opposed to trees).

- (list 'a 'b) = (cons 'a (cons 'b nil)) => (A B)
- (list 'a 'b 'c 'd (cons 'e 'f) 'g) =>
  (A B C D E F G)
- (list 'a 'b 'c 'd) = (list* 'a 'b 'c 'd '())

For a table of related items: See the section "Functions for Constructing Lists and Conses".
list* &rest args

Constructs and returns a list of its arguments, whose last cons is "dotted". It must be given at least one argument. Example:

```
(list* 'a 'b 'c 'd) =>
(a b c . d)
```

This is like

```
(cons 'a (cons 'b (cons 'c 'd)))
```

More examples:

```
(list* 'a 'b) =>
(a . b)
(list* 'a) => a
```

list* is like list, except that the last argument is not consed with nil. When applied to one argument, list* simply returns the argument. A true list is returned when the last argument of list* is a true list, such as nil. Using list* also helps avoid clumsy nesting calls to cons.

```
(list* 'a 'b 'c) = (cons 'a (cons 'b 'c))
= (A B . C)
(list* 'temp) => temp
(list* 'temp nil) = (list 'temp) => (temp)
```

When using list to create a new list from given elements, list* is the preferred function for adding a number of new elements to an already existing list:

```
(setq my-friends (list 'jim 'fred)) => (JIM FRED)
(setq my-friends
 (list* 'jack 'john 'bill my-friends))
 => (JACK JOHN BILL JIM FRED)
```

For a table of related items: See the section "Functions for Constructing Lists and Conses".

math:list-2d-array array

Returns a list of lists containing the values in array, which must be a two-dimensional array. There is one element for each row; each element is a list of the values in that row.

list-all-packages

Returns a list of all the packages that exist in Genera or CLOE.
The following example shows the definition of a macro similar to `do-all-symbols`, but which touches only external symbols.

```
(defmacro do-all-external-symbols ((variable) &body forms)
  (let ((package-variable (gensym)))
    `(dolist (,package-variable (list-all-packages))
       (do-external-symbols (,variable ,package-variable)
        ,forms))))
```

**list-array-leader** *array* &optional *limit*

Function

Creates and returns a list whose elements are those of *array*'s leader. *array* can be any type of array or a symbol whose function cell contains an array.

If *limit* is present, it should be an integer, and only the first *limit* (if there are more than that many) elements of *array*'s leader are used, and so the maximum length of the returned list is *limit*. If *array* has no leader, nil is returned.

For a table of related items: See the section "Copying an Array".

**list-in-area** *area* &rest *elements*

Function

Constructs and returns a list of its arguments, and takes an area number argument, and creates the list in that area. See the section "Areas".

**list-in-area** is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**list*-in-area** *area* &rest *args*

Function

Constructs and returns a list of its arguments, whose last cons is "dotted", and takes an area number argument, and creates the list in that area.

See the section "Areas".

**list*-in-area** is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**list-length** *list*

Function

Returns, as an integer, the length of *list*. **list-length** differs from **length** when *list* is circular. In these cases, **length** can fail to return, whereas **list-length** returns nil. For example:

```
(list-length '()) => 0
```
(list-length '(a b c d)) => 4

(list-length '(a (b c) d)) => 3

(let ((x (list 'a 'b 'c)))
  (rplacd (last x) x)
  (list-length x)) => NIL

If the argument is known to be non-circular, list-length is less efficient than length because it performs significantly more work to determine the existence of circularities.

(setq *print-circle* t)
(setq a '(1 2 3 4 5))
(list-length a) => 5
(rplacd (last a) (cddr a))
a => (1 2 . #1=(3 4 5 . #1#))
(list-length a) => nil

See the function length.

For a table of related items: See the section "Functions for Finding Information About Lists and Conses".

zl:listarray array &optional limit Function

Creates and returns a list whose elements are those of array. array can be any type of array or a symbol whose function cell contains an array.

If limit is present, it should be an integer, and only the first limit (if there are more than that many) elements of array are used, and so the maximum length of the returned list is limit.

If array is multidimensional, the elements are accessed in row-major order: the last subscript varies the most quickly.

listen &optional input-stream Function

The predicate listen returns t if there is a character immediately available from input-stream, and otherwise it returns nil. This is particularly useful when the stream obtains characters from an interactive device such as a keyboard. A call to read-char would simply wait until a character was available, but listen can sense whether or not to attempt input. On a non-interactive stream, the general rule is that listen returns t except when it's at EOF.

(listen)
=> NIL
(let ((c (read-char)))
  (list c
    (listen)
    (progn (unread-char c) (listen))
    (progn (peek-char) (listen))
    (progn (read-char) (listen)))
=> (#\x NIL T T NIL)

:listen

Message

Tests whether the user has pressed a key, perhaps trying to stop a program in progress. :listen does not err; it returns either non-nil or nil. This makes it useful as a wait function.

On an interactive device, :listen returns non-nil if any input characters are immediately available, or nil if not, which implies that :tyi would hang. If :tyi would err, that is not considered hanging, and :listen returns non-nil in this case.

On a noninteractive device, the operation always returns non-nil except at end-of-file, by virtue of the default handler.

zl:listify n

Function

Manufactures a list of n of the arguments of a lexpr. With a positive argument n, it returns a list of the first n arguments of the lexpr. With a negative argument n, it returns a list of the last (abs n) arguments of the lexpr. Basically, it works as if defined as follows:

(defun zl:listify (n)
  (cond ((minusp n)
           (listify1 (arg nil) (+ (arg nil) n 1)))
        (t
           (listify1 n 1)))

(defun listify1 (n m) ; auxiliary function.
  (do ((i n (1- i))
       (result nil (cons (arg i) result)))
      ((< i m) result)))

zl:listify exists only for compatibility with Maclisp lexprs. To write functions that can accept variable numbers of arguments, use the &optional and &rest keywords. See the section "Evaluating a Function Form".

listp object

Function

Returns t if its argument is a list, otherwise nil. This means (listp nil) is t. Note this distinction between listp and zl:listp. (zl:listp nil) is nil, since zl:listp returns t if its argument is a cons.
(listp object) = (or (consp object) (null object))

Example:
(listp '(5 9 12 16 8))
returns t, since the argument is a list. But:
(listp '5)
returns nil, since the argument is not a list.
(listp (cons 'a 'b)) => t
(listp 24) => nil

(if (listp object)
   (my-function (car object) (cdr object))
   (alt-function (test-for-type object)))

For a table of related items: See the section "Predicates that Operate on Lists".

listp object

Function

Returns t if its argument is a list, otherwise nil. This means (listp nil) is t. Note this distinction between listp and zl:listp. (zl:listp nil) is nil, since zl:listp returns t if its argument is a cons.

(listp object) = (or (consp object) (null object))

Example:
(listp '(5 9 12 16 8))
returns t, since the argument is a list. But:
(listp '5)
returns nil, since the argument is not a list.
(listp (cons 'a 'b)) => t
(listp 24) => nil

(if (listp object)
   (my-function (car object) (cdr object))
   (alt-function (test-for-type object)))

For a table of related items: See the section "Predicates that Operate on Lists".

zl:listp object

Function

In your new programs, we recommend that you use the function consp, which is the Common Lisp equivalent of zl:listp.

Returns t if its argument is anything (for example, a symbol, array, or flavor instance, etc.) except nil. If its argument is nil, zl:listp returns nil. Note that this means (zl:listp nil) is nil even though nil is the empty list.
For a table of related items, see the section "Predicates that Operate on Lists".

**load-byte** `from-value position size`  
*Function*

Like `ldb`, except that instead of using a byte specifier, the bit `position` and `size` are passed as separate arguments. The argument order is not analogous to that of `ldb` so that `load-byte` can be compatible with older versions of Lisp.

For a table of related items: See the section "Summary of Byte Manipulation Functions".

**sys:local-declarations**  
*Variable*

A list of local declarations. Each declaration is itself a list whose car is an atom which indicates the type of declaration. The meaning of the rest of the list depends on the type of declaration. For example, in the case of `special` and `zl:unspecial` the cdr of the list contains the symbols being declared.

The compiler is interested only in `special`, `zl:unspecial`, `macro`, and `arglist` declarations.

Local declarations are added to `sys:local-declarations` in two ways:

- Inside a `zl:local-declare`, the specified declarations are bound onto the front.

- If `sys:undo-declarations-flag` is `t`, some kinds of declarations in a file that is being compiled are consed onto the front of the list; they are not popped until `sys:local-declarations` is unbound at the end of the file.

Note: `zl:local-declare` and `sys:local-declarations` are available in Genera, but should not be used for new code. See the section "Lexical Scoping".

**zl:local-declare** `declarations &body body`  
*Special Form*

This function, while available in Genera, should not be used for new code. See the section "Lexical Scoping". See the section "Operators for Making Declarations".

A `zl:local-declare` form looks like this:

```
(zl:local-declare (declaration declaration ...) form1 form2 ...)  
```

Example:
(z1:local-declare ((special foo1 foo2))
(defun larry ()
    )
(defun george ()
    )
); end of z1:local-declare

z1:local-declare understands the same declarations as declare.
Each local declaration is consed onto the list sys:local-declarations while the
forms are being evaluated (in the interpreter) or compiled (in the compiler). This
list has two uses. First, it can be used to pass information from outer macros to
inner macros. Secondly, the compiler specially interprets certain declarations as lo-
cal declarations, which apply only to the compilation of the forms.

sys:localize-list list &optional area Function
Improves locality of incrementally constructed lists and association lists.
sys:localize-list returns either list or a copy of list, depending on how sparsely it
is stored in virtual memory.
The optional area argument is the number of the area in which to create the new
list. (Areas are an advanced feature of storage management. See the section "Areas".)

sys:localize-list is a Symbolics extension to Common Lisp.
For a table of related items: See the section "Functions for Copying Lists".

sys:localize-tree tree &optional (n-levels 100) area Function
Improves locality of incrementally constructed lists and trees. sys:localize-tree re-
turns either tree or a copy of tree, depending on how sparsely it is stored in virtual
memory.
The optional argument n-levels is the number of levels of list structure to localize.
This is especially useful for association lists, where the value of n-levels is set to
2.
The optional area argument is the number of the area in which to create the new
tree. (Areas are an advanced feature of storage management. See the section "Areas".)

sys:localize-tree is a Symbolics extension to Common Lisp.
For a table of related items: See the section "Functions for Copying Lists".

locally &body body Macro
Makes local pervasive declarations wherever you need them (wherever you can
legally place a form). No variables are bound by this form, and no declarations in
this form alter enclosing bindings. You can use the special declaration to perva-
sively affect references to, rather than bindings of, variables. For example:
In the following example, we call a value swapping function within the scope of a `locally` call, and use a declaration that calls for optimization with respect to execution speed:

```
(locally (declare (optimize speed))
  (swap-values item-a item-b))
```

Special declarations are allowed only to affect references. See the section "Operators for Making Declarations".

### `zl:locate-in-closure closure symbol` Function

This returns the location of the place in the dynamic closure `closure` where the saved value of `symbol` is stored. An equivalent form is `(locf (zl:symeval-in-closure closure symbol))`. See the section "Dynamic Closure-Manipulating Functions".

### `zl:locate-in-instance instance symbol` Function

Returns a locative pointer to the cell inside `instance` that holds the value of the instance variable named `symbol`, regardless of whether the instance variable was declared a `:locatable-instance-variable`.

In Symbolics Common Lisp, this operation is performed by:

```
(locf (scl:symbol-value-in-instance instance symbol))
```

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

### `location-boundp location` Function

Takes a locative pointer to designate the cell rather than a symbol. It returns `t` if the cell at `location` is bound to a value, and otherwise it returns `nil`. `location-boundp` is a version of `boundp` that can be used on any cell.

The following two calls are equivalent:

```
(location-boundp (locf a))
(variable-boundp a)
```

The following two calls are also equivalent. When `a` is a special variable, they are also the same as the two calls in the preceding example.

```
(location-boundp (value-cell-location 'a))
(boundp 'a)
```
location-contents locative

Returns the contents of the cell at which locative points. For example:

```
(location-contents (value-cell-location x))
```

is the same as:

```
(syneval x)
```

To store objects into the cell at which a locative points, you should use `(setf (location-contents x) y)` as shown in the following example:

```
(setf (location-contents (value-cell-location x)) y)
```

This is the same as:

```
(set x y)
```

Note that location-contents is not the right way to read hardware registers, since cdr (which is called by location-contents) will in some cases start a block-read and the second read could easily read some register you didn't want it to. Therefore, you should use car or sys:%p-ldb as appropriate for these operations.

location-makunbound loc &optional variable-name

Takes a locative pointer to designate the cell rather than a symbol. (makunbound is restricted to use with symbols.)

location-makunbound is a version of makunbound that can be used on any cell in the Symbolics Lisp Machine.

location-makunbound takes a symbol as an optional second argument: variable-name of the location that is being made unbound. It uses variable-name to label the null pointer it stores so that the Debugger knows the name of the unbound location if it is referenced. This is particularly appropriate when the location being made unbound is really a variable value cell of one sort or another, for example, closure or instance.

locative

Type Specifier

locativep x

Function

Returns t if its argument is a locative, otherwise nil.

locf reference

Macro

Takes a form that accesses some cell and produces a corresponding form to create a locative pointer to that cell. Examples:

```
(locf (array-leader foo 3)) ==> (ap-leader foo 3)
(locf a) ==> (variable-location 'a)
(locf (plist 'a)) ==> (property-cell-location 'a)
(locf (aref q 2)) ==> (aloc q 2)
```
If access-form invokes a macro or a substitutable function, locf expands the access-form and starts over again. This lets you use locf together with zl:defstruct accessor.

If access-form is (cdr list), locf returns the list itself instead of a locative.

See the section "Generalized Variables".

For a table of related items: See the section "Basic Array Functions".

**log number &optional base** Function

Computes and returns the logarithm of number in the base base, which defaults to e, the base of the natural logarithms. Note that the result can be a complex number even when the argument is noncomplex. This occurs if the argument is negative.

The range of the one-argument log function is that strip of the complex plane containing numbers with imaginary parts between -π (exclusive) and π (inclusive).

The range of the two-argument log function is the entire complex plane. It is an error if number or base is zero. Both arguments can be numbers of any type.

The result is always in complex or noncomplex floating-point format. Numeric type coercion is applied to the arguments where proper.

Examples:

```
(log 2) => 0.6931472
(log 16 2) => 4.0
(log -1.0) => #C(0.0 3.1415927)
(log -1 #C(0 1)) => #C(2.0 0.0)
```

For a table of related items, see the section "Powers of e and Log Functions".

**zl:log n** Function

Returns the natural logarithm of n. n must be positive, and can be of any numeric data type.

Example:

```
(zl:log 2) => 0.6931472
(log 81 3) \rightarrow 4.0
(log (exp 4)) \rightarrow 4.0
(log -1) \rightarrow #C(0.0 3.1415927)
```

For a table of related items: See the section "Powers of e and Log Functions" and see CLtL 204.

**logand &rest integers** Function
Returns the bit-wise logical and of its arguments. If no argument is given the result is -1, which is an identity for this operation.

Examples:

(\text{logand}) \Rightarrow -1
(\text{logand} 8) \Rightarrow 8
(\text{logand} 9 15) \Rightarrow 9
(\text{logand} 9 15 12) \Rightarrow 8

See the function \text{boole}.

For a table of related items, see the section "Functions Returning Result of Bit-wise Logical Operations".

\text{zl:logand} \ \text{number} \ \&\text{rest} \ \text{more-numbers}

\text{Function}

Returns the bit-wise logical and of its arguments. At least one argument is required. Examples:

(\text{zl:logand} \ #o3456 \ #o707) \Rightarrow \ #o406
(\text{zl:logand} \ #o3456 \ #o-100) \Rightarrow \ #o3400

For a table of related items: See the section "Functions Returning Result of Bit-wise Logical Operations" and see CLtL 221.

\text{logandc1} \ \text{integer1} \ \text{integer2}

\text{Function}

This is a non-associative bit-wise logical operation and takes exactly two arguments. It returns the bit-wise logical and of the complement of integer1 with integer2.

Examples:

(\text{logandc1} 15 \ 8) \Rightarrow 0
(\text{logandc1} 8 15) \Rightarrow 7
(\text{logandc1} 1 4) \Rightarrow 4
(\text{logandc1} 2 6) \Rightarrow 4

See the function \text{boole}.

For a table of related items, see the section "Functions Returning Result of Bit-wise Logical Operations".

\text{logandc2} \ \text{integer1} \ \text{integer2}

\text{Function}

This is a non-associative bit-wise logical operation and takes exactly two arguments. It returns the bit-wise logical and of integer1 with the complement of integer2.

Examples:
(logandc2 15 8) => 7
(logandc2 8 15) => 0
(logandc2 1 4) => 1
(logandc2 2 6) => 0

See the function **boole**.

For a table of related items, see the section "Functions Returning Result of Bitwise Logical Operations".

**logbitp** index integer  
*Function*

If `index` is a non-negative integer `j`, the predicate **logbitp** is true if bit `j` in `integer` (that bit whose weight is \(2^j\)) is a one-bit; otherwise it is false.

Examples:

- `(logbitp 1 8) => NIL`
- `(logbitp 1 10) => T`
- `(logbitp 0 6) => nil`
- `(logbitp 1 6) => T`
- `(logbitp 2 6) => T`

For a table of related items, see the section "Predicates for Testing Bits in Integers".

**logcount** integer  
*Function*

If `integer` is positive, determines and returns the number of one-bits in the binary representation of `integer`. If `integer` is negative, **logcount** determines and returns the number of 0 bits in the two's-complement binary representation of `integer`. The result is always a non-negative integer.

Examples:

- `(logcount 0) => 0`
- `(logcount 6) => 2`
- `(logcount -1) => 0`
- `(logcount -5) => 1 ; -5 is #b \ldots 11011`
- `(logcount 7) => 3`
- `(logcount -11) => 2`

For a table of related items, see the section "Functions Returning Components or Characteristics of Argument".

**sys:%logdpb** newbyte bytespec integer  
*Function*

Like **dpb**, except that it only returns fixnums, while **dpb** would produce a bignum result for arithmetic correctness. If the sign-bit (bit-32) changes, the result reflects the changed sign.
sys:logdpb is good for manipulating fixnum bit-masks such as are used in some internal system tables and data structures.

The behavior of sys:logdpb depends on the size of fixnums, so functions using it might not work the same way on future implementations of Symbolics Common Lisp. Its name starts with "%%" because it is more like machine-level subprimitives than other byte manipulation functions.

For a table of related items: See the section "Machine-Dependent Arithmetic Functions".

logeqv &rest integers

Returns the bit-wise logical equivalence (also known as exclusive nor) of its arguments interpreted as bit vectors. If no argument is given, the result is -1, which is an identity for this operation. If the integers (bit-vectors) are interpreted as sets, this operation represents iterated pairwise equivalence. Thus, an even number of small positive integer arguments returns a negative integer, and an odd number of small positive arguments returns a positive integer.

Examples:

( logeqv ) => -1
( logeqv 5 ) => 5
( logeqv -3 4 ) => 6
( logeqv 9 2 ) => -12
( logeqv -3 4 9 2 ) => 13 ; ( logeqv 6 -12 ) => 13
( logeqv 1 ) => 1
( logeqv 1 2 ) => -4
( logeqv 1 2 4 ) => 7

See the function boole.

For a table of related items, see the section "Functions Returning Result of Bitwise Logical Operations".

logior &rest integers

Returns the bit-wise logical inclusive or of its arguments.

If no argument is given, the result is zero. This is an identity for this operation.

Examples:

( logior ) => 0
( logior -5 ) => -5
( logior 3 10 ) => 11
( logior 4 8 2 ) => 14

See the function boole.

For a table of related items, see the section "Functions Returning Result of Bitwise Logical Operations".
**zl:logior** `number &rest more-numbers`  
*Function*

Returns the bit-wise logical *inclusive or* of its arguments. At least one argument is required. Example:

```
(zl:logior #o4002 #o67) => #o4067
```

For a table of related items: See the section "Functions Returning Result of Bit-wise Logical Operations".

**sys:%logldb** `byte spec integer`  
*Function*

Like `ldb`, except that it loads out of fixnums, allowing a byte size of 32 bits of the fixnum, including the sign bit. `sys:%logldb` also loads out of bignums, allowing a byte size of 32 bits, including the sign bit. The result of `sys:%logldb` can be negative when the size of the byte specified by `bytespec` is 32.

The behavior of `sys:%logldb` depends on the size of fixnums, so functions using it might not work the same way on future implementations of Symbolics Common Lisp. Its name starts with "%" because it is more like machine-level subprimitives than other byte manipulation functions.

For a table of related items: See the section "Machine-Dependent Arithmetic Functions".

**lognand** `integer1 integer2`  
*Function*

This is a non-associative bit-wise logical operation and takes exactly two arguments. It returns the logical *not-and* of its two arguments interpreted as bit vectors.

Examples:

```
(lognand 6 12) => -5     ;(lognot 4) => -5
(lognand 1 4) => -1
(lognand 1 -4) => -1
(lognand -1 4) => -5
(lognand -1 -4) => 3
(lognand 2 6) => -3
```

See the function `boole`.

For a table of related items, see the section "Functions Returning Result of Bit-wise Logical Operations".

**lognor** `integer1 integer2`  
*Function*

This is a non-associative bit-wise logical operation and takes exactly two arguments. It returns the logical *not-or* of its two arguments.

Example:
\[(\text{lognor} \ 3 \ 10) \Rightarrow -12\]
\[(\text{lognor} \ 1 \ 4) \Rightarrow -6\]
\[(\text{lognor} \ 2 \ 6) \Rightarrow -7\]

See the function \texttt{boole}.

For a table of related items, see the section "Functions Returning Result of Bitwise Logical Operations".

\textbf{lognot \ integer} \hspace{1cm} \textit{Function}

Returns the logical complement of \texttt{integer} interpreted as a bit vector. This is the same as \texttt{logxor}ing \texttt{integer} with -1. If \texttt{integer} is interpreted as a set, this operation represents complementation.

Example:

\[(\text{lognot} \ 3456) \Rightarrow -3457\]
\[(\text{lognot} \ 0) \Rightarrow -1\]
\[(\text{lognot} \ 1) \Rightarrow -2\]
\[(\text{lognot} \ -1) \Rightarrow 0\]
\[(\text{lognot} \ -2) \Rightarrow 1\]

For a table of related items: See the section "Functions Returning Result of Bitwise Logical Operations".

\textbf{logorc1 \ integer1 integer2} \hspace{1cm} \textit{Function}

This is a non-associative bit-wise logical operation and takes exactly two arguments. It returns the logical or of the complement of \texttt{integer1} with \texttt{integer2}.

Examples:

\[(\text{logorc1} \ -1 \ 11) \Rightarrow 11\]
\[(\text{logorc1} \ 11 \ -1) \Rightarrow -1\]
\[(\text{logorc1} \ 1 \ 4) \Rightarrow -2\]
\[(\text{logorc1} \ 2 \ 6) \Rightarrow -1\]

See the function \texttt{boole}.

For a table of related items, see the section "Functions Returning Result of Bitwise Logical Operations".

\textbf{logorc2 \ integer1 integer2} \hspace{1cm} \textit{Function}

This is a non-associative bit-wise logical operation and takes exactly two arguments. It returns the logical or of \texttt{integer1} with the complement of \texttt{integer2}.

Examples:
(logorc2 -1 11) => -1
(logorc2 11 -1) => 11
(logorc2 1 4) => -5
(logorc2 2 6) => -5

See the function `boole`.

For a table of related items, see the section "Functions Returning Result of Bit-wise Logical Operations".

**logtest integer1 integer2**

Returns `t` if any of the bits designated by the 1's in `integer1` are 1's in `integer2` (that is, if there exists at least one non-negative integer `j`, such that bit `j` in `integer1` and bit `j` in `integer2` are both 1's).

Examples:

(logtest 10 4) => NIL
(logtest 9 1) => T
(logtest 11 3) => T

For a table of related items, see the section "Predicates for Testing Bits in Integers".

**logxor &rest integers**

Returns the bit-wise logical exclusive or of its arguments. If no argument is given, the result is zero. This is an identity for this operation.

Examples:

(logxor) => 0
(logxor 5) => 5
(logxor 3 4) => 7
(logxor 9 2) => 11
(logxor 3 4 9 2) => 12 ; (logxor 7 11) => 12

See the function `boole`.

For a table of related items, see the section "Functions Returning Result of Bit-wise Logical Operations".

**zl:logxor integer &rest more-integers**

Returns the bit-wise logical exclusive or of its arguments. At least one argument is required.

Example:

(zl:logxor #o2531 #o7777) => #o5246

For a table of related items: See the section "Functions Returning Result of Bit-wise Logical Operations" and see CLtL 221.
long-float

**Type Specifier**

long-float is the type specifier symbol for the predefined Lisp double-precision floating-point number type.

The type long-float is a subtype of the type float. In Symbolics Common Lisp, the type long-float is identical to the type double-float.

The type long-float is disjoint with the types short-float, and single-float.

Examples:

```lisp
(typep 0d0 'long-float) => T
(subtypep 'long-float 'double-float) => T and T ; subtype and certain
(commonp 1.5d9) => T
(equal-typep 'long-float 'double-float) => T
(sys:double-float-p 1.5d9) => T
```

See the section "Data Types and Type Specifiers".
See the section "Numbers".

long-float-epsilon

**Constant**

The value of this constant is the smallest positive floating-point number e of a format such that it satisfies the expression:

```lisp
(not (= (float 1 e) (+ (float 1 e) e)))
```

In Symbolics Common Lisp long-float-epsilon has the same value as double-float-epsilon, namely: 1.1102230246251568d-16.

long-float-negative-epsilon

**Constant**

The value of this constant is the smallest positive floating-point number e of a format such that it satisfies the expression:

```lisp
(not (= (float 1 e) (- (float 1 e) e)))
```

In Symbolics Common Lisp the value of long-float-negative-epsilon is the same as that of double-float-negative-epsilon, namely: 5.551115123125784d-17.

long-site-name

**Function**

Returns a string that is the full name of your site. This is the contents of the Pretty-name field in your site's namespace object.

The CLOE Runtime environment does not provide a uniform way to obtain a "site" designation. If the value of the variable cloe::*long-site-name* is nil, you are prompted to enter the correct values for your site. Initially, cloe::*long-site-name* is set to "CLOE-USER-SITE".
**loop** &rest forms  

**Macro**

**loop** is a Lisp macro that provides a programmable iteration facility. The Symbolics Common Lisp implementation of **loop** is an extension of the Common Lisp specification for this macro in Guy L. Steele's *Common Lisp: the Language*. The Symbolics Common Lisp version, **loop** is similar to the Zetalisp version, except that **loop** allows its body to be a sequence of lists, for example:

```
(let ((i 0))
  (loop
    (print i)
    (incf i)
    (when (> i 1) (return (values))))))
```

The general approach is that a form introduced by the word **loop** generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (**prologue**) code, a body that can be executed several times, and some exit (**epilogue**) code. Variables can be declared local to the loop. The features are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The **loop** form consists of a series of clauses, each introduced by a keyword symbol. Forms appearing in or implied by the clauses of a **loop** form are classed as those to be executed as initialization code, body code, and/or exit code; within each part of the template that **loop** fills in, they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side effects can be used, and one piece of code might depend on following another for its proper operation.

If entries are added to or deleted from the loop macro while **loop** is in progress, the results are unpredictable, with one exception: if the function calls **remhash** to remove the entry currently being processed by the body, or performs a **setf** of **gethash** on that entry to change the associated value, then those operations will have the intended effect.

Note that **loop** forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English.

**Compatibility Note:** The Symbolics Common Lisp version of this function allows you to control its iteration by using keywords. The version of **loop** as specified in **Cltl** does not allow atoms in the body of the loop.

---

**zl:loop** x &optional ignore  

**Macro**

A Lisp macro that provides a programmable iteration facility. **zl:loop** is obsolete; use **loop** instead.

The general approach is that a form introduced by the word **zl:loop** generates a single program loop, into which a large variety of features can be incorporated.
The loop consists of some initialization (prologue) code, a body that can be executed several times, and some exit (epilogue) code. Variables can be declared local to the loop. The features are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The `zl:loop` form consists of a series of clauses, each introduced by a keyword symbol. Forms appearing in or implied by the clauses of a `zl:loop` form are classed as those to be executed as initialization code, body code, and/or exit code; within each part of the template that `zl:loop` fills in, they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side effects can be used, and one piece of code might depend on following another for its proper operation.

Note that `zl:loop` forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English.

Here are some examples to illustrate the use of `zl:loop`.

`print-elements-of-list` prints each element in its argument, which should be a list. It returns `nil`.

```
(defun print-elements-of-list (list-of-elements)
  (zl:loop for element in list-of-elements
    do (print element))) => PRINT-ELEMENTS-OF-LIST
```

`gather-alist-entries` takes an association list and returns a list of the "keys"; that is, `((foo 1 2) (bar 259) (baz))` returns `(foo bar baz)`.

```
(defun gather-alist-entries (list-of-pairs)
  (zl:loop for pair in list-of-pairs
    collect (car pair))) => GATHER-ALIST-ENTRIES
```

`extract-interesting-numbers` takes two arguments, which should be integers, and returns a list of all the numbers in that range (inclusive) that satisfy the predicate `interesting-p`.

```
(defun extract-interesting-numbers (start-value end-value)
  (zl:loop for number from start-value to end-value
    when (interesting-p number) collect number)) => EXTRACT-INTERESTING-NUMBERS
```

`find-maximum-element` returns the maximum of the elements of its argument, a one-dimensional array. For Maclisp, `aref` could be a macro that turns into either `funcall` or `zl:arraycall` depending on what is known about the type of the array.

```
(defun find-maximum-element (an-array)
  (zl:loop for i from 0 below (array-dimension-n 1 an-array)
    maximize (aref an-array i))) => FIND-MAXIMUM-ELEMENT
```
**my-remove** is like the Lisp function **zl:delete**, except that it copies the list rather than destructively splicing out elements. This is similar, although not identical, to the **zl:remove** function.

```
(defun my-remove (object list)
  (zl:loop for element in list
    unless (equal object element) collect element))
=> MY-REMOVE
```

**find-frob** returns the first element of its list argument that satisfies the predicate **frobp**. If none is found, an error is generated.

```
(defun find-frob (list)
  (loop for element in list
    when (frobp element) return element
    finally (ferror nil "No frob found in the list ~S" list)))
=> FIND-FROB
```

In many of the clause descriptions, an optional *data-type* is shown. This is a slot reserved for data type declarations; it is currently ignored.

**future-common-lisp:loop** &rest *keywords-and-forms*  

*Macro*

The macro **future-common-lisp:loop** performs iteration by executing a series of forms one or more times. Loop keywords are symbols recognized by **future-common-lisp:loop**. The provide such capabilities as control of direction of iteration, accumulation of values inside the loop body, and evaluation of expressions that precede or follow the loop body.

For **future-common-lisp:loop** without clauses, each form is evaluated in turn from left to right. When the last form has been evaluated, then the first form is evaluated again, and so on, in a never-ending cycle. **future-common-lisp:loop** establishes an implicit block named **nil**. The execution of **future-common-lisp:loop** can be terminated explicitly, by using **return**, **throw** or **return-from**, for example.

The syntax and usage of **future-common-lisp:loop** is relatively complex. For complete information, see the section "Using **future-common-lisp:loop**".

**loop-finish**  

*Macro*

**(loop-finish)** causes the iteration to terminate "normally", the same as implicit termination by an iteration-driving clause, or by the use of **while** or **until** — the epilogue code (if any) is run, and any implicitly collected result is returned as the value of the **loop**. For example:

```
(loop for x in '(1 2 3 4 5 6)
  collect x
  do (cond ((= x 4) (loop-finish))))
=> (1 2 3)
```

This particular example would be better written as **until** (= x 4) in place of the **do** clause.
See the section "End Tests for **loop**".

**si:loop-named-variable** *keyword*

Function

Used when an iteration path function desires to make an internal variable accessible to the user. Call this function only from within an iteration path function. If *keyword* has been specified in a **using** phrase for this path, the corresponding variable is returned; otherwise, **gensym** is called and that new symbol returned. Within a given path function, this routine should only be called once for any given keyword.

If you specify a **using** preposition containing any keywords for which the path function does not call **si:loop-named-variable**, **loop** informs you of the error. See the section "Iteration Paths for **loop**".

**si:loop-tassoc** *token* **keyword-alist**

Function

The **assoc** variant of **si:loop-tequal**.

See the section "Defining Iteration Paths".

**si:loop-tequal** *token* *keyword*

Function

The **loop** token comparison function.

*token* is any Lisp object. *keyword* must be an atomic symbol. The function returns **t** if *token* and *keyword* represent the same token, comparing them in a manner appropriate for the implementation.

See the section "Defining Iteration Paths".

**si:loop-tmember** *token* **keyword-list**

Function

The **member** variant of **si:loop-tequal**.

See the section "Defining Iteration Paths".

**lower-case-p** *char*

Function

Returns **t** if *char* is a lowercase letter:

\[
\begin{align*}
(\text{lower-case-p} \, \#\text{\textbackslash a}) & \Rightarrow \mathbf{T} \\
(\text{lower-case-p} \, \#\text{\textbackslash A}) & \Rightarrow \mathbf{NIL}
\end{align*}
\]

For a table of related items, see the section "Character Predicates".

**lsh** *number* *count*

Function

Returns *number* shifted left *count* bits if *count* is positive or zero, or *number* shifted right \(|*count*|\) bits if *count* is negative. Zero bits are shifted in (at either end) to
fill unused positions. number and count must be fixnums. Since the result is also a
fixnum, bits shifted off either end are lost. (In some applications you might find
ash useful for shifting bignums.)

Note that like the Zetalisp functions whose name begins with the percent-sign (%),
lsh is machine-dependent.

Examples:

(lsh 4 1) => #o10
(lsh #o14 -2) => #o3
(lsh -1 1) => #o-2
(lsh -100 27) => -536870912 ;(ash -100 27) => -1342177280

For a table of related items: See the section "Machine-Dependent Arithmetic Func-
tions".

machine-instance

Function

Returns a string that is the name of your machine.

(machine-instance) => "WOMBAT"

This is the contents of the Host field in your machine's namespace object. See the
section "Why do you name machines and printers?".

machine-type

Function

Returns a string that identifies the kind of hardware you are using.

(machine-type) => "Symbolics 3620"

For the CLOE Developer,

(machine-type) => "Symbolics"

and for the CLOE Application Generator,

(machine-type) => "Intel"

machine-version

Function

Under Genera, returns the board-level hardware information about your machine.
This is the same as the information displayed by the Show Machine Configuration
command for your machine.

Under CLOE, returns a string indicating the current version of the machine for
current implementation. For example, for the CLOE Developer you might get
something like the following:

(machine-version) => "3640"

and for the CLOE Application Generator
(machine-version)
=="386"

macro name lambda-list &body body

Special Form

The primitive special form for defining macros. A macro definition looks like this:

(macro name (form env)
 body)

name can be any function spec. form and env must be variables. body is a sequence of Lisp forms that expand the macro; the last form should return the expansion. defmacro is usually preferred in practice.

macroexpand macro-call &optional env dont-expand-special-forms for-declares

Function

If macro-call is a macro form, macroexpand expands it repeatedly by making as many repeated calls to macroexpand-1 as required until it is not a macro form, and returns two values: the final expansion and t. Otherwise, it returns macro-call and nil. The optional env environment parameter conveys information about local macro definitions that are defined via macrolet. (See the section "Lexical Environment Objects and Arguments".)

Compatibility Note: The optional argument dont-expand-special-forms, is a Symbolics extension to Common Lisp, which prevents macro expansion of forms that are both special forms and macros. dont-expand-special-forms will not work in other implementations of Common Lisp including CLOE.

(defmacro nand (&rest args) '(not (and ,args)))

(macroexpand '((nand foo (eq bar baz) (> foo bar))))

==> (not (and foo (eq bar baz) (> foo bar)))

The following example shows the probable results of three calls to macroexpand-1 from within a call to macroexpand:

(defmacro and-op (op &rest args) '(',op ,args))

(macroexpand '((and-op or (eq bar baz) (> foo bar))))

==> (or (eq bar baz) (> foo bar))

(macroexpand-1 (and-op or (eq bar baz) (> foo bar)))

==> (cond ((eq bar baz)) (t (> foo bar))) t
(macroexpand-1 (cond ((eq bar baz)) (t (> foo bar))))
=> (if (eq bar baz) (eq bar baz) (> foo bar)) t

=> (if (eq bar baz) (eq bar baz) (> foo bar)) t

**macroexpand-1** *macro-call* &optional *env* *dont-expand-special-forms*  

*Function*

If *macro-call* is a macro form, **macroexpand-1** expands it (once) and returns the expanded form and t. Otherwise, it returns *macro-call* and *nil*. The optional *env* environment parameter is conveys information about local macro definitions as defined via macrolet.

( Defmacro nand (&rest args) `'(not (and ,args)))

(macroexpand-1 `'(nand foo (eq bar baz) (> foo bar)))

=> (not (and foo (eq bar baz) (> foo bar))) t

( Defmacro and-op (op &rest args) `'(,op ,args))

(macroexpand-1 `'(and-op or (eq bar baz) (> foo bar)))

=> (or (eq bar baz) (> foo bar)) t

(See the section "Lexical Environment Objects and Arguments".)

**Compatibility Note:** The optional argument *dont-expand-special-forms*, is a Symbolics extension to Common Lisp, which prevents macro expansion of forms that are both special forms and macros. *dont-expand-special-forms* will not work in other implementations of Common Lisp including CLOE. See the variable *macroexpand-hook*.

**macroexpand-hook**  

*Variable*

The value is used as the expansion interface hook by **macroexpand-1**. When **macroexpand-1** determines that a symbol names a macro, it obtains the expansion function for that macro. The value of *macroexpand-hook* is called as a function of three arguments: the expansion function, *form*, and *env*. The value returned from this call is the expansion of the macro call.

The initial value of *macroexpand-hook* is funcall, and the net effect is to invoke the expansion function, giving it *form* and *env* as its two arguments.

This special variable allows for more efficient interpretation of code, for example, by allowing caching of macro expansions. Such efficiency measures are unnecessary in compiled environments such as the CLOE runtime system.

**macro-function** *function*
Function

Tests whether its argument is the name of a macro. function should be a symbol. If function has a global function definition that is a macro definition, the expansion function (a function of two arguments, the macro-call form and an environment) is returned. The function macroexpand is the best way to invoke the expansion function.

If function has no global function definition, or has a definition as an ordinary function or as a special form but not as a macro, then nil is returned. In the following example, macro-function (before using funcall) tests an argument intended as a function.

(defun foo (function-arg arg-arg)
  (if (macro-function function-arg)
      (do-something-else arg-arg)
      (funcall function-arg arg-arg (cadr arg-arg)))))

Usually, macroexpand is used to expand a macro. However, in the following example of a highly simplified definition of macroexpand-1, we see how to expand a macro by using macro-function.

(defun simple-macroexpand-1(form)
  (let ((name (first form))
        (expander (macro-function name)))
    (if expander
      (values (funcall expander form) t)
      (values form nil))))

It is possible for both macro-function and special-form-p to be true of a symbol. This is so because it is permitted to implement any macro also as a special form for speed.

macro-function cannot be used to determine whether a symbol names a locally defined macro established by macrolet; macro-function can examine only global definitions.

z1:setf can be used with macro-function to install a macro as a symbol’s global function definition:

For example:

(z1:setf (macro-function symbol) fn)

The value installed must be a function that accepts two arguments, an entire macro call and an environment, and computes the expansion for that call. Performing this operation causes the symbol to have only that macro definition as a global function definition; any previous definition, whether as a macro or as a function, is lost.

macrolet macros &body body

Defines, within its scope, a macro. It establishes a symbol as a name denoting a macro, and defines the expander function for that macro. defmacro does this
globally; **macrolet** does it only within the (lexical) scope of its body. A macro so defined can be used as the car of a form within this scope. Such forms are expanded according to the definition supplied when interpreted or compiled.

The syntax of **macrolet** is identical to that of **flet** or **labels**: it consists of clauses defining local, lexical macros, and a body in which the names so defined can be used. **macros** a list of clauses each of which defines one macro. Each clause is identical to the cdr of a **defmacro** form: it has a name being defined (a symbol), a macro pseudo-argument list, and an expander function body.

The pseudo-argument list is identical to that used by **defmacro**. It is a pattern, and can use appropriate lambda-list keywords for macros, including &environment. See the section "Lexical Environment Objects and Arguments".

The following example of **macrolet** is for demonstration only. If the macro **square** needed to be open-coded, was long and cumbersome, or was used many times, then the use of **macrolet** would be suggested.

```
(defun square-coordinates (point)
  (macrolet ((square (x) '(* ,x ,x)))
    (setf (point-x point) (square (point-x point))
      (point-y point) (square (point-y point)))))
```

```
(defstruct point x y) => POINT
(setq p1 (make-point :x 3 :y 4)) => #S(POINT :X 3 :Y 4)
(square-coordinates p1) => 16
```

```
(defun foo (x)
  (macrolet ((do-it (var n)
    ' (case ,var
      , (do ((i 0 (+ i 1))
        (1 '()))
      ((= i n) (nreverse l))
        (push (list i (format nil "~R" i)
        1))))))
    (do-it x 100)))
```

```
(foo 12) => "twelve"
```

The following example implements a macro to establish a context where items can be added to the end of list. This is similar to the way **push** adds to the beginning of a list. We use **macrolet** to ensure that **push-onto-end** has access to the pointer until the last cons of the list.
(defmacro with-end-push2 (list &body body)
  (let (((lastptr (gensym)))
        '(let (((,lastptr (last ,list)))
            (macrolet ((push-onto-end (val)
                          (rplacd ,',lastptr
                            (setq ,',lastptr (cons ,val nil))))
                        ,body)))))

(defun example-3 ()
  (let ((mylist (list 1 2 3))
        (a-list (list 'a 'b 'c 'd))
        (with-end-push2 mylist
          (dolist (l a-list mylist)
            (push-onto-end l))))))

(example-3)

It is important to realize that macros defined by macrolet are run (when the compiler is used) at compile time, not run-time. The expander functions for such macros, that is, the actual code in the body of each macrolet clause, cannot attempt to access or set the values of variables of the function containing the use of macrolet. Nor can it invoke run-time functions, including local functions defined in the lexical scope of the macrolet by use of flet or labels. The expander function can freely generate code that uses those variables and/or functions, as well as other macros defined in its scope, including itself.

There is an extreme subtlety with respect to expansion-time environments of macrolet. It should not affect most uses. The macro-expander functions are closed in the global environment; that is, no variable or function bindings are inherited from any environment. This also means that macros defined by macrolet cannot be used in the expander functions of other macros defined by macrolet within the scope of the outer macrolet. This does not prohibit either of the following:

- Generation of code by the inner macro that refers to the outer one.
- Explicit expansion (by macroexpand or macroexpand-1), by the inner macro, of code containing calls to the outer macro. Note that explicit environment management must be utilized if this is done. See the section "Lexical Environment Objects and Arguments".


Creates and returns a new array. dimensions is the only required argument. dimensions is a list of integers that are the dimensions of the array; the length of the list is the dimensionality, or rank of the array.
Create a two-dimensional array
(make-array '(3 4) :element-type 'string-char)

You can use these element types: bit, string-char, (unsigned-byte 8), (unsigned-byte 16), (signed-byte 8), and (signed-byte 16).

For convenience when making a one-dimensional array, the single dimension can be provided as an integer rather than a list of one integer.

Create a one-dimensional array of five elements.
(make-array 5)

The initialization of the elements of the array depends on the element type. By default, the array is a general array, the elements can be any type of Lisp object, and each element of the array is initially nil. However, if the :element-type option is supplied, and it constrains the array elements to being integers or characters, the elements of the array are initially 0 or characters whose character code is 0 and style is NIL NIL NIL NIL. You can specify initial values for the elements by using the :initial-contents or :initial-element options.

Compatibility Note: The optional arguments :displaced-conformally, :area, :leader-list, :leader-length, and :named-structure-symbol are Symbolics extensions to Common Lisp, and are not available in CLOE.

For a table of related items: See the section "Basic Array Functions".

See the section "Examples of make-array".

If you are using CLOE, see the section "Keyword Options for make-array".


We recommend using make-array instead of zl:make-array. See the function make-array.

Creates and returns a new array. dimensions is the only required argument. dimensions is a list of integers that are the dimensions of the array; the length of the list is the dimensionality, or rank of the array. For the one-dimensional case you can just give the integer.

zl:make-array returns two values: the newly created array, and the number of words allocated in the process of creating the array. The second value is the sys:%structure-total-size of the array. Note that make-array returns only one value, the newly created array.

Most of the keyword options to zl:make-array have the same meaning as the keyword options with the same name that can be given to make-array. See the section "Keyword Options for make-array".

:initial-value The :initial-value keyword for zl:make-array has the same meaning as the :initial-element keyword for make-array.
The :type option for zl:make-array is used for the same purpose as is the :element-type option for make-array; that is, to specify that the elements of the array should be of a certain type. The value of the :type option is the symbolic name of one of the Zetalisp array types, which include:

- sys:art-q
- sys:art-q-list
- sys:art-nb
- sys:art-string
- sys:art-fat-string
- sys:art-boolean
- sys:art-fixnum

The default type of array is sys:art-q, a general array. See the section "Zetalisp Array Types".

The initialization of the elements of the array depends on the type of array. If the array is of a type whose elements can only be integers or characters, element of the array are initially 0 or character code 0. Otherwise, each element is initially nil.

zl:make-array-into-named-structure array

Function

Turns array into a named structure, and returns it.

make-char char &optional (bits 0) (font 0)

Function

Takes the argument char, which must be a character object. bits and font must be non-negative integers. make-char sets the bits field to bits and returns the new character. If make-char cannot construct a character given its arguments, it returns nil.

To set the bits of the character, supply one of the character bits constants as the bits argument. See the section "Character Bit Constants".

(make-char \A char-meta-bit) => \m-A

Since the value of char-font-limit is 1, the only valid value of font is 0, since Symbolics does not support font numbers. The only reason to use the font option would be when writing a program intended to be portable to other Common Lisp systems. Common Lisp supports font attributes for character objects, Symbolics Common Lisp does not.

In Genera, make-char does not change character styles. If you want to construct a new character with a specified character style, use make-character. See the function make-character.

For a table of related items, see the section "Making a Character".
**make-character** *char* &key (bits 0) (style nil)

Function

Takes an argument *char*, which must be a character object, and returns a new character with the same code, but having the specified bits and style.

To set the bits of the character, supply one of the character bits constants as the value of the :bits keyword. See the section "Character Bit Constants". For example:

```lisp
(make-character \1 :bits char-control-bit) => \c-1
```

To set the character style of the character, use the :style keyword and supply a list of the form (:family :face :size). Any of the elements of this list can be nil. For example:

```lisp
(make-character \A :style '(nil :italic nil)) => \A
```

For a table of related items, see the section "Making a Character".

**make-concatenated-stream** &rest *streams*

Function

Returns a stream that only works in the input direction. Input is taken from the first of the *streams* until it reaches EOF (end-of-file); then that stream is discarded, and input is taken from the next of the *streams*, and so on. If no arguments are given, the result is a stream with no content; any input attempt will result in EOF.

In the following example, three file input streams are created using open. These streams are read from inside a loop by using **make-concatenated-stream**.

```lisp
(with-open-file (stream1 *stream-name1* :direction :input)
  (with-open-file (stream2 *stream-name2* :direction :input)
    (with-open-file (stream3 *stream-name3* :direction :input)
      (let ((input '()))
        (cat-stream (make-concatenated-stream stream1 stream2 stream3))))
      (loop
        (setq input (read cat-stream nil :eof))
        (unless (and input (not (eq input :eof)))
          (return t))
        (process-input input)))))))
```

**make-condition** *condition-name* &rest init-options

Function

Creates a condition object of the specified *condition-name* with the specified init-options. This object can then be signalled by passing it to **signal** or **error**. Note that you are not supposed to design functions that indicate errors by returning error objects; functions should always indicate errors by signalling error objects. This function makes it possible to build complex systems that use subroutines to generate condition objects so that their callers can signal them.
For a table of related items available in Genera: See the section "Condition-Checking and Signalling Functions and Variables".

sys:make-coroutine-bidirectional-stream function &rest arguments  
This function is obsolete. See the function sys:open-coroutine-stream.

sys:make-coroutine-input-stream function &rest arguments  
This function is obsolete. See the function sys:open-coroutine-stream.

sys:make-coroutine-output-stream function &rest arguments  
This function is obsolete. See the function sys:open-coroutine-stream.

make-dispatch-macro-character char &optional non-terminating-p (a-readtable *readtable*)
Causes char to be a dispatching macro character in readtable. If non-terminating-p is non-nil (it defaults to nil), it will be a non-terminating macro character, which means that it may be embedded within extended tokens. make-dispatch-macro-character returns t.

Initially, every character in the dispatch table has a character-macro function that signals an error. Use set-dispatch-macro-character to define entries in the dispatch table.

(make-dispatch-macro-character #\[ t *readtable*)
(set-dispatch-macro-character #\[ #\ macfun)
(values (read-from-string "[\+]"))
=> #\+

make-dynamic-closure symbol-list function
Creates and returns a dynamic closure of function over the variables in symbol-list. Note that all variables on symbol-list must be declared special.

To test whether an object is a dynamic closure, use (typep x :closure). (typep x :closure) is equivalent to (zl:closurep x). See the section "Dynamic Closure-Manipulating Functions".

make-echo-stream input-stream output-stream

function
This function, which is part of the Common Lisp standard, is not currently available in the Symbolics implementation of Common Lisp under Genera.

In CLOE, `make-echo-stream` creates and returns an input/output stream that takes input from `input-stream`, echoes the input to `output-stream`, and sends output to `output-stream`.

```lisp
(with-open-file (instream "foo" :direction :input)
  (with-open-file (outstream "bar.out" :direction :output)
    (let ((my-io-stream (make-echo-stream instream outstream)))
      ...
      (my-decide *decider* (read my-io-stream)) ; these reads are echoed
      (if (eq *decision* 'sell)
          (sell-action (read my-io-stream)) ; to outstream.
      ))))
```

The function `make-echo-stream` is thus handy for implementing ‘dribble’ facilities, to record interactions.

### `zl:make-equal-hash-table` &rest options

Function

Creates a new hash table using the `equal` function for comparison of the keys. This function calls `make-instance` using the `si:equal-hash-table` flavor, passing `options` to `make-instance` as init options. See the flavor `si:equal-hash-table`. This function is obsolete; use `make-hash-table` with the :test keyword instead.


Function

Creates and returns a new table object. This function calls `make-instance` using a basic table flavor and mixins for the necessary additional flavors as specified by the options.

`make-hash-table` takes the following keyword arguments:

- **:name** A symbol that identifies the table in progress notes.
- **:test** Determines how keys are compared. Its argument can be any function; `eq` is the default. If you supply one of the following values or predicates the hash table facility automatically supplies a :hash-function: `eq`, `eql`, `equal`, `char-equal`, `char=`, `string-equal`, `#string-equal`, `string=`, `zl:equal`, `zl:string-equal`, `zl:string=`. If you supply a value or predicate that is not on this list, you must supply a :hash-function explicitly. Note: the :test and :hash-function interact closely, and must agree with each other.
:size An integer representing the initial size of the table. The table will be made large enough to hold this many entries without growing.

:area If :area is nil (the default), the *default-cons-area* is used. Otherwise, the number of the area that you wish to use. This keyword is a Symbolics extension to Common Lisp.

:hash-function Specifies a replacement hashing function. The default is based on the :test predicate. This keyword is a Symbolics extension to Common Lisp.

:rehash-before-cold Causes a rehash whenever the hashing algorithm has been invalidated, during a Save World operation. Thus every user of the saved world does not have to waste the overhead of rehashing the first time they use the table after cold booting.

For eq tables, hashing is invalidated whenever garbage collection or world compression occurs because the hash function is sensitive to addresses of objects, and those operations move objects to different addresses. For equal tables, the hash function is sensitive to addresses of some objects, but not to others. The table remembers whether it contains any such objects.

Normally a table is automatically rehashed “on demand” the first time it is used after hashing has become invalidated. This first gethash operation is therefore much slower than normal.

The :rehash-before-cold keyword should be used on tables that are a permanent part of your world, likely to be saved in a world saved by Save World, and to be touched by users of that world. This applies both to tables in Genera and to tables in user-written subsystems saved in a world.

This keyword is a Symbolics extension to Common Lisp.

:rehash-after-full-gc Similar to :rehash-before-cold. Causes a rehash whenever the garbage collector performs a full gc. This keyword is a Symbolics extension to Common Lisp.

:entry-size This keyword is obsolete. :entry-size 2 is equivalent to :number-of-values 1. :entry-size 1 is equivalent to :number-of-values 0. This keyword is a Symbolics extension to Common Lisp.

:number-of-values Specifies the number of values associated with the key to be stored in the table. Currently, the only valid values are 0 and 1. If 0 is specified, the table functions return t for the value of the entry. This keyword is a Symbolics extension to Common Lisp.

:store-hash-code Specifies that the table system store the hash code for each key with the key. This keyword makes make-hash-table run
faster, since its use avoids the need to run a test function, unless the hash codes are the same. Use of this keyword increases the size of the table. Since \texttt{gethash} searches for keys equivalent to the supplied key under the supplied value of the \texttt{:test} argument, \texttt{:store-hash-code t} improves performance if the \texttt{:test} function pages or is slow. This keyword is a Symbolics extension to Common Lisp.

\textbf{:mutating} \hfill\break

Turns mutation on and off. The overhead involved with specifying this keyword is relatively higher for small tables than for large ones. The default value is \texttt{t}. This keyword is a Symbolics extension to Common Lisp.

\textbf{:initial-contents} \hfill\break

Set the initial contents for the new table. It can be either a table object to be copied, or a sequence of keys and values, for example:

\begin{verbatim}
'(KEY1 VALUE1 ... KEYn VALUEn)
\end{verbatim}

This keyword is a Symbolics extension to Common Lisp.

\textbf{:locking} \hfill\break

One of the following locking strategies: \texttt{:process}, \texttt{:without-interrupts}, \texttt{nil}, or a cons consisting of a lock and an unlock function. The default is to lock against other processes. This keyword is a Symbolics extension to Common Lisp.

\textbf{:ignore-gc} \hfill\break

By default, if the hash function is sensitive to the garbage collector, the table is protected against GC flip. If you supply this keyword, the table is not protected.

If the hash function utilizes the address of a Lisp object that might be changed by the GC, the hash function must recompute the hash code if that address is changed. \texttt{:ignore-gc} asserts that the hash function never uses such addresses, so that it need not recompute the codes. The default depends on the hash function: if it's one of a small set of functions that Lisp knows do not depend on addresses, this defaults to \texttt{t} (meaning yes, it can ignore the GC). Otherwise, it chooses \texttt{nil}, which is always safe. \texttt{t} might make your program run faster (avoiding rehashes at GC time) but might also break your program (if the hash function depends on address values). This keyword is a Symbolics extension to Common Lisp.

\textbf{:gc-protect-values} \hfill\break

The default is \texttt{t}. If \texttt{nil}, table entries are automatically deleted if a value becomes unreachable other than through the table. This keyword is a Symbolics extension to Common Lisp.

\textbf{:growth-factor} \hfill\break

A synonym for \texttt{:rehash-size}. If the keyword is an integer, it is the number of entries to add, and if it is a floating-point number, it is the ratio of the new size to the old size. If the value is neither an integer or a floating-point number, an error is signalled. This keyword is a Symbolics extension to Common Lisp.
:growth-threshold  A synonym for :rehash-threshold. If it is an integer greater than zero and less than the :size, it is related to the number of entries at which growth should occur. The threshold is the current size minus the :growth-threshold. If it is a floating-point number between zero and one, it is the percentage of entries that can be filled before growth will occur. If the value is neither an integer or a floating-point number, an error is signalled. This keyword is a Symbolics extension to Common Lisp.

:rehash-size  The growth factor of the table when it becomes full. If the value of the keyword is an integer, it is the number of entries to add, and if it is a floating-point number, it is the ratio of the new size to the old size. If the value is neither an integer or a floating-point number, an error is signalled.

:rehash-threshold  How full the table can become before it must grow. If it is an integer greater than zero and less than the value of :size, it is related to the number of entries at which growth should occur. The threshold is the current size minus the :growth-threshold. If it is a floating-point number between zero and one, it is the percentage of entries that can be filled before growth will occur. If the value is neither an integer nor a floating-point number, an error is signalled.

If you are using CLOE, zl:make-hash-table returns a newly created hash table with size entries. Argument test must be eq, eq1 or equal expressed as either symbols or as the function-quoted objects. Argument rehash-size can be an integer that provides the number of entries to add, or a floating point number that indicates the portion of the previous size to grow the hash table. Argument rehash-threshold also may be an integer or floating point number, and indicates the maximum capacity of the hash table before it should grow.

(setq hash-table-1 (make-hash-table))

(setq hash-table-2
  (make-hash-table :size (* number-of-my-symbols 100)
                   :rehash-size 2.0
                   :rehash-threshold 0.8
                   :test 'eq))


For a table of related items: See the section "Table Functions".


Function
Creates a new hash table using the `eq` function for comparison of the keys. This function calls `make-instance` using the `si:eq-hash-table` flavor, passing `options` to `make-instance` as init options. See the flavor `si:eq-hash-table`.

This is obsolete; use `make-hash-table` with the `:test` keyword instead.

```
(make-instance flavor-name &rest init-options)
```

Creates and returns a new instance of the flavor named `flavor-name`, initialized according to `init-options`, which are alternating keywords and arguments. All `init-options` are passed to any methods defined for `make-instance`. If `compile-flavor-methods` has not been done in advance, `make-instance` causes the combined methods of a program to be compiled, and the data structures to be generated. This is sometimes called composing the flavor. `make-instance` also checks that the requirements of the flavor are met. Requirements of the flavor are
set up with these `defflavor` options: `:required-flavors`, `:required-methods`, `:required-init-keywords`, and `:required-instance-variables`.

`init-options` can include:

`:initable-instance-variable value` You can supply keyword arguments to `make-instance` that have the same name as any instance variables specified as `:initable-instance-variables` in the `defflavor` form. Each keyword must be followed by its initial value. This overrides any defaults given in `defflavor` forms.

`:init-keyword value` You can supply keyword arguments to `make-instance` that have the same name as any keywords specified as `:init-keywords` in the `defflavor` form. Each keyword must be followed by a value. This overrides any defaults given in `defflavor` forms.

`:allow-other-keys t` Specifies that unrecognized keyword arguments are to be ignored.

`:allow-other-keys :return` Specifies that a list of unrecognized keyword arguments are to be the second return value of `make-instance`. Otherwise only one value is returned, the new instance.

`:area number` Specifies the area number in which the new instance is to be created. Note that you can use the `:area-keyword` option to `defflavor` to change the `:area` keyword to `make-instance` to a keyword of your choice, such as `:area-for-instances`.

Any ancillary values constructed by `make-instance` (other than the instance itself) are constructed in whatever area you specify for them; this is not affected by using the `:area` keyword. For example, if you supply a variable initialization that causes consing, that allocation is done in whatever area you specify for it, not in this area. For example:

```
(defflavor foo ((foo-1 (make-array 100)))
  ()
)
```

In this example the array is consed in `sys:default-cons-area`.

`:area nil` Specifies that the new instance is to be created in the `sys:default-cons-area`. This is the default, unless the `:default-init-plist` option is used to specify a different default for `:area`.

If not supplied in the `init-options` argument to `make-instance`, the `:default-init-plist` option to the `defflavor` form is consulted for any default values for initable instance variables, init keywords, and the `:area` and `:allow-other-keys` options.
An alternative way to make instances is to use constructors. One advantage in using constructor functions is that they are much faster than using `make-instance`. You can define constructors by using the `:constructor` option; for more information, see the section "Complete Options for `defflavor`".

If you want to know what the allowed keyword arguments to `make-instance` are, use the Show Flavor Initializations command. See the section "Show Flavor Commands". `c-sh-f` works too, if the flavor name is constant.

You can define a method to run every time an instance of a certain flavor is created. For information, see the section "Writing Methods for `make-instance`".

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".
Creates and returns a new instance of the flavor named `flavor-name`, initialized according to `init-options`, which are alternating keywords and arguments. All `init-options` are passed to any methods defined for `make-instance`.

If `compile-flavor-methods` has not been done in advance, `make-instance` causes the combined methods of a program to be compiled, and the data structures to be generated. This is sometimes called composing the flavor. `make-instance` also checks that the requirements of the flavor are met. Requirements of the flavor are set up with these `defflavor` options: `:required-flavors`, `:required-methods`, `:required-init-keywords`, and `:required-instance-variables`.

`init-options` can include:

`:initable-instance-variable value`
You can supply keyword arguments to `make-instance` that have the same name as any instance variables specified as `:initable-instance-variables` in the `defflavor` form. Each keyword must be followed by its initial value. This overrides any defaults given in `defflavor` forms.

`:init-keyword value`
You can supply keyword arguments to `make-instance` that have the same name as any keywords specified as `:init-keywords` in the `defflavor` form. Each keyword must be followed by a value. This overrides any defaults given in `defflavor` forms.

`:allow-other-keys t` Specifies that unrecognized keyword arguments are to be ignored.

`:allow-other-keys :return` Specifies that a list of unrecognized keyword arguments are to be the second return value of `make-instance`. Otherwise only one value is returned, the new instance.

If not supplied in the `init-options` argument to `make-instance`, the `:default-init-plist` option to the `defflavor` form is consulted for any default values for initable instance variables, init keywords, and the `:allow-other-keys` options.

If you want to know what the allowed keyword arguments to `make-instance` are, use the Show Flavor Initializations command.

You can define a method to run every time an instance of a certain flavor is created:

```lisp
(clos:make-instance class &rest initargs)  Generic Function
```
Creates, initializes, and returns a new instance of the given class.

`class` The name of the class, or a class object.

`initargs` Alternating initialization argument names and values. The `initargs` are used to initialize the new instance. The set of valid initialization argument names includes:
• **clos:make-instance**, which does the following:

1. Checks the validity of the *initargs* and signals an error if an invalid initialization argument name is detected. See the section "Declaring Initargs for a Class".

2. Creates a new instance.

3. Calls the **clos:initialize-instance** generic function with the instance, and the initialization arguments provided to **clos:make-instance** followed by the default initialization arguments of the class. (This order of initialization arguments ensures that all initialization arguments provided to **clos:make-instance** are used to fill slots first, and then the default initialization arguments are used to fill slots that are still unbound.)

4. Fills any unbound slots with values according to the default initialization arguments of the class. The default initialization arguments are specified by the **:default-initargs** class option to **clos:defclass**.

5. When finished, returns the initialized instance.

The default primary method for **clos:initialize-instance** calls the **clos:shared-initialize** generic function with the instance, **t**, and the initialization arguments provided to **clos:initialize-instance**.

Note that the usual way for users to customize the initialization behavior is to specialize **clos:initialize-instance** by writing after-methods. Any applicable after-methods for **clos:initialize-instance** are called after the primary method for **clos:initialize-instance**. A user-defined primary method would override the default method, and thus could prevent the usual slot-filling behavior.

The default primary method for **clos:shared-initialize** does the following:

1. Fills slots with values according to the *initargs*. That is, for any initialization argument name that is associated with a slot, the value of the slot is initialized according to the argument given to **clos:make-instance**.
2. Fills any unbound slots indicated by the second argument to `clos:shared-initialize` with values according to the initform of the slot. The initform is specified by the :initform slot option to `clos:defclass`.

Users can define after-methods for `clos:shared-initialize`, to customize the initialization behavior that occurs in several cases. Note that a user-defined primary method for `clos:shared-initialize` would override the default method, and thus could prevent the usual slot-filling behavior. The `clos:shared-initialize` generic function is called in these cases:

- When an instance is first created; that is, when `clos:make-instance` is called.
- When an instance is reinitialized; that is, when `clos:reinitialize-instance` is called.
- When the class of an instance is changed; that is, when `clos:update-instance-for-different-class` is called.
- When a class is redefined; that is, when `clos:update-instance-for-redefined-class` is called.

Any slot that is not filled by `clos:shared-initialize` is left unbound.

The generic function `clos:make-instance` itself is not intended to be specialized by applications programmers. (Instead, it is intended to be specialized by meta-object programmers who wish to customize the behavior of `clos:make-instance` for a metaclass other than `clos:standard-class`).

### `clos:make-instances-obsolete` class

Generic Function

Called automatically when a class is redefined to trigger the updating of instances. Users can call `clos:make-instances-obsolete` to trigger the class redefinition process without actually redefining the class; the purpose of this would be to invoke the `clos:update-instance-for-redefined-class` generic function.

`class` The class whose instances should be updated. This can be the name of a class or a class object.

The modified class is returned.

### `make-list` size &key :initial-element :area

Function

Creates and returns a list containing size elements, each of which is initialized to the value supplied for the :initial-element keyword. The value of size should be a non-negative integer. For example:

```
(make-list 5) => (NIL NIL NIL NIL NIL)

(make-list 3 :initial-element 'rah) => (RAH RAH RAH)
```
:initial-element  The value to be assigned to each element of the created list. The default is nil).

:area  optional argument that is the number of the area in which to create the new list. (Areas are an advanced feature of storage management, and are not available in CLOE. See the section "Areas").

Compatibility Note: :area is a Symbolics extension to Common Lisp and is not available in CLOE.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**zl:make-list**  
*length*  
&key  *area*  :initial-value  
Function

Creates and returns a list containing *length* elements. *length* should be an integer. The keywords can be either of the following:

:area  Either an area number (an integer), or nil to mean the default area. The value specifies in which area the list should be created. Note that you cannot use this option in Cloe. See the section "Areas".

:initial-value  The initial value of all elements of the list. It defaults to nil.

zl:make-list always creates a cdr-coded list. See the section "Cdr-Coding". Examples:

(zl:make-list 3) => (nil nil nil)
(zl:make-list 4 :initial-value 7) => (7 7 7 7)

When zl:make-list was originally implemented, it took exactly two arguments: area and length. This obsolete form is still supported so that old programs will continue to work, but the new keyword-argument form is preferred.

For a table of related items: See the section "Functions for Constructing Lists and Conses" and see CLtL 267.

**clos:make-load-form**  
*object*  
Generic Function

Provides a way to use an instance of a user-defined CLOS class (that is, an instance whose metaclass is clos:standard-class or clos:structure-class) as a constant in a program compiled with compile-file. Users can define a method for clos:make-load-form that describes how an equivalent object can be reconstructed when the compiled-code file is loaded.

Compile-file calls clos:make-load-form on an object needed at load time, if the object’s metaclass is clos:standard-class. Compile-file will call clos:make-load-form only once for any given object (compared with eq) within a single file. If clos:make-load-form is called and no user-defined method is applicable, an error is signaled.
The argument \texttt{object} is an object needed at load-time.

\texttt{clos:make-load-form} returns two values. The first value, called the "creation form", is a form that, when evaluated at load time, should return an object that is equivalent to \texttt{object}.

The second value, called the "initialization form", is a form that, when evaluated at load time, should perform further initialization of the object. The value returned by the initialization form is ignored. If the \texttt{clos:make-load-form} method returns only one value, the initialization form is \texttt{nil}, which has no effect. If the object used as the argument to \texttt{clos:make-load-form} appears as a constant in the initialization form, at load time it will be replaced by the equivalent object constructed by the creation form; this is how the further initialization gains access to the object.

Both the creation form and the initialization form can contain references to instances of user-defined CLOS classes. However, there must not be any circular dependencies in creation forms. An example of a circular dependency is when the creation form for the object X contains a reference to the object Y, and the creation form for the object Y contains a reference to the object X. A simpler example would be when the creation form for the object X contains a reference to X itself.

Initialization forms are not subject to any restriction against circular dependencies, which is the entire reason that initialization forms exist. See the example of circular data structures below.

The creation form for an object is always evaluated before the initialization form for that object. When either the creation form or the initialization form references other objects of user-defined types that have not been referenced earlier in the \texttt{compile-file}, the compiler collects all of the creation and initialization forms. Each initialization form is evaluated as soon as possible after its creation form, as determined by data flow. If the initialization form for an object does not reference any other objects of user-defined types that have not been referenced earlier in the \texttt{compile-file}, the initialization form is evaluated immediately after the creation form. If a creation or initialization form \texttt{F} references other objects of user-defined types that have not been referenced earlier in the \texttt{compile-file}, the creation forms for those other objects are evaluated before \texttt{F}, and the initialization forms for those other objects are also evaluated before \texttt{F} whenever they do not depend on the object created or initialized by \texttt{F}. Where the above rules do not uniquely determine an order of evaluation, which of the possible orders of evaluation is chosen is unspecified.

While these creation and initialization forms are being evaluated, the objects are possibly in an uninitialized state, analogous to the state of an object between the time it has been created and it has been processed fully by \texttt{clos:initialize-instance}. Programmers writing methods for \texttt{clos:make-load-form} must take care in manipulating objects not to depend on slots that have not yet been initialized.

Examples:
In this example, an equivalent instance of **my-class** is reconstructed by using the values of two of its slots. The value of the third slot is derived from those two values.

Another way to write the last form in the above example is to use **clos:make-load-form-saving-slots**:

```
(defmethod make-load-form ((self my-class))
  (make-load-form-saving-slots self '(a b)))
```

In this example, instances of **my-frob** are "interned" in some way. An equivalent instance is reconstructed by using the value of the name slot as a key for searching existing objects. In this case the programmer has chosen to create a new object if no existing object is found; an alternative would be to signal an error in that case.

```
;; Example 3
(defclass tree-with-parent ()
  ((parent :accessor tree-parent)
   (children :initarg :children)))
(defmethod make-load-form ((x tree-with-parent))
  (values
   ;; creation form
   '(make-instance ',(class-of x)
     :children ,(slot-value x 'children))
   ;; initialization form
   '(setf (tree-parent ',x) ,(slot-value x 'parent))))
```

In this example, the data structure to be dumped is circular, because each parent has a list of its children and each child has a reference back to its parent. Suppose **clos:make-load-form** is called on one object in such a structure. The creation form creates an equivalent object and fills in the children slot, which forces cre-
At this point none of the parent slots have been filled in. The initialization form fills in the parent slot, which forces creation of an equivalent object for the parent if it was not already created. Thus the entire tree is recreated at load time. At compile time, `clos:make-load-form` is called once for each object in the tree. All the creation forms are evaluated, in unspecified order, and then all of the initialization forms are evaluated, also in unspecified order.

`clos:make-load-form-saving-slots object &optional save-slots`  
Function

Used in the bodies of methods for `clos:make-load-form`. The argument `object` is an object needed at load-time. The argument `save-slots` is a list of the names of the slots to preserve; it defaults to all of the local slots.

`clos:make-load-form-saving-slots` returns forms that construct an equivalent object using `clos:make-instance` and `setf` of `clos:slot-value` for slots with values, or `clos:slot-makunbound` for slots without values, or other functions of equivalent effect.

`clos:make-load-form-saving-slots` returns two values, thus it can deal with circular structures. `clos:make-load-form-saving-slots` works for instances of user-defined classes; that is, instances whose metaclass is `clos:standard-class` or `clos:structure-class`.

See the generic function `clos:make-load-form`.

`clos:make-method form`  
Macro

A list such as `(#:make-method form)` can be used instead of a method object as the first subform of `clos:call-method` or as an element of the second subform of `clos:call-method`.

`form`  
Specifies a method object whose method function has a body that is the given form. Note that `form` is not evaluated.

`make-package`  
Function

Makes a new package and returns it. `make-package` is the primitive subroutine called by `defpackage`. An error is signalled if the package name or nickname conflicts with an existing package. `make-package` takes the same arguments as `defpackage` except that standard `&key` syntax is used, and there is one additional keyword, `:invisible`. 
When an argument is called a name, it can be either a symbol or a string. When an argument is called a package, it can be the name of the package as a symbol or a string, or the package itself.

The keyword arguments are:

:use ‘(package package...)
  External symbols and relative name mappings of the specified packages are inherited. If only a single package is to be used, the name rather than a list of the name can be passed. If no package is to be used, specify nil. The default value for :use is cl.

(:nicknames name name...) for defpackage
:nicknames '(name name...) for make-package
  The package is given these nicknames, in addition to its primary name.

Compatibility Note: Symbolics Common Lisp under Genera provides additional functionality with these keywords, which are extensions to Common Lisp:

(:prefix-name name) for defpackage
:prefix-name name for make-package
  This name is used when printing a qualified name for a symbol in this package. You should make the specified name one of the nicknames of the package or its primary name. If you do not specify :prefix-name, it defaults to the shortest of the package’s names (the primary name plus the nicknames).

:invisible boolean
  If true, the package is not entered into the system’s table of packages, and therefore cannot be referenced via a qualified name. This is useful if you simply want a package to use as a data structure, rather than as the package in which to write a program.

(:shadow name name...) for defpackage
:shadow '(name name...) for make-package
  Symbols with the specified names are created in this package and declared to be shadowing.

(:export name name...) for defpackage
:export '(name name...) for make-package
  Symbols with the specified names are created in this package, or inherited from the packages it uses, and declared to be external.

(:import symbol symbol...) for defpackage
:import '(name name...) for make-package
  The specified symbols are imported into the package. Note that unlike :export, :import requires symbols, not names; it matters in which package this argument is read.

(:shadowing-import symbol symbol...) for defpackage
:shadowing-import '(symbol symbol...) for make-package

The same as :import but no name conflicts are possible; the symbols are declared to be shadowing.

(:import-from package name name...) for defpackage

:import-from '(package name name...) for make-package

The specified symbols are imported into the package. The symbols to be imported are obtained by looking up each name in package.

(defpackage only) This option exists primarily for system bootstrapping, since the same thing can normally be done by :import. The difference between :import and :import-from can be visible if the file containing a defpackage is compiled; when :import is used the symbols are looked up at compile time, but when :import-from is used the symbols are looked up at load time. If the package structure has been changed between the time the file was compiled and the time it is loaded, there might be a difference.

(:relative-names (name package) (name package)...) - defpackage

:relative-names '((name package) ...) - make-package

Declares relative names by which this package can refer to other packages. The package being created cannot be one of the packages, since it has not been created yet. For example, to be able to refer to symbols in the common-lisp package print with the prefix lisp: instead of cl: when they need a package prefix (for instance, when they are shadowed), you would use :relative-names like this:

(defun package (use cl)
  (:shadow error)
  (:relative-names (lisp common-lisp)))

(let ((*package* (find-package 'my-package)))
  (print (list 'my-package::error 'cl:error)))

(:relative-names-for-me (package name) ...) for defpackage

:relative-names-for-me '((package name) ...) for make-package

Declares relative names by which other packages can refer to this package.

(defpackage only) It is valid to use the name of the package being created as a package here; this is useful when a package has a relative name for itself.

(:size number) for defpackage

:size number for make-package

The number of symbols expected in the package. This controls the initial size of the package's hash table. You can make the :size specification an underestimate; the hash table is expanded as necessary.

(:hash-inherited-symbols boolean) for defpackage

:hash-inherited-symbols boolean for make-package

If true, inherited symbols are entered into the package's hash table to speed up symbol lookup. If false (the default), looking up a symbol in this package searches the hash table of each package it uses.
(:external-only boolean) for defpackage
:external-only boolean for make-package

If true, all symbols in this package are external and the package is locked.
This feature is only used to simulate the old package system that was used
before Release 5.0. See the section "External-only Packages and Locking".

(:include package package...) for defpackage
:include '(package package...) for make-package

Any package that uses this package also uses the specified packages. Note
that if the :include list is changed, the change is not propagated to users
of this package. This feature is used only to simulate the old package sys-
tem that was used before Release 5.0.

(:new-symbol-function function) for defpackage
:new-symbol-function function for make-package

function is called when a new symbol is to be made present in the package.
The default is si:pkg-new-symbol unless :external-only is specified. Do not
specify this option unless you understand the internal details of the package
system.

(:colon-mode mode) for defpackage
:colon-mode mode for make-package

If mode is :external, qualified names mentioning this package behave dif-
ferently depending on whether ;" or ;::" is used, as in Common Lisp ;;";
names access only external symbols. If mode is :internal, ;" names access
all symbols. :external is the default. See the section "Specifying Internal
and External Symbols in Packages".

(:prefix-intern-function function) for defpackage
:prefix-intern-function function for make-package

The function to call to convert a qualified name referencing this package
with ;" (rather than ;::") to a symbol. The default is intern unless (:colon-
mode :external) is specified. Do not specify this option unless you under-
stand the internal details of the package system.

make-plane rank &key (:type 'sys:art-q) :default-value (:extension 32) :initial-
dimensions :initial-origins

Function

Creates and returns a plane. rank is the number of dimensions. options is a list of
alternating keyword symbols and values. The allowed keywords are:

:type The array type symbol (for example, sys:art-1b) specifying the type of the
array out of which the plane is made.

:default-value
The default component value.
:extension
The amount by which to extend the plane. See the section "Planes".

:initial-dimensions
A list of dimensions for the initial creation of the plane. You might want to use this option to create a plane whose first dimension is a multiple of 32, so you can use bitblt on it. The default is 1 in each dimension.

:initial-origins
A list of origins for the initial creation of the plane. The default is all zero.

Example:
(make-plane 2 :type sys:art-4b :default-value 3)
creates a two-dimensional plane of type sys:art-4b, with default value 3.

For a table of related items, see the section "Operations on Planes".

make-random-state &optional state

Returns a new object of type random-state, which the function random can use as its state argument.

If state is nil or omitted, make-random-state returns a copy of the current random-number state object (the value of variable *random-state*).

If state is a state object, a copy of that state object is returned.

If state is t, the function returns a new state object that has been "randomly" initialized.

Examples:
(setq x (make-random-state)) => #.(RANDOM-STATE 71 1695406379...)
;; the value of x is now a random state
(setq copy-x (make-random-state x)) => #.(RANDOM-STATE 71...)
;; this makes a copy of random state x
;; a way to get reproducibly random numbers
(equalp (make-random-state t) *random-state*) => nil

For a table of related items, see the section "Random Number Functions".


Makes rasters; this should be used instead of make-array when making arrays that are rasters. make-raster-array is similar to make-array, but make-raster-array takes width and height as separate arguments instead of taking a single dimensions argument. If the raster is to be used with bitblt, the width times the number of bits per array element must be a multiple of 32.
The *make-array-options* are the options that can be given to *make-array*. For information on those options: See the section "Keyword Options for *make-array*".

When you cannot use *make-raster-array*, for example from the :*make-array* option to *defstruct* constructors, you should use *raster-width-and-height-to-make-array-dimensions* instead.

For a table of related items: See the section "Operations on Rasters".

### zl:*make-raster-array* width height &rest make-array-options

This function is provided for compatibility with previous releases. Use the Common Lisp function, *make-raster-array*.

### make-sequence type size &key :initial-element :area

Returns a sequence of type *type* and of length *size*, each of whose elements has been initialized to the value of the :*initial-element* argument (or nil if none is specified). If :*initial-element* is specified, the value must be an object that can be an element of a sequence of type *type*. For example:

```
(make-sequence '(vector double-float) 5 :initial-element 1d0)
=> #(1.0d0 1.0d0 1.0d0 1.0d0 1.0d0)
```

```
(make-sequence 'list 4 :initial-element 'a) => (a a a a)
```

```
(make-sequence 'string 4 :initial-element #\a) => "aaaa"
```

If :*initial-element* is a fat character, under Genera, *make-sequence* makes a fat string (a string of element type *character*).

The keyword :*area* is the number of the area in which to create the new alist. (Areas are an advanced feature of storage management.) :*area* is a Symbolics extension to Common Lisp, and is not supported in CLOE. See the section "Areas". See the function *vector*. See the function *make-list*.

For a table of related items: See the section "Sequence Construction and Access".

### make-string size &key :initial-element :element-type :area

Returns a simple string of length *size*. It constructs a one-dimensional array without fill pointer or displacement, to hold elements of type *character*, or any of its subtypes, that is, *string-char*, or *standard-char*. Depending on their character type, Genera strings created with *make-string* can therefore be either fat or thin. When using CLOE, strings made with *make-string* always have elements of type *string-char*.

The ability to create fat as well as thin strings represents an extension of the *make-string* function as presented in Guy L. Steele’s *Common Lisp: the Language*. 
The optional keywords are as follows:

:initial-element Each element of the new array is initialized to the character specified by this keyword; this character must correspond to the type specified by :element-type, if any. If no initial element is specified, array elements are initialized to characters with a char-code of 0, whose type corresponds to the type specified by :element-type; if :element-type is also unspecified, make-string builds a thin string.

:element-type Specifies the type of characters in the string and if you are using Genera, must be of type character, or any of its subtypes. If this keyword is left unspecified, the string type corresponds to the type of the character specified in :initial-element. If both keywords are omitted, make-string builds a thin string. :element-type is a Symbolics extension to Common Lisp, and not available when using CLOE.

:area Specifies the area in which to create the array. :area should be an integer or nil to mean the default area. :area is a Symbolics extension to Common Lisp, and not available under CLOE.

The examples below show the interaction of the keywords :initial-element and :element-type.

Since make-string only lets you build simple character arrays, you must use the array-specific function make-array to build more complex character arrays.

Examples:

; :initial-element and :element-type are omitted. String is thin.
(string-char-p (char (make-string 5) 1)) => T

; :initial-element and :element-type specify a thin string.
(string-char-p (char (make-string 5 :initial-element #\C
                                :element-type 'string-char) 0)) => T

; :initial-element and :element-type specify a fat string.
(string-fat-p (make-string 5 :initial-element #\hyper-C
                                :element-type 'character)) => T

; :element-type is omitted, and :initial-element
; is a standard character. String is thin.
(string-char-p (char (make-string 5 :initial-element #\a) 2)) => T

; :element-type is omitted, and :initial-element
; is a fat character. String is fat.
(string-fat-p (make-string 3 :initial-element #\hyper-super-a)) => T
; :initial-element is omitted and
; :element-type is a subtype of character. String is thin.
(string-fat-p (make-string 4 :element-type 'string-char)) => NIL

; :initial-element is omitted and
; :element-type is of type character. String is fat.
(string-fat-p (make-string 4 :element-type 'character)) => T

(make-array 5 :element-type 'string-char) => "DDDDD"
; returns a simple, thin string

(make-array 3 :element-type 'character :initial-element #\hyper-super-q
=> "<H-S-Q><H-S-Q><H-S-Q>" ; returns a fat, simple string

(make-string 4 :area working-storage-area) => "DDDD"

Under CLOE, make-string always creates a simple string. If an adjustable string
or a string with a fill-pointer is required, use make-array instead of make-string.
The following call to make-array creates a non-simple string.

(setq str (make-array 10
    :element-type 'string-char
    :fill-pointer 0
    :adjustable t))

=> ""

(push #\a str)
(push #\b str)
(push #\c str)

(char str 2) => #\c

For a table of related items: See the section "String Construction".

make-string-input-stream string &optional (start 0) end

Function

Returns an input stream. The input stream will supply, in order, the characters in
the substring of string delimited by start and end. After the last character has
been supplied, the stream will then be at end-of-file.

(make-string-input-stream "Hello")
=> #<LEXICAL-CLOSURE CLI::STRING-INPUT-STREAM 10223204>

(make-string-input-stream "Hello" 1 3)
=> #<LEXICAL-CLOSURE CLI::STRING-INPUT-STREAM 10224324>
(defvar input-string "foo bar baz")
(let ((my-stream
  (make-string-input-stream
    input-string
    (position-if #'alphanumericp input-string))))
  (read-char my-stream))

=> #\f

Often it is preferable to use with-input-from-string.

make-string-output-stream

Function

Returns an output stream that will accumulate all string output given it for the benefit of the function get-output-stream-string.

(setq stream (make-string-output-stream))
  => #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 44310040>

(setq output-string 'hello) => HELLO

(write output-string :stream stream) => HELLO

(get-output-stream-string stream) => "HELLO"

(defvar *heading* '("Name " "Rank " "Serial-number "))

(let ((my-stream (make-string-output-stream))
  (list-of-strings *heading*))
  (dolist (str list-of-strings)
    (princ str my-stream))
  (get-output-stream-string my-stream))
  => "Name Rank Serial-number ">

Often it is more convenient to use with-output-to-string.

make-symbol print-name &optional permanent-p

Function

Creates a new uninterned symbol whose print-name is the string print-name. The value and function bindings are unbound and the property list is empty.

Symbolics Common Lisp provides the optional argument permanent-p. If permanent-p is specified, it is assumed that the symbol is going to be interned and probably kept around forever; in this case it and its print-name are put in the proper areas. If permanent-p is nil (the default), the symbol goes in the default area and print-name is not copied. permanent-p is mostly for the use of intern itself and might not work in other implementations of Common Lisp.

Examples:
Note that the symbol is not interned; it is simply created and returned.

If a symbol has lowercase characters in its print-name, the printer quotes the name using slashes or vertical bars. The vertical bars inhibit the Lisp reader's normal action, which is to convert a symbol to uppercase upon reading it. See the section "What the Printer Produces".

Example:

```
(setq a (make-symbol "Hello")) ; => Hello
(princ a) ; prints out Hello
```

See the section "Functions for Creating Symbols".

### `make-syn-stream` symbol

Creates and returns a "synonym stream" (syn for short). `symbol` can be either a symbol or a locative.

If `symbol` is a symbol, the synonym stream is actually an uninterned symbol named `#symbol-syn-stream`. This generated symbol has a property that declares it to be a legitimate stream. This symbol is the value of `symbol`'s `si:syn-stream` property, and its function definition is forwarded to the value cell of `symbol` using a `sys:dtp-external-value-cell-pointer`. Any operations sent to this stream are redirected to the stream that is the value of `symbol`.

If `symbol` is a locative, the synonym stream is an uninterned symbol named `#:syn-stream`. This generated symbol has a property that declares it to be a legitimate stream. The function definition of this symbol is forwarded to the cell designated by `symbol`. Any operations sent to this stream are redirected to the stream that is the contents of the cell to which `symbol` points.

Synonym streams should not be passed between processes, since the streams to which they redirect operations are specific to a process.

### `make-synonym-stream` stream-symbol

Creates and returns a new stream that reads or writes indirectly to the stream denoted by `stream-symbol`. If the dynamic variable `stream-symbol` is rebound to a new stream, the operations of the synonym stream are redirected to the newly bound stream.

Under CLOE, the following streams are initially all bound to synonym-streams which do input or output via `*terminal-io*`, `*standard-input*`, `*standard-output*`, `*error-output*`, `*trace-output*`, `*query-io*`, and `*debug-io*`.

Under CLOE/Runtime, in the following example, `my-output-stream` is bound to a synonym stream using the variable `*file-stream*`. This variable is initially bound to `*standard-output*`. As the value of `*file-stream*` changes, the output of the synonym stream is directed to the stream currently the value of `*file-stream*`. 
(defvar *file-stream* *standard-output*)
(setq my-output-stream
    (make-synonym-stream '*file-stream*))

(with-open-file (*file-stream* *out-file-name1* :direction :output)
    (process data)
    (format my-output-stream stuff))

(setq *file-stream* *standard-output*)
(format my-output-stream more-stuff)

(with-open-file (*file-stream* *out-file-name2* :direction :output)
    (process data)
    (format my-output-stream other-stuff))

**make-two-way-stream** *input-stream output-stream*  
*Function*

Returns a bidirectional stream that gets its input from *input-stream* and sends its output to *output-stream*.

(with-open-stream (stream1 *stream-name1* :direction :input
    :element-type 'character)
    (with-open-stream (stream2 *stream-name2* :direction :output
    :element-type 'character)
    (let ((input '()))
        (two-way (make-two-way-stream stream1 stream2)))
    (loop
        (setq input (read-char two-way nil :eof))
        (unless (and input (not (eq input :eof)))
            (return t))
        (write-char input two-way))))

**zl:maknam** *charl*  
*Function*

Returns an uninterned symbol whose print-name is a string made up of the characters in *charl*. This function is provided mainly for Maclisp compatibility.

Examples:

(zl:maknam '(a b #\0 d)) => #:AB0D
(zl:maknam '(1 2 #\ "b") => #:1↓\b

**makunbound** *symbol*  
*Function*

Causes *symbol* to become unbound, and returns its argument. Example:
(setq a 1)
a => 1
(makunbound 'a)
a => causes an error.

Other examples:
(defvar *alarms*)
(boundp '*alarms*) => nil
(setq *alarms* 20)
(boundp '*alarms*) => t
(makunbound '*alarms*)
(boundp '*alarms*) => nil

See the section "Functions Relating to the Value of a Symbol".

makunbound-globally var

Function

Works like makunbound but sets the global value regardless of any bindings currently in effect.

makunbound-globally operates on the global value of a special variable; it bypasses any bindings of the variable in the current stack group. It resides in the global package.

makunbound-globally does not work on local variables. See the section "Functions Relating to the Value of a Symbol".

makunbound-in-closure closure symbol

Function

Makes symbol be unbound in the environment of closure; that is, it does what makunbound would do if you restored the value cells known about by closure. If symbol is not closed over by closure, this is just like makunbound. See the section "Dynamic Closure-Manipulating Functions".

map result-type function sequence &rest more-sequences

Function

Applies function to sequences, and returns a new sequence such that element j of the new sequence is the result of applying function to element j of each of the argument sequences. The returned sequence is as long as the shortest of the input sequences. function must take at least as many arguments as there are sequences provided, and at least one sequence must be provided.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:

(map 'list #'-'(4 3 2 1) '(3 2 1 0)) => (1 1 1 1)

(map 'string #(lambda (x) (if (oddp x) '#1 '#0)) '(1 2 3 4)) =>
"1010"
If *function* has side effects, it can count on being called first on all of the elements with index 0, then on all of those numbered 1, and so on.

The type of the result sequence is specified by the argument *result-type* (which must be a subtype of the type sequence), as for the function *coerce*. In addition, you can specify *nil* for the result type, meaning that no result sequence is to be produced. In this case *function* is invoked only for effect, and *map* returns *nil*. This gives an effect similar to *mapc*.

In the following example, *map* is used to define a function which works like *pairlis*, but takes vectors as input.

```lisp
(defun make-alist-from-vectors (vector1 vector2)
  (map 'list #'cons vector1 vector2))

(make-alist-from-vectors '(first second third) '(1 2 3))
=> ((FIRST . 1) (SECOND . 2) (THIRD . 3))
```

For a table of related items: See the section "Mapping Functions".

For a table of related items: See the section "Mapping Sequences".

---

**zl:map** fcn list &rest more-lists  Function

The Common Lisp function, *mapl*, is preferred.

Applies *fcn* to *list* and to successive sublists of that list. If all the lists are not of the same length, the iteration terminates when the shortest list runs out, and excess sublists of it are ignored.

**zl:map** works like *maplist*, except that it does not construct a list to return. Use **zl:map** when the *fcn* is being called merely for its side effects, rather than its returned values.

Examples:

```lisp
(zl:map #'equal '(2 3 4) '(2 3 4)) => (2 3 4)
(zl:map #'(lambda (x y) (if (equal x y)(princ "equal ")))(2 3 4) '(2 3 4))
=> equal equal equal
(zl:map #'(lambda (x) (if (member (car x) (cdr x)) nil (princ (car x))) (princ " "))(a b a c b)) => A C B (A B A C B)
```

For a table of related items: See the section "Mapping Functions".

For a table of related items: See the section "Mapping Sequences".

---

**:map-hash** function &rest args  Message
For each entry in the hash table, calls function on the key of the entry and the value of the entry. If args are supplied, they are passed along to function following the value of the entry argument. This message is obsolete; use maphash instead.

**map-into** result-sequence function sequence &rest more-sequences  

Function

Destructively modifies the result-sequence to contain the results of applying function to each element in the argument sequences in turn. The modified result-sequence is returned.

map-into differs from map in that it modifies an existing sequence rather than creating a new one.

The arguments result-sequence and sequences can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

function must take at least as many arguments as there are sequences provided, and at least one sequence must be provided.

For example:

```lisp
(setf n-list (list "12345"))
=> ("12345")
```

```lisp
(map-into n-list #'parse-integer n-list)
=> (12345)
```

If function has side effects, it can count on being called first on all of the elements with index 0, then on all of those numbered 1, and so on.

The function is applied to the minimum of the length of result-sequence and the shortest sequence, and if the result is longer than the shortest sequence, the remaining elements are not changed.

For tables of related items:

See the section "Mapping Functions".
See the section "Mapping Sequences".

**zl:mapatoms** function &optional (pkg *package*) (inherited-p t)  

Function

Applies function to each of the symbols in package. function should be a function of one argument. If inherited-p is t, this is all symbols accessible to package, including symbols it inherits from other packages. If inherited-p is nil, function only sees the symbols that are directly present in package.

Note that when inherited-p is t symbols that are shadowed but otherwise would have been inherited are seen; this slight blemish is for the sake of efficiency. If this is a problem, function can try zl:intern in package on each symbol it gets, and ignore the symbol if it is not eq to the result of zl:intern; this measure is rarely needed.
**zl:mapatoms-all** function

Applies *function* to all of the symbols in all of the packages in existence, except for invisible packages. *function* should be a function of one argument. Note that symbols that are present in more than one package are seen more than once.

Example:

```
(zl:mapatoms-all
 (function
  (lambda (x)
    (and (alphalessp 'z x)
         (print x))))
```

**mapc** *fcn* list &rest more-lists

Like *mapcar*, except that it does not return any useful value.

*mapc* applies *fcn* to successive elements of the argument lists. If the lists are not of the same length, the iteration terminates when the shortest list runs out.

*fcn* must take as many arguments as there are lists.

*mapc* is used when *fcn* is being called merely for its side effects, rather than its returned values.

Examples:

```
(mapc #'set '(A B C) '(11 22 33))
=> (A B C)

(mapc #'(lambda (x y) (if (= (+ x y) 3) (princ "three ")))
     '(1 2 3) '(2 1 3))
=> three three (1 2 3)

(mapc #'(lambda (x) (setf (get x 'color) t)) '(red blue green yellow))

(get 'red 'color) => T
```

For a table of related items: See the section "Mapping Functions".

**mapcan** *fcn* list &rest more-lists

Applies *fcn* to *list* and to successive elements of that list. This function is like *mapcar*, except that it combines the results of the function using *nconc* instead of *list*.

*fcn* must take as many arguments as there are lists.

Examples:
(mapcan #'(lambda (x) (if (equal x 3) nil (princ x))) '(1 2 3 4)) => 124NIL

(mapcan #'(lambda (x) (and (integerp x) (list x))) '(1 2.3 3. 4 'd 0)) => (1 3 4 0)

If mapcar were used for the above example, the result would be as follows:

(mapcar #'(lambda (x) (if (equal x 3) nil (princ x))) '(1 2 3 4)) => 124(1 2 NIL 4)

(mapcar #'(lambda (x) (and (integerp x) (list x))) '(1 2.3 3. 4 'd 0)) => ((1) NIL (3) (4) NIL (0))

(mapcar #'(lambda (x) (if (integerp x) (cons x nil)) (list 'a 3 'b 4 2)) => (3 4 2)

For a table of related items: See the section "Mapping Functions".

mapcar fcn list &rest more-lists

fcn is a function that takes as many arguments as there are lists in the call to mapcar. For example, since expt takes two arguments the following use of mapcar is incorrect:

Wrong:

(mapcar #'expt '(1 2 3 4 5) '(43 2 1 4 2) '(2 3 2 3 2))

Right:

(mapcar #'expt '(1 2 3 4 5) '(43 2 1 4 2))

In the correct example, mapcar calls expt repeatedly, each time using successive elements of the first list as its first argument and successive elements of the second list as its second argument. Thus, mapcar calls expt with the arguments 1 and 43, 2 and 2, 3 and 1, 4 and 4, and 5 and 2 and returns a list of the five results.

Examples:

(mapcar #'- '(3 4 2 5) '(1 1 2 3)) => (2 3 0 2)

(mapcar #'= '(1 2 3 4) '(1 2 3 8)) => (T T T NIL)

(mapcar #'(lambda (x) (if (numberp x) 0 1)) '(1 2 3 'k "hi" 'fly)) => (0 0 0 1 1 1)

(mapcar #'list '( 'hot 'cat 'sam 'new) '( 'dog 'hat 'man 'york)) => (('HOT 'DOG) ('CAT 'HAT) ('SAM 'MAN) ('NEW 'YORK))
(mapcar #'+ '(1 2 3 4) (circular-list 1)) => (2 3 4 5)

(mapcar #'= '(1 2 3 4 5) '(2 2)) => (NIL T)

(mapcar #'+ '(5 25 33)) => (6 26 34)

For a table of related items: See the section "Mapping Functions".

**mapcon** \( fcn \) \( list \) &rest more-lists

*Function*

Applies \( fcn \) to \( list \) and to successive sublists of that list rather than to successive elements.

This function is like **maplist**, except that it combines the results of the function using \( \text{nconc} \) instead of \( \text{list} \).

\( fcn \) must take as many arguments as there are lists.

**mapcon** could have been defined by:

```
(defun mapcon (f x y)
  (apply 'nconc (maplist f x y)))
```

Of course, this definition is less general than the real one.

Examples:

```
(mapcon #'(lambda (x y) (and (equal y x)(list x)) )
  '(yo 'ho 'woo 'wa) '(hi 'ho 'woo 'wa))
=> ('HO 'WOO 'WA) ('WOO 'WA) ('WA))
```

If **maplist** were used for the above example the result would look as follows:

```
(maplist #'(lambda (x y) (and (equal y x)(list x)) )
  '(yo 'ho 'woo 'wa) '(hi 'ho 'woo 'wa))
=> (NIL (((HO 'WOO 'WA)) ('WOO 'WA)) ('WA)))
```

```
(mapcon #'(lambda (x) (list (length x) x) (list 'a 'b 'c))
  => (3 (A B C) 2 (B C) 1 (C))
```

For a table of related items: See the section "Mapping Functions".

**maphash** \( \text{function table} \)

*Function*

For each entry in \( \text{table} \), calls \( \text{function} \) on the key of the entry and the value of the entry. If entries are added to or deleted from the hash table while a **maphash** is in progress, the results are unpredictable, with one exception: if the function calls **remhash** to remove the entry currently being processed by the function, or performs a **setf** of **gethash** on that entry to change the associated value, then those operations will have the intended effect. In the following example, **maphash** returns \( \text{nil} \).
;; alter every entry in MY-HASH-TABLE, replacing the value with its square root. Entries with negative values are removed.
(maphash #'(lambda (key val)
              (if (minusp val)
                  (remhash key my-hash-table)
                (setf (gethash key my-hash-table)
                      (sqrt val))))
     my-hash-table)

The following example illustrates a maphash call that removes all entries whose keys equal their corresponding values.

(maphash #'(lambda (key val)
              (if (eq key val) (remhash key 'my-hash-table)))
     'my-hash-table)

For a table of related items: See the section "Table Functions".

zl:maphash-equal function hash-table &rest args

For each entry in hash-table, calls function on the key of the entry and the value of the entry. If args are supplied, they are passed along to function following the value of the entry. This message is obsolete; use maphash instead.

mapl fcn list &rest more-lists

Applies fcn to list and to successive sublists of that list. If all the lists are not of the same length the iteration terminates when the shortest list runs out and excess sublists of it are ignored.

mapl works like maplist, except that it does not accumulate the results of calling fcn. Use mapl when fcn is being called merely for its side effects, rather than its returned value.

(mapl #'print '(a b c))
=>
(A B C)
(B C)
(C)
(A B C)

For a table of related items: See the section "Mapping Functions".

maplist fcn list &rest more-lists

Applies fcn to list and to successive sublists of that list rather than to successive elements as does mapcar. It returns a list that accumulates the results of the successive calls to fcn.

fcn must take as many arguments as there are lists.
Examples:

(maplist #'append '(a b c d) '(1 2 3 4))
=> ((A B C D 1 2 3 4) (B C D 2 3 4) (C D 3 4) (D 4))

(maplist #'(lambda (a-list) (cons 'twiddle a-list))
  '(blank dee dumb))
=> ((TWIDDLE BLANK DEE DUMB) (TWIDDLE DEE DUMB) (TWIDDLE DUMB))

(maplist #'equal '("car" "house" "door" "barn")
  '("cat" "hat" "door" "barn"))
=> (NIL NIL T T)

(maplist #'length '(a b c d)) => (4 3 2 1)

(maplist #'identity '(a b c)) => ((a b c) (b c) (c))

For a table of related items: See the section "Mapping Functions".

**mask-field bytespec integer**

Similar to ldb ("load byte"), but the specified byte of integer is returned as a number in the position specified by bytespec in the returned word, instead of in position 0 as with ldb. integer must be an integer.

bytespec is built using function byte with bit size and position arguments. This function can be used with setf and a suitable integer to update the place. The result of a deposit-field operation on the value is stored in the updated place. Example:

(mask-field (byte 8 1) 257) => 256
(mask-field (byte 6 3) #o4567) => #o560
(setq place-numb #b100 new-byte #b100111) => 39
(setf (mask-field (byte 8 3) place-numb) new-byte) => 39
(format nil "--D #b"~B" place-numb place-numb) => 36 #b100100

For a table of related items: See the section "Summary of Byte Manipulation Functions".

**max number &rest numbers**

Returns the largest of its arguments. At least one argument is required. The arguments can be of any noncomplex numeric type. The result type is the type of the largest argument. An error is returned if any of the arguments are complex or not numbers.

Example:

(max 1 3 2) => 3
(max 5.0 42 6.7 8 3.2 12) => 42
For a table of related items, see the section "Numeric Comparison Functions".

**maximize keyword for loop**

```lisp
maximize expr {data-type} {into var}
```

Computes the maximum of `expr` over all iterations. `data-type` defaults to `number`. Note that if the loop iterates zero times, or if conditionalization prevents the code of this clause from being executed, the result is meaningless. If `loop` can determine that the arithmetic being performed is not contagious (by virtue of `data-type` being `fixnum` or `flonum`), it can choose to code this by doing an arithmetic comparison rather than calling `max`. As with the `sum` clause, specifying `data-type` implies that both the result of the `max` operation and the value being maximized is of that type. When the epilogue of the `loop` is reached, `var` has been set to the accumulated result and can be used by the epilogue code.

It is safe to reference the values in `var` during the loop, but they should not be modified until the epilogue code for the loop is reached.

Examples:

```lisp
(defun maxi (my-list)
  (loop for x from 0
        for item in my-list
        maximize item into result1
        finally (return result1))) => MAXI
(maxi '(1 2 4 5 8 7 6)) => 8
```

Not only can there be multiple accumulations in a `loop`, but a single accumulation can come from multiple places within the same `loop` form, if the types of the collections are compatible. `maximize` and `minimize` are compatible.

See the section "Accumulating Return Values for `loop`".

**zl:mem pred item list**

*Function*

Returns `nil` if `item` is not one of the elements of `list`. Otherwise, it returns the sublist of `list` beginning with the first occurrence of `item`; that is, it returns the first cons of the list whose car is `item`. The comparison is made by `pred`. Because `zl:mem` returns `nil` if it does not find anything, and something non-nil if it finds something, it is often used as a predicate.

`zl:mem` is the same as `zl:memq` except that it takes a predicate of two arguments, which is used for the comparison instead of `eq` (`zl:mem 'eq a b`) is the same as `(zl:memq a b`). (`zl:mem 'equal a b`) is the same as `(member a b)`.

`zl:mem` is usually used with equality predicates other than `eq` and `equal`, such as `=`, `char-equal` or `string-equal`. It can also be used with noncommutative predicates. The predicate is called with `item` as its first argument and the element of
list as its second argument, so:

\[(zl:mem \#'< 4 list)\]

finds the first element in list for which \((< 4 x)\) is true; that is, it finds the first element greater than 4.

For a table of related items: See the section "Functions for Searching Lists".

**zl:memass** \(\text{pred item list} \quad \text{Function}\)

Looks up item in the association list list. The value returned is the portion of the list beginning with the first pair whose car matches item, according to pred. Returns nil if none matches.

\[(\text{car (zl:memass } x y z)) = (zl:ass x y z)\].

See the function zl:mem. As with zl:mem, you can use noncommutative predicates; the first argument to the predicate is item and the second is the indicator of the element of list.

For a table of related items: See the section "Functions that Operate on Association Lists".

**member** \&rest list \quad \text{Type Specifier}

Allows the definition of a data type consisting of objects that are elements of list. An object is of this type if it is eql to one of the objects specified in list. As a type specifier, member can only be used in list form.

Examples:

\[(\text{typep 3 '(member 1 2 3)) => T}\]
\[(\text{typep 'a '(member a b c)) => T}\]
\[(\text{subtypep 'member one two three) '(member one two three four))\]
\[=> T \text{ and } T\]
\[(\text{sys:type-arglist 'member}) => (&\text{REST LIST}) \text{ and } T\]

See the section "Data Types and Type Specifiers". See the section "Lists".

**member** item list &key (test #eql) :test-not (key #identity) \quad \text{Function}

Searches list for an element that matches item according to the predicate supplied for :test. In no element matches item, nil is returned; otherwise the tail of list, beginning with the first element that satisfied the predicate, is returned. The keywords are:

: **test**

Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.
:test-not

Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key

If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

The list is searched on the top level only. For example:

(member 'item '(a b c)) => NIL

(member 'item '(a #\Space item 5/3)) => (ITEM 5/3)

member can be used as a predicate, since the value it returns is eq to the portion of the list it matches. This implies that rplaca or rplacd can be used to alter the found list element, as long as a check is made first that member did not return nil. For example:

(setq list '(loon eagle heron)) => (LOON EAGLE HERON)

(if (member 'eagle list)
  (rplaca (member 'eagle list) 'hawk)) => (HAWK HERON)

list => (LOON HAWK HERON)

In the following example, member implements the Common Lisp function union:

(defun my-union( list1 list2 &key (test #'eql) :test-not nil :key #'identity )
  (let ((result list2)
        (element nil))
    (if list2
      (dolist (element list1)
        (unless (member element list2 :test test :test-not test-not :key key)
          (setq result (cons element result))))
      (setq result list1))
    result))

For a table of related items: See the section "Functions for Searching Lists".

zl:member item in-list

Function

Returns nil if item is not one of the elements of in-list. Otherwise, it returns the sublist of in-list that begins with the first occurrence of item; that is, it returns the first cons of the list whose car is item. The comparison is made by zl:equal.

zl:member could have been defined by:

(defun zl:member (item list)
  (cond ((null list) nil)
        ((equal item (car list)) list)
        (t (zl:member item (cdr list)))) )
For a table of related items: See the section "Functions for Searching Lists".

**member-if**  
predicate list &key :key  

*Function*

Searches for an element in list that satisfies predicate. If none is found, **member-if** returns nil; otherwise it returns the tail of list beginning with the first element that satisfied the predicate. The list is searched on the top level only. **member-if** is similar to **member**. For example:

```
(member-if #'numberp '(a #\Space 5/3 item)) => (5/3 ITEM)
```

**:key**  
If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

The following example defines a retrieval function that searches an association list. This function returns the tail of the list, beginning with the pair that matches the key. Non-public data is not retrieved when stored before the pair with the appropriate key.

```
(defun secure-retrieve( alist )
  (member-if #'(lambda(x)(string= "NAME" x))
      alist :key #'car))
(setq jones
  '((SALARY . 23000)(NAME . "John Jones")
  (TITLE . "Account rep") (HIRE-DATE . 3-3-76)))
(secure-retrieve jones) =>
  ((NAME . "John Jones") (TITLE . "Account rep")
  (HIRE-DATE . 3-3-76))
```

Note that the :key argument of #car extracts the matching field designator from the a-list pair.

For a table of related items: See the section "Functions for Searching Lists".

**member-if-not**  
predicate list &key key  

*Function*

Searches for the first element in list that does not satisfy predicate. If every element satisfies the predicate, **member-if-not** returns nil; otherwise it returns the tail of list, beginning with the first element that did not satisfy the predicate. The list is searched on the top level only. **member-if-not** is similar to **member**. For example:

```
(member-if-not #'numberp '(4.0 #\Space 5/3 item)) =>
  (4.0 #\Space 5/3 ITEM)
```

```
(member-if-not #'numberp '(5/3 4.0)) => NIL
```

**:key**  
If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

The following example defines a retrieval function that searches an association list.
This function returns the tail of the list, beginning with the first pair that does not match a particular key. Non-public data is not retrieved when stored before the pair with the appropriate key.

```lisp
(defun secure-retrieve (alist)
  (member-if-not #'(lambda(x)(or (string= "SALARY" x)
                                  (string= "RELIGION" x)))
                 alist :key #'car))
(setq jones
  '((SALARY . 23000)(NAME . "John Jones")
   (TITLE . "Account rep") (HIRE-DATE . 3-3-76)))
(secure-retrieve jones) =>
((NAME . "John Jones") (TITLE . "Account rep")
 (HIRE-DATE . 3-3-76))
```

For a table of related items: See the section "Functions for Searching Lists".

---

**zl:memq item in-list**

Returns nil if *item* is not one of the elements of *in-list*. Otherwise, it returns the sublist of *in-list* that begins with the first occurrence of *item*; that is, it returns the first cons of the list whose car is *item*. The comparison is made by eq. Because zl:memq returns nil if it does not find anything, and something non-nil if it finds something, it is often used as a predicate. Examples:

- `(zl:memq 'a '(1 2 3 4)) => nil`
- `(zl:memq 'a '(g (x a y) c a d e a f)) => (a d e a f)`

Note that the value returned by zl:memq is eq to the portion of the list beginning with *a*. Thus you can use rplaca on the result of zl:memq, if you first check to make sure zl:memq did not return nil. Example:

```lisp
(let ((sublist (zl:memq x z))) ; search for x in the list z.
  (if (not (null sublist)) ; if it is found,
      (rplaca sublist y))) ; replace it with y.
```

zl:memq could have been defined by:

```lisp
(defun zl:memq (item list)
  (cond ((null list) nil)
        ((eq item (car list)) list)
        (t (zl:memq item (cdr list)))))
```

zl:memq is hand-coded in microcode and therefore especially fast.

For a table of related items: See the section "Functions for Searching Lists".

---

**merge result-type sequence1 sequence2 predicate &key key**

Destructively merges the sequences according to an order determined by *predicate*. The result is a sequence of type *result-type*, which must be a subtype of sequence, as for the function coerce.
Sequence1 and sequence2 can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

predicate should take two arguments and return a non-nil value if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (the the appropriate sense), then predicate should return nil.

The merge function determines the relationship between two elements by giving keys extracted from the elements to predicate. The :key function, when applied to an element, should return the key for that element. The :key function defaults to the identity function, thereby making the element itself be the key.

The :key function should not have any side effects. A useful example of a :key function would be a component selector function for a defstruct structure, used to merge a sequence of structures.

If the :key and predicate functions always return, the merging function will always terminate. The result of merging two sequences x and y is a new sequence z, such that the length of z is the sum of the lengths of x and y, and z contains all of the elements of x and y. If x1 and x2 are two elements of x, and x1 precedes x2 in x, then x1 precedes x2 in z, and similarly for the elements of y. In short, z is an interleaving of x and y.

Moreover, if x and y were correctly sorted according to predicate, then z will also be correctly sorted. For example:

```
(merge 'list '(1 3 4 6 7) '(2 5 8) #'<) => (1 2 3 4 5 6 7 8)
```

If x or y is not so sorted, then z will not be sorted, but will nevertheless be an interleaving of x and y. For example:

```
(merge 'list '(3 6 4 1 7) '(2 5 8) #'<) => (2 3 5 6 4 1 7 8)
```

```
(setq a (vector 1 2 5) b (vector 2 3 4))
```

```
(merge 'list a b #'<) => (1 2 2 3 4 5)
```

Note in the previous example that the input sequences are vectors, but merge produces the requested list. In the following example, input sequences are of different types. This generally results in reduced efficiency. Also, the result is not completely in order because the sequence c is not sorted according to #'<.

```
(setq c (3 2 1) d #(1 2 4))
```

```
(merge c d #'<) =>'(1 2 3 2 1 4)
```

In the previous example, the elements from c are the elements in positions 2 through 4 in the merged list.

The merging operation is guaranteed to be stable, that is, if two or more elements are considered equal by predicate, then the elements from sequence1 will precede
those from sequence2 in the result. The predicate is assumed to consider two elements from x and y to be equal if \((\text{funcall } \text{predicate } x \ y)\) and \((\text{funcall } \text{predicate } y \ x)\) are both false. For example:

\[
(\text{merge } \text{'string } "BOY" \ "nosy" \ #\text{char-lessp}) \Rightarrow \text{"BnOsYy"}
\]

The result can not be "BnoOsYy", "BnOosyY", or "BnoOsyY", because the function \text{char-lessp} ignores case, and so considers the characters Y and y to be equal. Since Y and y are equal, the stability property then guarantees that the character from the first argument (Y) must precede the one from the second argument (y).

For a table of related items: See the section "Sorting and Merging Sequences".

clos:method-combination-error \text{format-string} \ &\text{rest} \ \text{args}

\text{Function}

Signals an error within method combination; it should be called only within the dynamic extent of a method-combination function.

\text{format-string} \quad \text{A control string that can be given to } \text{format.}

\text{args} \quad \text{Arguments required by the } \text{format-string.}

\text{flavor:method-options} \ \text{function-spec}

\text{Function}

Returns the \((\text{options}...)\) portion of the \text{function-spec}. \text{options} is the \text{options} argument that was given in the \text{defmethod} form for this method, such as \text{:before} or \text{:progn}. See the section "Function Specs for Flavor Functions".

The \((\text{options}...)\) portion is the \text{cdddr} of the \text{function-spec}. Functions specs for methods are in the form:

\[
(\text{type generic flavor options}...)
\]

type is typically \text{flavor:method}.

This is useful in the bodies of \text{define-method-combination} forms. The definition of the \text{:case} method combination type provides a good example of the use of \text{flavor:method-options}. See the section "Examples of \text{define-method-combination}".

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

clos:method-qualifiers \text{method}

\text{Generic Function}

Returns a list of the qualifiers of the \text{method}.

\text{method} \quad \text{A method object.}

This special form goes into a loop in which it reads forms and sequentially expands them, printing out the result of each expansion (using the grinder to improve readability). See the section "Functions for Formatting Lisp Code". It terminates when you press the END key. If you type in a form that is not a macro form, there are no expansions and so it does not type anything out, but just prompts you for another form. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

For example:

```
(mexp)
Type End to stop expanding forms
```

Macro form → (loop named t until nil return 5)
(2L:LOOP NAMED T UNTIL NI RETURN 5) →
(PROG T NIL
  SI:NEXT-LOOP AND NIL
    (GO SI:END-LOOP))
  (RETURN 5)
  (GO SI:NEXT-LOOP)
SI:END-LOOP)

Macro form → (defparameter foo bar) →
(PROGN (EVAL-WHEN (COMPILE)
    (COMPILER:SPECIAL-2 'FOO))
  (EVAL-WHEN (LOAD EVAL)
    (SI:DEFCONST-1 FOO BAR NIL)))

See the section "Expanding Lisp Expressions in Zmacs". That section describes two editor commands that allow you to expand macros — c-sh-M and m-sh-M. There is also the Command Processor command, Show Expanded Lisp Code. See the document *Genera User's Guide*.

**min number &rest numbers**

Returns the smallest of its arguments. At least one argument is required. The arguments can be of any noncomplex numeric type. The result type is the type of the smallest argument. An error is returned if any of the arguments are complex or not numbers.

Example:

```
(min 1 3 2) => 1
(min 5.0 42 6.7 8 3.2 12) => 3.2
```

For a table of related items, see the section "Numeric Comparison Functions".
minimize keyword for loop

\texttt{minimize expr \{data-type\} \{into var\}}

Computes the minimum of \textit{expr} over all iterations. \textit{data-type} defaults to \texttt{number}. Note that if the loop iterates zero times, or if conditionalization prevents the code of this clause from being executed, the result is meaningless. If \texttt{loop} can determine that the arithmetic being performed is not contagious (by virtue of \textit{data-type} being \texttt{fixnum} or \texttt{flonum}), it can choose to code this by doing an arithmetic comparison rather than calling \texttt{min}. As with the \texttt{sum} clause, specifying \textit{data-type} implies that both the result of the \texttt{min} operation and the value being minimized is of that type. When the epilogue of the \texttt{loop} is reached, \textit{var} has been set to the accumulated result and can be used by the epilogue code.

It is safe to reference the values in \textit{var} during the loop, they should not be modified until the epilogue code for the loop is reached.

Examples:

\begin{verbatim}
(defun mini (my-list)
  (loop for x from 0
        for item in my-list
        minimize item into result1
        finally (return result1)))  => MINI
(mini '(3 4 5 6 0 8 7)) => 0
\end{verbatim}

Not only can there be multiple accumulations in a \texttt{loop}, but a single accumulation can come from multiple places \textit{within the same} \texttt{loop} form, if the types of the collections are compatible. \texttt{minimize} and \texttt{maximize} are compatible.

See the section "Accumulating Return Values for \texttt{loop}"

\texttt{zl:minus x} \hspace{1cm} \textbf{Function}

Returns the negative of \textit{x}. \texttt{zl:minus} is similar to \texttt{-} used with one argument.

Examples:

\begin{verbatim}
(zl:minus 1) => -1
(zl:minus -3.0) => 3.0
\end{verbatim}

For a table of related items, see the section "Arithmetic Functions".

\texttt{minusp number} \hspace{1cm} \textbf{Function}

Returns \texttt{t} if its argument is a negative number, strictly less than zero. Otherwise it returns \texttt{nil}. If \textit{number} is not a noncomplex number, \texttt{minusp} signals an error.

Examples:
(minusp -5) => T
(minusp 0) => NIL
(minusp 0.0) => NIL
(minusp -0.0) => NIL
(minusp least-negative-single-float) => t
(minusp least-positive-single-float) => nil

For a table of related items, see the section "Numeric Property-checking Predicates".

mismatch sequence1 sequence2 &key :from-end (test #'eql) :test-not :key (:start1 0) (:start2 0) :end1 :end2 Function

Compares the specified subsequences of sequence1 and sequence2 element-wise. If they are of equal length and match in every element, the result is nil. Otherwise, the result is a non-negative integer representing the index within sequence1 of the leftmost position at which the two subsequences fail to match, or, if one subsequence is shorter than and a matching prefix of the other, the result is the index relative to sequence1 beyond the last position tested.

For example:

(mismatch '(loon heron stork) '(loon heron stork)) => NIL
(mismatch '(hawk loon owl pelican) '(hawk loon eagle pelican)) => 2
(mismatch '(1 2 3) '(1 2 3 4 5)) => 3

If the value of the :from-end keyword is non-nil, one plus the index of the rightmost position in which the sequences differ is returned. In effect, the (sub)sequences are aligned at their right-hand ends and the last elements are compared, then the ones before, and so on. The index returned is again an index relative to sequence1. For example:

(mismatch '(hawk loon owl pelican) '(hawk loon eagle pelican) :from-end t) => 3

:test specifies the test to be performed. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is true. Where testfun is the test function specified by :test, keyfn is the function specified by :key and x is an element of the sequence. The default test is eql.

For example:

(mismatch '(2 3 4) '(1 2 3) :test #'>) => NIL

:test-not is similar to :test, except that the sense of the test is inverted. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is false.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.
For example:

```
(mismatch '((north 1)(south 2)) '((right 1)(left 2)) :key #'second)
=> NIL
```

For a table of related items: See the section "Searching for Sequence Items".

**mod number divisor**

*Function*

Divides `number` by `divisor`, converting the quotient into an integer and truncating the result toward negative infinity. Returns the remainder. This is the same as the second value of `floor number divisor`.

When there is no remainder, the returned value is 0.

The arguments can be integers or floating-point numbers.

Examples:

```
(mod 3 2) => 1
(mod -3 2) => 1
(mod 3 -2) => -1
(mod -3 -2) => -1
(mod 4 -2) => 0
(mod 3.8 2) => 1.8
(mod -3.8 2) => 0.20000005
```

Related Functions:

- `floor`
- `rem`

For a table of related items, see the section "Arithmetic Functions".

**mod n**

*TypeSpecifier*

Defines the set of non-negative integers less than `n`. This is equivalent to `(integer 0 n-1)`, or to `(integer 0 (n))`.

As a type specifier, `mod` can only be used in list form.

Examples:

```
(typep 3 '(mod 4)) => T
(typep 5 '(mod 4)) => NIL
(typep 4 '(mod 4)) => NIL
(subtypep 'bit '(mod 2)) => T and T
(sys:type-arglist 'mod) => (N) and T
```

See the section "Data Types and Type Specifiers". For a discussion of the function `mod`: See the section "Numbers".

**:modify-hash key function &rest args**

*Message*
Combines the actions of `:get-hash` and `:put-hash`. It lets you both examine the value for a particular key and change it. It is more efficient because it does the hash lookup once instead of twice.

It finds `value`, the value associated with `key`, and `key-exists-p`, which indicates whether the key was in the table. It then calls `function` with `key`, `value`, `key-exists-p`, and `other-args`. If no value was associated with the key, then `value` is `nil` and `key-exists-p` is `nil`. It puts whatever value `function` returns into the hash table, associating it with `key`:

```
(send new-coms ':modify-hash k foo a b c) =>
(funcall foo k val key-exists-p a b c)
```

This function is obsolete; use `modify-hash` instead.

```
modify-hash table key function
```

Function

Combines the action of `setf` of `gethash` into one call to `modify-hash`. It lets you both examine the value of `key` and change it. It is more efficient because it does the lookup once instead of twice.

Finds the value associated with `key` in `table`, then calls `function` with `key`, this value, a flag indicating whether or not the value was found. Puts whatever is returned by this call to `function` into `table`, associating it with `key`. Returns the new value and the key of the entry. **Note:** The actual key stored in `table` is the one that is used on `function`, not the one you supply with `key`.

For a table of related items: See the section "Table Functions".

```
*modules*
```

Variable

This special variable has as its value a list of names of the modules that have been loaded into the lisp system.

```
=> *modules*
    (TURBINE-PACKAGE GENERATOR-PACKAGE LISP)
```

```
most-negative-double-float
```

Constant

The floating-point number in double-float format closest in value (but not equal to) negative infinity.

```
most-negative-fixnum
```

Constant

The fixnum closest in value to negative infinity.

```
most-negative-long-float
```

Constant
The floating-point number in long-float format closest in value (but not equal to) negative infinity. In Symbolics Common Lisp this constant has the same value as **most-negative-double-float**.

**most-negative-short-float**

The floating-point number in short-float format closest in value (but not equal to) negative infinity. In Symbolics Common Lisp this constant has the same value as **most-negative-single-float**.

**most-negative-single-float**

The floating-point number in single-float format closest in value (but not equal to) negative infinity.

**most-positive-double-float**

The floating-point number in double-float format which is closest in value (but not equal to) positive infinity.

**most-positive-fixnum**

The value of **most-positive-fixnum** is that fixnum closest in value to positive infinity.

**most-positive-long-float**

The value of **most-positive-long-float** is that floating-point number in long-float format which is closest in value (but not equal to) positive infinity. In Symbolics Common Lisp this constant has the same value as **most-positive-double-float**.

**most-positive-short-float**

The value of **most-positive-short-float** is that floating-point number in short-float format which is closest in value (but not equal to) positive infinity. In Symbolics Common Lisp this constant has the same value as **most-positive-single-float**.

**most-positive-single-float**

The value of **most-positive-single-float** is that floating-point number in single-float format which is closest in value (but not equal to) positive infinity.

**mouse-char-p** char

Returns **t** if **char** is a mouse character, **nil** otherwise.
**zl:multiple-value**  
*vars value*  
*Special Form*

Used for calling a function that is expected to return more than one value. This is the Zetalisp name for **multiple-value-setq**. See the section "Special Forms for Receiving Multiple Values".

**multiple-value-bind**  
*vars value &body body &whole form &environment env*  
*Special Form*

Similar to **multiple-value-setq**, but locally binds the variables that receive the values, rather than setting them, and has a body — a set of forms that are evaluated with these local bindings in effect. First form is evaluated. Then the variables are bound to the values returned by form. Then the body forms are evaluated sequentially, the bindings are undone, and the result of the last body form is returned.

```lisp
(let ((ret1 '())
  (ret2 nil))
  (multiple-value-setq (ret1 ret2) (subtypep type-1 type-2))
  (if ret2
    (values ret1 ret2)
    (and (multiple-value-setq (ret1 ret2)
      (my-even-more-expensive-subtype type-1 type-2))
      (if ret2
        (values ret1 ret2)
        (error "Could not determine if ~A is a subtype of ~A." type-1 type-2)))))
```

See the section “Special Forms for Receiving Multiple Values”.

CLOE Note: This is a macro in CLOE.

**multiple-value-call**  
*function &rest args*  
*Special Form*

First evaluates function to obtain a function. It then evaluates all the forms in args, gathering together all the values of the forms (not just one value from each). It gives these values as arguments to the function and returns whatever the function returns.

For example, suppose the function **frob** returns the first two elements of a list of numbers:

```
(multiple-value-call #'+ (frob '(1 2 3)) (frob '(4 5 6)))  
<=>(+ 1 2 4 5) => 12.  
```

```lisp
(defmacro get-values (form)
  `(multiple-value-call #'(lambda (&rest args) (format nil "("-"a"-, '-') args))
    ,form))

(get-values (get-decoded-time)) => "40, 58, 8, 25, 8, 1984, 1, T, 5"
```
(get-values (floor 9 2)) => "4, 1"

(get-values (+ 9 2)) => "11"

See the section “Special Forms for Receiving Multiple Values”.

**multiple-value-list form**

Evaluates form and returns a list of the values it returned. This is useful for when you do not know how many values to expect.

Examples:

```lisp
(setq a (multiple-value-list (intern "goo")))
a => (goo nil)
```

This is similar to the example of **multiple-value-setq**; a is set to a list of two elements, the two values returned by **intern**.

In this example, **multiple-value-list** implements a very simplistic trace function (traces functions that return multiple values).

```lisp
(defun trace-function (function-name &rest args)
  (let ((fundef (symbol-function function-name))
         (result '()))
    (format *trace-output*
            "~&Entering ~a with arguments ~{ ~a~}" function-name args)
    (setq result (multiple-value-list (apply fundef args)))
    (format *trace-output*
            "~&Exiting ~a with values ~{ ~a~}" function-name result)
    (values-list result)))
```

CLOE Note: This is a macro in CLOE.

**multiple-value-prog1** value &body body

Evaluates its first form argument and saves the values produced. Then evaluates the remaining forms and discards the returned values. The values saved from evaluating the first form are returned. This special form is like **prog1** except that its first form returns multiple values, **multiple-value-prog1** returns those values. In certain cases, **prog1** is more efficient than **multiple-value-prog1**, which is why both special forms exist.

See the section “Special Forms for Receiving Multiple Values”.

**flavor:multiple-value-prog2** before result &rest after

Function
Evaluates the forms and returns all the values of the second form. This is similar to multiple-value-prog1.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section “Summary of Flavor Functions and Variables”.

**multiple-value-setq vars value**

*Function*

Used for calling a function that is expected to return more than one value. value is evaluated, and the vars are set (not lambda-bound) to the values returned by value. If more values are returned than there are variables, the extra values are ignored. If there are more variables than values returned, extra values of nil are supplied. If nil appears in the var-list, then the corresponding value is ignored (you can't use nil as a variable.) Example:

(multiple-value-setq (symbol already-there-p) (intern "goo"))

In addition to its first value (the symbol), intern returns a second value, which is nil if the symbol returned as the first value was created by intern. If the symbol was already interned, the value is :internal, :external, :inherited, depending on the symbol found. (See the function intern.)

So if the symbol goo was already known and an internal symbol in the package, the variable already-there-p is set to :internal, if goo is unknown, the value of already-there-p is nil.

**multiple-value-setq** is usually used for effect rather than for value; however, its value is defined to be the first of the values returned by form.

Evaluates form and sets the variables in the list variables to those values. Excess values are discarded, and excess variables are set to nil. Returns the first value obtained from evaluating form. If no values are produced, nil is returned.

(multiple-value-setq (quotient remainder) (truncate 13 5))

The function multiple-value-setq can be used to obtain multiple values, each of which is used in further computation.

(let ((ret1 '()))
  (ret2 nil))
(multiple-value-setq (ret1 ret2) (subtypep type-1 type-2))
(if ret2
  (values ret1 ret2)
  (and (multiple-value-setq (ret1 ret2)
                           (my-even-more-expensive-subtype type-1 type-2))
       (if ret2
           (values ret1 ret2)
           (error "Could not determine if ~A is a subtype of ~A." type-1 type-2))))

See the section “Special Forms for Receiving Multiple Values”.

CLOE Note: This is a macro in CLOE.
**multiple-values-limit**  
*Constant*

A positive integer that is the upper exclusive bound on the number of values that can be returned from a function. The current value is 128 for 3600-series machines, 50 for Ivory-based machines, and 128 for CLOE.

**math:multiply-matrices**  
*Function*

**Description**

Multiplies `matrix-1` by `matrix-2`. If `matrix-3` is supplied, `math:multiply-matrices` stores the results into `matrix-3` and returns `matrix-3`; otherwise it creates an array to contain the answer and returns that. All matrices must be two-dimensional arrays, and the first dimension of `matrix-2` must equal the second dimension of `matrix-1`.

**Example**

```
(math:multiply-matrices [1 2 3] [4 5 6] [7 8 9])
```

**Arguments**

- `matrix-1` (matrix)
- `matrix-2` (matrix)
- `matrix-3` (optional, matrix)

**Call Forms**

`(math:multiply-matrices matrix-1 matrix-2 &optional matrix-3)`

---

**(flavor:method :remove si:heap)**  
*Method*

Removes the top item from the heap and returns it and its key as values. The third value is `nil` if the heap was empty; otherwise it is `t`.

For a table of related items: See the section "Heap Functions and Methods".

**(flavor:method :top si:heap)**  
*Method*

Returns the value and key of the top item on the heap. The third value is `nil` if the heap was empty; otherwise it is `t`.

For a table of related items: See the section "Heap Functions and Methods".

**name-char**  
*Function*

Accepts a string, or a string coercible object, as an argument. If `name` is the same as the name of a character object, that object is returned; otherwise `nil` is returned. `name-char` does not recognize names with modifier bit prefixes such as "hyper-space".

```
(name-char "Tab") => \Tab
(name-char "Newline") => \Newline

(char-code (name-char "Space")) => 32
```

For a table of related items, see the section "Character Names".

**sys:name-conflict**  
*Flavor*

Any sort of name conflict occurred (there are specific flavors, built on `sys:name-conflict`, for each possible type of name conflict). The following proceed types might be available, depending on the particular error:

- The :skip proceed type skips the operation that would cause a name conflict.
The :shadow proceed type prefers the symbols already present in a package to conflicting symbols that would be inherited. The preferred symbols are added to the package’s shadowing-symbols list.

The :export proceed type prefers the symbols being exported (or being inherited due to a use-package) to other symbols. The conflicting symbols are removed if they are directly present, or shadowed if they are inherited.

The :unintern proceed type removes the conflicting symbol.

The :shadowing-import proceed type imports one of the conflicting symbols and makes it shadow the others. The symbol to be imported is an optional argument.

The :share proceed type causes the conflicting symbols to share value, function, and property cells. It as if globalize were called.

The :choose proceed type pops up a window in which the user can choose between the above proceed types individually for each conflict.

**named** Keyword for loop

**named** name

Gives the prog that loop generates a name of name, so that you can use the return-from form to return explicitly out of that particular loop:

```
(loop named sue
  ...
  do (loop ... do (return-from sue value) ...) 
  ...
)
```

The return-from form shown causes value to be immediately returned as the value of the outer loop. Only one name can be given to any particular loop construct. This feature does not exist in the Maclisp version of loop, since Maclisp does not support "named progs".

See the section “loop Clauses”.

**named-structure-invoke** operation structure &rest args

Function

Calls the the handler function of the named structure symbol, found as the value of the named-structure-invoke property of the symbol, with the appropriate arguments. Operation should be a keyword symbol, and structure should be a named structure.

**named-structure-p** structure

Function

This semi-predicate returns nil if structure is not a named structure; otherwise it returns structure’s named structure symbol.

**named-structure-symbol** named-structure

Function
Returns named-structure's named structure symbol: if named-structure has an array leader, element 1 of the leader is returned, otherwise element 0 of the array is returned. Named-structure should be a named structure.

**nbutlast** list &optional (n 1)

Function

Destructive version of butlast; it changes the cdr of the second-to-last cons of the list to nil. If there is no second-to-last cons (that is, if the list has fewer than two elements) it returns nil. **nbutlast** returns all the conses in the list except for the last one. Examples:

```
(setq foo '(a b c d))
(nbutlast foo) => (a b c)
foo => (a b c)
(nbutlast '(a)) => nil
(setq a '(1 2 3 4 5 6 7))
(nbutlast a) => (1 2 3 4 5 6)
(nbutlast a 4) => (1 2)
a => (1 2)
```

For a table of related items: See the section "Functions for Modifying Lists".

**nconc** &rest arg

Function

Concatenates its arguments and returns the resulting list. The arguments are changed, rather than copied. Example:

```
(setq x '(a b c))
(setq y '(d e f))
(nconc x y) => (a b c d e f)
x => (a b c d e f)
```

Note that the value of x is now different, since its last cons has been changed (by rplacd) to the value of y. If

```
(nconc x y)
```

were evaluated again, it would yield a piece of "circular" list structure, whose printed representation would be `(a b c d e f d e f ...), repeating forever.

**nconc** could have been defined by:

```
(defun nconc (x y) ; for simplicity, this definition
  ; only works for 2 arguments.
  (cond ((null x) y) ; hook y onto x
         (t (rplacd (last x) y) ; and return the modified x.
            x)))
```

**nconc** performs destructive operations on lists except if the first argument to the function is nil. For example:
(defvar *g* nil)
(defvar *h* '(a))
(defvar *i* '(b c))

(nconc *g* *i*) => (B C)
*g* => NIL

(nconc *h* *i*) => (A B C)
*h* => (A B C)

But:

(setq *g* (nconc *g* *i*)) => (B C)
*g* => (B C)

Do not use nconc for destructive operations with t or nil. For example:

(nconc nil (ncons 'b)) => (B)

The following does not signal an error:

(nconc 'a (ncons 'b)) => (B)

In the following example, push and nreverse sort queued entries in order of priority, and nconc resets the queue.

(defun sort-queue-2 (in-queue)
  "Sorts arg first by priorities (car element), then by original order."
  (let ((for-queue1 '())
        (for-queue2 '())
        (for-queue3 '()))
    (dolist (queue-element in-queue)
      (case (car queue-element)
        (1 (push queue-element for-queue1))
        (2 (push queue-element for-queue2))
        (3 (push queue-element for-queue3))))
    ;; reverse the temporary lists
    ;; that were built by push
    (setq queue-all
          (nconc (nreverse for-queue1)
                 (nreverse for-queue2)
                 (nreverse for-queue3))))

(setq queue-all
      '((1 element-a) (2 element-b) (3 element-c) (2 element-d) (1 element-e)))
(sort-queue queue-all) =>
  ((1 ELEMENT-A) (1 ELEMENT-E) (2 ELEMENT-B) (2 ELEMENT-D) (3 ELEMENT-C))

For a table of related items: See the section "Functions for Constructing Lists and Conses".

nconc keyword for loop

nconc expr {into var}
Causes the values of \textit{expr} on each iteration to be \texttt{nconc}ed together, for example:

\begin{verbatim}
(loop for i from 1 to 3
    nconc (list i (* i i)))
=> (1 1 2 4 3 9)
\end{verbatim}

When the epilogue of the \texttt{loop} is reached, \textit{var} has been set to the accumulated result and can be used by the epilogue code. It is safe to reference the values in \textit{var} during the loop, but they should not be modified until the epilogue code for the loop is reached.

The forms \texttt{nconc} and \texttt{nconc}ing are synonymous.

Examples:

\begin{verbatim}
(defun indexing (small-list)
  (loop for x from 0
    for item in small-list
    nconc (list x item)))
(indexing '(a b c d)) => INDEXING
\end{verbatim}

is equivalent to

\begin{verbatim}
(defun indexing (small-list)
  (loop for x from 0
    for item in small-list
    nconc (list x item)))
(indexing '(a b c d)) => INDEXING
\end{verbatim}

Not only can there be multiple accumulations in a \texttt{loop}, but a single accumulation can come from multiple places \textit{within the same loop} form, if the types of the collections are compatible. \texttt{nconc}, \texttt{collect}, and \texttt{append} are compatible.

See the section "Accumulating Return Values for \texttt{loop}".

\texttt{ncons} \textit{x} \hspace{1cm} \textit{Function}

Creates a new cons, whose \textit{car} is \textit{x} (where \textit{x} can be anything) and whose \textit{cdr} is \texttt{nil}. \texttt{(ncons} \textit{x}) \texttt{is the same} \texttt{(cons} \textit{x} \texttt{nil)}. The name of the function is from "\texttt{nil-cons}".

Example:

\begin{verbatim}
(ncons '(5))
\end{verbatim}

returns a new cons whose \textit{cdr} is \texttt{nil}:

\begin{verbatim}
(('5))
\end{verbatim}

To test if a cons has been created, apply the predicate \texttt{endp} to the new cons:

\begin{verbatim}
(endp '('5))
\end{verbatim}

This returns \texttt{nil}, since \texttt{endp} returns \texttt{nil} when applied to a cons.

\texttt{ncons} is a Symbolics extension to Common Lisp.
For a table of related items: See the section "Functions for Constructing Lists and Conses".

**ncons-in-area x area**

Create a cons, whose car is x and whose cdr is nil, in the specified area. (Areas are an advanced feature of storage management. See the section "Areas").

**ncons-in-area** is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**neq x y**

(neq x y) = (not (eq x y)). This is provided simply as an abbreviation for typing convenience.

**never keyword for loop**

**never expr**

Causes the loop to return t if expr never evaluates non-null. This is equivalent to always (not expr). If the loop terminates before expr is ever evaluated, the epilogue code is run and the loop returns t.

**never expr** is like (and (not expr1) (not expr2) ...). If the loop terminates before expr is ever evaluated, never is like (and).

If you want a similar test, except that you want the epilogue code to run if expr evaluates non-null, use until.

Examples:

```lisp
(defun loop-never(my-list)
  (loop for x in my-list
        finally (print "what you going to do next ?")
        do
          (princ x) (princ " ")
          do
          and never (equal x 'a)) => LOOP-NEVER

  (loop-never '(b c a e)) => (B C A E)

  (loop-never '(a a)) => A NIL
```

See the section "Aggregated Boolean Tests for loop".
clos:next-method-p

Function

Called within the body of a method to determine whether a next method exists; returns true if a next method exists, otherwise returns false.

clos:next-method-p has lexical scope and indefinite extent.

nintersection list1 list2 &key (test #:eql) :test-not (key #:identity) Function

The destructive version of intersection. It takes list1 and list2 and returns a new list containing everything that is an element of both lists, using the cells of list1 to construct the result. The value of list2 is not altered. The keywords are:

:test Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

See the function intersection. For example:

(setq a-list '(a b c)) => (A B C)

(setq b-list '(f a d)) => (F A D)

(nintersection a-list b-list) => (A)

a-list => (A)

b-list => (F A D)

In the following example, we want the list chips-32-data, to include only chips on the approved list. We use nintersection to destructively alter chips-32-data.

(setq chips-approved
       '(68000 68010 68020 80186 80286 80386))

(setq chips-32-data '(68020 32032 80386))

(setq chips-32-data
       (nintersection chips-32-data chips-approved))

chips-32-data => (68020 80386)

chips-approved =>
       (68000 68010 68020 80186 80286 80386)
For a table of related items: See the section "Functions for Comparing Lists".

**zl:nintersection &rest lists**  
*Function*

Takes any number of `lists` that represent sets and returns a new list that represents the intersection of all the sets it is given, by destroying any of the `lists` passed as arguments and reusing the conses. `zl:nintersection` uses `eq` for its comparisons. You cannot change the function used for the comparison.  
`(zl:nintersection)` returns `nil`.  
For a table of related items: See the section "Functions for Comparing Lists".

**ninth list**  
*Function*

Takes a list as an argument, and returns the ninth element of `list`. `ninth` is identical to  
`(nth 8 list)`

For example:

```lisp
(setq letters '(a b c d e f g h i j k l)) => (A B C D E F G H I J K L)

(ninth letters) => I
```

This function is provided because it makes more sense than using `nth` when you are thinking of the argument as a list rather than just as a cons.

For a table of related items: See the section "Functions for Extracting from Lists".

**nleft n l &optional tail**  
*Function*

Returns a "tail" of `l` consisting of the last `n` elements of `l`, that is, one of the conses that makes up `l`, or `nil`. If `n` is too large, `nleft` returns `l`. Example:

```lisp
(nleft 2 '(bass bluefish tuna))
```

returns the last 2 conses:

```
(bluefish tuna)
```

`(nleft n l tail)` takes the cdr of the original `l` and returns a list such that taking `n` more cdrs of it would yield `tail`. You can see that when `tail` is `nil`, this is the same as the two-argument case. If `tail` is not `eq` to any tail of `l`, `nleft` returns `nil`. Example:

```lisp
(setq z '(a b c d e)) => (A B C D E)
(setq y (cdddr z)) => (D E)
(nleft 2 z y) => (B C D E)
```

`nleft` is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Extracting from Lists".
\textbf{nlistp} \(x\)

\textit{Function}

Returns \(t\) if its argument \(x\) is not a list, otherwise \textit{nil}. This means \((\text{nlistp}\ \text{nil})\) is \textit{nil}. \text{nlistp} can be thought of as \((\text{not-listp})\). Note this distinction between \text{nlistp} and \text{zl:nlistp}. \((\text{zl:nlistp}\ \text{nil})\) is \(t\), since \text{zl:nlistp} returns \text{nil} if its argument is a cons.

Example:

\[(\text{nlistp}\ '(\text{heron} \text{sandpiper} \text{bluejay}))\]

returns \textit{nil}, since this argument is a list.

But:

\[(\text{nlistp}\ '"\text{sss}"")\]

returns \(t\) since its argument is not a list.

\text{nlistp} is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Predicates that Operate on Lists".

\textbf{zl:nlistp} \(x\)

\textit{Function}

Equivalent to \textit{atom}, so it returns \(t\).

\textbf{nodeclare} \textit{keyword for loop}

\textbf{nodeclare} \textit{variable-list}

The variables in \textit{variable-list} are noted by \textit{loop} as not requiring local type declarations. Consider the following:

\begin{verbatim}
(declare (special k) (fixnum k))
(defun foo (l)
  (loop for x in l as k fixnum = (f x) ...))
\end{verbatim}

If \(k\) did not have the \textit{fixnum} data-type keyword given for it, then \textit{loop} would bind it to \textit{nil}, and some compilers would complain. On the other hand, the \textit{fixnum} keyword also produces a local \textit{fixnum} declaration for \(k\); since \(k\) is special, some compilers complain (or error out). The solution is to do:

\begin{verbatim}
(defun foo (l)
  (loop nodeclare (k)
    for x in l as k fixnum = (f x) ...))
\end{verbatim}

which tells \textit{loop} not to make that local declaration. The \textbf{nodeclare} clause must come \textit{before} any reference to the variables so noted. Positioning it incorrectly causes this clause to not take effect, and cannot be diagnosed. See the macro \textit{loop}.

This exists for compatibility with other implementations of \textit{loop}. 
not $x$  

*Function*

Returns $t$ if $x$ is nil, otherwise returns nil. **null** is the same as not; both functions are included for the sake of clarity. Use **null** to check whether something is nil; use **not** to invert the sense of a logical value. Even though Lisp uses the symbol nil to represent falseness, you should not make understanding of your program depend on this. For example, one often writes:

```lisp
(cond ((not (null lst)) ...) 
      ( ... ))
```

rather than

```lisp
(cond (lst ...) 
      ( ... ))
```

There is no loss of efficiency, since these compile into exactly the same instructions.

The following example searches a list:

```lisp
(defun my-search(l key)
  (if (null l) nil
      (or (equal (car l) key)
          (search (cdr l) key))))
```

See the function null.

not $type$  

*TypeSpecifier*

Defines the set of objects that are not of the specified type. As a type specifier, **not** can only be used in list form.

Examples:

```lisp
(typep "music" '(not integer)) => T
(subtypep 'nil '(not t)) => T and T
(subtypep 'nil '(not integer)) => T and T
(subtypep 'bit (not nil)) => T and T
(equal-typep t (not nil)) => T
(sys:type-arglist 'not) => (TYPE) and T
```

See the section "Data Types and Type Specifiers". See the section "Predicates".

notany predicate sequence &rest more-sequences  

*Function*

Returns nil as soon as any invocation of predicate returns a non-nil value. predicate must take as many arguments as there are sequences provided. predicate is first applied to the elements of the sequences with an index of 0, then with an index of 1, and so on, until a termination criterion is reached or the end of the shortest of the sequences is reached. If the end of a sequence is reached, notany returns a non-nil value. Thus considered as a predicate, it is true if no invocation of predicate is true.
sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:

```lisp
(notany #'oddp '(1 2 5)) => NIL

(notany #'equal '(0 1 2 3) '(3 2 1 0)) => T
```

If predicate has side effects, it can count on being called first on all those elements with an index of 0, then all those with an index of 1, and so on.

The following example demonstrates how notany implements a test to determine if an element of a sequence exceeds a critical value.

```lisp
(setq limit-value 1024 sequence (vector 16 64 512 128 32))

(notany #'(lambda(x) (> x limit-value)) sequence) => t
```

For a table of related items: See the section "Predicates that Operate on Sequences".

noteworthy predicate sequence &rest more-sequences Function

Returns a non-nil value as soon as any invocation of predicate returns nil. predicate must take as many arguments as there are sequences provided. predicate is first applied to the elements of the sequences with an index of 0, then with an index of 1, and so on, until a termination criterion is reached or the end of the shortest of the sequences is reached. If the end of a sequence is reached, noteworthy returns nil. Thus considered as a predicate, it is true if not every invocation of predicate is true.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:

```lisp
(notevery #'oddp '(1 2 5)) => T

(notevery #'equal '(1 2 3) '(1 2 3)) => NIL

(setq limit-value 212 sequence (vector 16 64 512 128 32))

(notevery #'(lambda(x) (<= x limit-value)) sequence) => t
```

If predicate has side effects, it can count on being called first on all those elements with an index of 0, then all those with an index of 1, and so on.

For a table of related items: See the section "Predicates that Operate on Sequences".

notinline Declaration
(notinline function1 function2 ... ) specifies that it is undesirable to compile the specified functions in-line. This declaration is pervasive, that is, it affects all code in the body of the form.

Note that rules of lexical scoping are observed; if one of the functions mentioned has a lexically apparent local definition (as made by flet or labels), then the declaration applies to that local definition and not to the global function definition.

See the section "Declaration Specifiers".

clos:no-next-method generic-function calling-method &rest args   Generic Function

Provides a mechanism for users to control what happens when clos:call-next-method is called, and no next method exists. The default method for clos:call-next-method signals an error.

The typical way to specialize clos:call-next-method is to define a primary method, which would override the default primary method.

This generic function is called automatically, and is not intended to be called by users.

generic-function The generic function of method.
calling-method The method whose call to clos:call-next-method resulted in this call to clos:no-next-method.
args A list of arguments to clos:call-next-method.

nreconc l tail   Function

Reverses the elements of l, concatenates them with the elements of tail, and returns the resulting list. Modifies both arguments. (nreconc l tail) is exactly the same as (nconc (zl:nreverse l) tail) except that it is more efficient. Both l and tail should be lists. Example:

```
(setq x '(a b c))
(setq y '(d e f))
(nreconc x y) => (c b a d e f)
x => undefined
```

nreconc could have been defined by:

```
(defun nreconc (l tail)
  (cond ((null l) tail)
        ((nreverse1 l tail)))
)

(defun nreverse1 (l tail) ; auxiliary function
  (cond ((null (cdr l)) (rplacd l tail))
        ((nreverse1 (cdr l) (rplacd l tail)))))
;; this last call depends on order of argument evaluation.
```
Note: `nreconc` actually works differently, and uses both `rplacd` and element shuffling. It therefore rarely causes `rplacd-forwarding`.

In the following example, `nreconc` sorts queued entries in order of priority.

```lisp
(defun sort-queue (in-queue)
  "Sorts arg first by priorities (car element), then by original order."
  (let ((for-queue1 '())
        (for-queue2 '())
        (for-queue3 '()))
    (dolist (queue-element in-queue)
      (case (car queue-element)
        (1 (push queue-element for-queue1))
        (2 (push queue-element for-queue2))
        (3 (push queue-element for-queue3))))
    ;; reverse the temporary lists
    ;; that were built by push
    (nreconc for-q1
             (nreconc for-q2 (nreverse for-q3))))
    (setq queue-all
      '(((1 element-a) (2 element-b) (3 element-c) (2 element-d) (1 element-e)))
      (sort-queue queue-all) =>
      ((1 ELEMENT-A) (1 ELEMENT-E) (2 ELEMENT-B) (2 ELEMENT-D) (3 ELEMENT-C)))
```

See the section "Cdr-Coding".

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**nreverse** sequence

*Function*

Returns a sequence containing the same elements as `sequence`, but in reverse order. The result may or may not be `eq` to the argument, so it is usually wise to say something like `(setq x (nreverse x))`, because `(nreverse x)` is not guaranteed to leave the reversed value in `x`.

`sequence` can be either a list or a vector (one-dimensional array). Note that `nil` is considered to be a sequence, of length zero.

For example:

```lisp
(setq item-list '(heron stork loon owl)) => (HERON STORK LOON OWL)

(nreverse item-list) => (OWL LOON STORK HERON)

item-list => (HERON)
```

When used on a list, `nreverse` reverses the list by shuffling list elements or by calling `rplacd` on conses making up the list, or both. `nreverse` rarely causes `rplacd-forwarding`. For example, under Genera, this usually returns a cdr-coded list:

```lisp
(nreverse (list 'a 'b 'c))
```
**Note:** The exact list destruction which occurs when using `nreverse` is undefined. It depends on the the cdr-coding of the list and the machine type. See the section "Cdr-Coding".

`nreverse` is the destructive version of `reverse`.

The following example creates a list of primes from 2 to 100, and demonstrates how `nreverse` restores a list of elements, built by `push`, to source order:

```lisp
(do ((i 2 (+ i 1))
     (return-list '()))
    ((= i 100)(nreverse return-list))
  (if (primep i)
      (push i return-list)))
```

Generally, use `nreverse` only with recently consed lists, or lists that are known to be dispensable. In other cases, `reverse` might be more appropriate.

For a table of related items: See the section "Functions for Modifying Lists".

For a table of related items: See the section "Sequence Modification".

---

**Function**

`zl:nreverse l`

Reverses its argument, which should be a list, by shuffling list elements or by calling `rplacd` on conses making up the list, or both. `zl:nreverse` rarely causes `rplacd`-forwarding. The following usually returns a cdr-coded list:

**(nreverse (list 1 2 3))**

Here is an example of `zl:nreverse`:

`(zl:nreverse '(a b c)) => (c b a)`

**Note:** The exact list destruction which occurs when using `zl:nreverse` is undefined. It depends on the the cdr-coding of the list and the machine type. See the section "Cdr-Coding".

For a table of related items: See the section "Functions for Modifying Lists".

---

**Function**

`nset-difference list1 list2 &key (test #'eql) test-not (key #'identity)`

Returns a new list of elements of `list1` that do not appear in `list2`, using the cells of `list1` to construct the result. The value of `list2` is not altered. Destructive version of `set-difference`. The keywords are:

:`test` Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The `item` matches the specification only if the predicate returns `t`. If :test is not supplied, the default operation is `eql`.

:`test-not` Similar to :test, except that `item` matches the specification only if there is an element of the list for which the predicate returns `nil`. 
If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

See the function `set-difference`. For example:

```
(setq a-list '(eagle hawk loon pelican)) => (EAGLE HAWK LOON PELICAN)
(setq b-list '(owl hawk stork)) => (OWL HAWK STORK)
(nset-difference a-list b-list) => (EAGLE LOON PELICAN)
```

In the following example, we no longer want the list of approved chips `chips-approved` to include any chips with a 32 bit data path. We use `nset-difference` to destructively alter `chips-approved`:

```
(setq chips-approved '(68000 68010 68020 80186 80286 80386))
(setq chips-32-data '(68020 32032 80386))
(setq chips-approved (nset-difference chips-approved chips-32-data))
```

For a table of related items: See the section "Functions for Comparing Lists".

**nset-exclusive-or** `list1 list2` &key `(test #eql) :test-not (#identity) :key nil`  

Destructive version of `set-exclusive-or`. It returns a list of elements that appear in exactly one of `list1` and `list2`, and alters values of the list arguments during the operation. The keywords are:

- **:test** Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The `item` matches the specification only if the predicate returns `t`. If :test is not supplied, the default operation is `eql`.

- **:test-not** Similar to :test, except that `item` matches the specification only if there is an element of the list for which the predicate returns `nil`.

- **:key** If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

See the function `set-exclusive-or`. For example:
(setq a-list '(eagle hawk loon pelican)) => (EAGLE HAWK LOON PELICAN)

(setq b-list '(owl hawk stork)) => (OWL HAWK STORK)

(nset-exclusive-or a-list b-list) => (EAGLE LOON PELICAN OWL STORK)

a-list => (EAGLE HAWK LOON PELICAN)
b-list => (OWL STORK)

For a table of related items: See the section "Functions for Comparing Lists".

**nstring-capitalize** string &key (:start 0) :end

Function

Returns string modified such that for every word in string, the initial character, if case-modifiable, is uppercased. All other case-modifiable characters in the word are lowercased. This function is the destructive version of string-capitalize.

For the purposes of string-capitalize, a word is defined as a consecutive subsequence of alphanumeric characters or digits, delimited at each end either by a non-alphanumeric character, or by an end of string.

The keywords let you select portions of the string argument for uppercasing. These keyword arguments must be non-negative integer indices into the string array. The entire argument, string, is returned, however.

If string is not a string, an error is signalled.

: start  Specifies the position within string from which to begin uppercasing (counting from 0). Default is 0, the first character in the string. : start must be ≤ : end.

: end  Specifies the position within string of the first character beyond the end of the operation. Default is nil, that is, the operation continues to the end of the string.

Examples:

(nstring-capitalize " a bUNch of WOrDs" :start 0 :end 3) => " A bUNch of WOrDs"

(nstring-capitalize " a bUNch of WOrDs" :start 8) => " a bUNch Of Words"

(nstring-capitalize " 1234567 a bunch of numbers" :start 1 :end 5) => " 1234567 a bunch of numbers"
(setq a-string "poppy SEED")

(nstring-capitalize a-string)
=> "Poppy Seed"

a-string => "Poppy Seed"

For a table of related items: See the section "String Conversion".

**nstring-capitalize-words** string &key (start 0) (end nil)  

*Function*

The destructive version of **string-capitalize-words**.

**nstring-capitalize-words** returns string, modified such that hyphens are changed to spaces and initial characters of each word are capitalized if they are case-modifiable.

If string is not a string, an error is signalled. See the function **string**.

The keywords let you select portions of the string argument for uppercasing. These keyword arguments must be non-negative integer indices into the string array. The entire argument, string, is returned, however.

:start  Specifies the position within string from which to begin uppercasing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end  Specifies the position within string of the first character beyond the end of the uppercasing operation. Default is nil, that is, the operation continues to the end of the string.

Examples:

(nstring-capitalize-words "three-hyphenated-words")
=> "Three Hyphenated Words"

(nstring-capitalize-words  "three-hyphenated-words" :end 5)
=> "Three-hyphenated-words"

(nstring-capitalize-words  "three-hyphenated-words" :start 6)
=> "three-Hyphenated Words"

For a table of related items: See the section "String Conversion".

**nstring-downcase** string &key (start 0) (end nil)  

*Function*

Returns string, modified to replace its uppercase alphabetic characters by the corresponding lowercase characters. This function is the destructive version of the function **string-downcase**.

If string is not a string, an error is signalled.
See the function `string`.

The keywords let you select portions of the string argument for lowercasing. These keyword arguments must be non-negative integer indices into the string array. The entire argument, `string`, is returned, however.

:start Specifies the position within `string` from which to begin lowercasing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.
:end Specifies the position within `string` of the first character beyond the end of the lowercasing operation. Default is `nil`, that is, the operation continues to the end of the string.

Examples:

`(nstring-downcase "WHAT TIME IS IT !!!!") => "what time is it !!!!"
(nstring-downcase "A BUNCH OF WORDS" :start 2 :end 7) => "A bunch OF WORDS"
(nstring-downcase "A BUNCH OF WORDS" :start 11) => "A BUNCH OF words"
(setq string "THREE UPPERCASE WORDS") => "THREE UPPERCASE WORDS"
(nstring-downcase string :start 0 :end 5 ) => "three UPPERCASE words"
(nstring-downcase string :start 16 :end nil) => "three UPPERCASE words"
string => "three UPPERCASE words"

For a table of related items: See the section "String Conversion".

**nstring-upcase string &key (start 0) (end nil)**

Function

Returns `string`, modified by replacing its lowercase alphabetic characters by the corresponding uppercase characters. This function is the destructive version of the function `string-upcase`.

If `string` is not a string, an error is signalled. See the function `string`.

The keywords let you select portions of the string argument for uppercasing. These keyword arguments must be non-negative integer indices into the string array. The entire string argument is returned, however.

:start Specifies the position within `string` from which to begin uppercasing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.
:end Specifies the position within `string` of the first character beyond the end of the uppercasing operation. Default is `nil`, that is, the operation continues to the end of the string.

Characters not in the standard character set are unchanged.

Examples:
(nstring-upcase "a four word string" :start 2 :end 6)
  => "a FOUR word string"
(nstring-upcase "a four word string" :start 12)
  => "a four word STRING"
(setq a-string "poppy SEED")

(nstring-upcase a-string)
  => "POPPY SEED"

a-string => "POPPY SEED"

For a table of related items: See the section "String Conversion".

\[\text{nsublis} \ \text{alist} \ \text{tree} \ \&\text{rest} \ \text{args} \ \&\text{key (:test #'eql) :test-not (:key #'identity)}\]

Function

Destructive version of \text{sublis}. It makes substitutions for objects in a tree, altering the relevant parts of \text{tree}. See the function \text{sublis}.

The keywords are:

\textbf{:test} \quad \text{Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.}

\textbf{:test-not} \quad \text{Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.}

\textbf{:key} \quad \text{If not nil, should be a function of one argument that will extract the part to be tested from the whole element.}

Example:

\[
\text{(setq exp '(*(x y) (+ x y))) => (* X Y) (+ X Y)}\]
\[
\text{(nsublis '(* . 100)) exp) => (* 100 Y) (+ 100 Y)}\]
\[
\text{exp => (* 100 Y) (+ 100 Y)}\]

Thus, \text{nsublis} is comparable to several \text{nsubst} operations in parallel. The following example shows that sequential calls to \text{nsubst} can not replace every \text{nsublis}.

\[
\text{(setq alist (pairlis '(monkey zebra) '(zebra monkey)))}
\text{(setq newthing '(is-taller monkey zebra))}
\text{(nsublis alist newthing) => (IS-TALLER ZEBRA MONKEY)}\]

For a table of related items: See the section "Functions for Modifying Lists".
**zl:nsublis** **alist form**

Destructive version of **sublis**. Makes substitutions for symbols in a tree, but changes the original tree instead of creating a new tree.

**zl:nsublis** could have been defined by:

```lisp
(defun zl:nsublis (alist tree)
  (cond ((atom tree)
          (let ((tem (assq tree alist)))
            (if tem (cdr tem) tree)))
        (t (rplaca tree (zl:nsublis alist (car tree)))
           (rplacd tree (zl:nsublis alist (cdr tree)))
           tree)))
```

In your new programs, we recommend that you use the function **nsublis**, which is the Common Lisp equivalent of **zl:nsublis**.

For a table of related items: See the section "Functions for Modifying Lists".

**nsubst** **new old tree** **&rest args** **&key** (:test **eql**) :test-not (:key **identity**) **Function**

Destructive version of **subst**. It changes tree by substituting new for every subtree or leaf of tree that matches old according to :test. See the function **subst**. The keywords are:

: **test**
  Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The *item* matches the specification only if the predicate returns t. If :test is not supplied, the default operation is **eql**.

: **test-not**
  Similar to :test, except that *item* matches the specification only if there is an element of the list for which the predicate returns nil.

: **key**
  If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

For example:

```lisp
(setq bird-list '((waders (flamingo stork) raptors (eagle hawk)))) =>
(WADERS (FLAMINGO STORK) RAPTORS (EAGLE HAWK))

(nsubst 'heron 'stork bird-list) =>
(WADERS (FLAMINGO HERON) RAPTORS (EAGLE HAWK))

bird-list => (WADERS (FLAMINGO HERON) RAPTORS (EAGLE HAWK))

(setq sentence '(((SUB (PN . Avery)) (PRED (V . was) (ADJ . cool)))
                  (SUB (RPN . he)) (PRED (V . was) (ADJ . calm)))
              (SUB (RPN . he)) (PRED (V . was) (ADJ . suave))))
```
(nsubst '(PN . Avery) 'RPN sentence :key '#'(lambda(x)(and (consp x)(car x))))
=>
((SUB (PN . Avery)) (PRED (V . was) (ADJ . cool)))
((SUB (PN . Avery)) (PRED (V . was) (ADJ . calm)))
((SUB (PN . Avery)) (PRED (V . was) (ADJ . suave)))

For a table of related items: See the section "Functions for Modifying Lists".

zl:nsubst new old s-exp

Destructive version of subst. Changes s-exp by replacing each element occurrence of old with new. zl:nsubst could have been defined as
(defun nsubst (new old tree)
  (cond ((eq tree old) new) ; if item eq to old, replace.
        ((atom tree) tree) ; if no substructure, return arg.
        (t ; otherwise, recurse.
            (rplaca tree (nsubst new old (car tree)))
            (rplacd tree (nsubst new old (cdr tree)))
            tree)))

nsubst-if new predicate tree &rest args &key :key

Destructive version of subst-if. It change tree by substituting new for every sub-tree or leaf of tree that satisfies predicate. See the function subst-if. The keyword is:

: key If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

For example:
(setq item-list '(numbers (1.0 2 5/3) symbols (foo bar)))
=> (NUMBERS (1.0 2 5/3) SYMBOLS (FOO BAR))

(nsubst-if '3.1415 #'numberp item-list)
=> (NUMBERS (3.1415 3.1415 3.1415) SYMBOLS (FOO BAR))

item-list => (NUMBERS (3.1415 3.1415 3.1415) SYMBOLS (FOO BAR))
(setq b '(1 2 (AA BB (3 BB)) CC DD 4))

(nsubst-if 'ZZ #'numberp b)
=> (ZZ ZZ (AA BB (ZZ BB)) CC DD ZZ)

b => (ZZ ZZ (AA BB (ZZ BB)) CC DD ZZ)
The following call to `nsubst-if` uses an anonymous function. After the call, `a` is altered according to the results returned by `nsubst-if`.

```
(setq a '("In" "our" "prairie" "home" "we" "read"
   "The" "Prairie" "Home" "Companion"))
```

```
(nsubst-if "Gopher"
   #'(lambda (comparator)(string= comparator "Prairie"))
=>
("In" "our" "prairie" "home" "we" "read"
   "The" "Gopher" "Home" "Companion")
```

For a table of related items: See the section "Functions for Modifying Lists".

**nsubst-if-not**

```lisp
new predicate tree &rest args &key :key
```

Function

Destructive version of `subst-if-not`. It changes `tree` by substituting `new` for every subtree or leaf of `tree` that does not satisfy `predicate`. See the function `subst-if-not`. The keyword is:

`:key` If not `nil`, should be a function of one argument that will extract the part to be tested from the whole element.

For example:

```
(setq item-list '((numbers 1.0 2 5/3 symbols foo bar))
  => (NUMBERS 1.0 2 5/3 SYMBOLS FOO BAR)

(nsubst-if-not '3.1415 #' (numbers 1.0 2 5/3 symbols foo bar))
```

```
item-list
```

In the following example, the key function ensures that the test is not applied to the entire list.

```
(setq prop-results '((integer nil nil float))
(nsubst-if-not t #'null prop-results
  :key #'(lambda(x)(and (atom x) x)))
  => (t nil nil t)
```

```
prop-results => (t nil nil t)
```

For a table of related items: See the section "Functions for Modifying Lists".

**nsubstitute**

```lisp
newitem olditem sequence &key (test #eql) :test-not (#key #identity)
:from-end (:start 0) :end :count
```

Function

Returns a sequence of the same type as the argument `sequence` which has the same elements, except that those in the subsequence delimited by `:start` and `:end`
and satisfying the predicate specified by the :test keyword have been replaced by newitem. The argument sequence is destroyed during construction of the result, but the result may or may not be eq to sequence.

For example:

```
(setq letters '(a b c)) => (A B C)
(nsubstitute 'a 'b '(a b c)) => (A A C)
letters => (A B C)
```

However,

```
letters => (A B C)
(nsubstitute 'b 'c letters) => (A B B)
letters => (A B B)
```

newitem and olditem can be any Symbolics Common Lisp object but newitem must be a suitable element for sequence.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence of length zero.

:test specifies the test to be performed. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is true. Where testfun is the test function specified by :test, keyfn is the function specified by :key and x is an element of the sequence. The default test is eql.

For example:

```
(nsubstitute 0 3 '(1 1 4 4 2) :test #'<) => (1 1 0 0 2)
```

:test-not is similar to :test, except that the sense of the test is inverted. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is false.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```
(nsubstitute 1 2 '(((1 1) (1 2) (4 3)) :key #'second) => ((1 1) 1 (4 3))
(nsubstitute 'a 'b '(((a b) (b c) (b b)) :key #'second) => (A (B C) A)
```

A non-nil :from-end specification matters only when the :count argument is provided; in that case only the rightmost :count elements satisfying the test are replaced.

For example:

```
(nsubstitute 'hi 'b '(b a b) :from-end t :count 1 )
=> (B A HI)
```

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:from-end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).
:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

```
(ns substitute 'a 'B '(b a b) :start 1 :end 3) => (B A A)
(ns substitute 'a 'b '(b a b) :end 2) => (A A B)
(ns substitute 'a 'b '(b a b) :end 3) => (A A A)
```

A non-nil :count, if supplied, limits the number of elements altered; if more than :count elements satisfy the test, then of these elements only the leftmost are replaced, as many as specified by :count. A negative :count argument is equivalent to a :count of 0.

For example:

```
(ns substitute 'a 'b '(b b a b b) :count 3) => (A A A A B)
```

To perform destructive substitutions throughout a tree: See the function nsubst.

ns substitute is case-insensitive.

ns substitute is the destructive version of substitute.

For a table of related items: See the section "Sequence Modification".

### ns substitute-if

newitem predicate sequence &key :key :from-end (:start 0) :end :count

`Function`

Returns a sequence of the same type as the argument sequence which has the same elements, except that those in the subsequence delimited by :start and :end and satisfying predicate have been replaced by newitem. The argument sequence is destroyed during construction of the result, but the result may or may not be eq to sequence.

For example:

```
(setq numbers '(a b)) => (A B)
(ns substitute-if 3 #'numberp numbers) => (A B)
numbers => (A B)
```

However,

```
numbers => (1 1 19)
(ns substitute-if 2 #'numberp numbers) => (2 2 2)
numbers => (2 2 2)
```

newitem can be any Symbolics Common Lisp object but must be a suitable element for the sequence.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.
The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```lisp
(nsubstitute-if 1 #'oddp '((1 1) (1 2) (4 3)) :key #'second)
=> (1 (1 2) 1)
```

A non-nil :from-end specification matters only when the :count argument is provided; in that case only the rightmost :count elements satisfying the test are replaced.

For example:

```lisp
(nsubstitute-if 'hi #'atom '(b 'a b) :from-end t :count 1)
=> (B 'A HI)
```

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

```lisp
(nsubstitute-if 1 #'zerop '(0 1 0) :start 1 :end 3) => (0 1 1)
(nsubstitute-if 1 #'zerop '(0 1 0) :start 0 :end 2)  => (1 1 0)
(nsubstitute-if 1 #'zerop '(0 1 0) :end 1)  => (1 1 0)
```

A non-nil :count, if supplied, limits the number of elements altered; if more than :count elements satisfy the test, then of these elements only the leftmost are replaced, as many as specified by :count. A negative :count argument is equivalent to a :count of 0.

For example:

```lisp
(nsubstitute-if 'see 'atom '(b b a b b) :count 3)
=> (SEE SEE SEE B B)
(setq alist (pairlis '(second third start end) '(11 21 13 43)))

(nsubstitute-if '((boundary 42) #'(lambda(x)(member x '(start end middle)))
alist :key #'car))

alist => ((BOUNDARY 42)(BOUNDARY 42)(THIRD 21)(SECOND 11))
```

nsubstitute-if is the destructive version of substitute-if.

For a table of related items: See the section "Sequence Modification".
nsubstitute-if-not newitem predicate sequence &key :key :from-end (start 0) :end :count

Function

Returns a sequence of the same type as the argument sequence which has the same elements, except that those in the subsequence delimited by :start and :end which do not satisfy predicate have been replaced by newitem. The argument sequence is destroyed during construction of the result, but the result may or may not be eq to sequence.

For example:

(setq numbers '(0 0 0)) => (0 0 0)
(nsubstitute-if-not 1 #'numberp numbers) => (0 0 0)
numbers => (0 0 0)

However,

numbers => (1 0 0)
(nsubstitute-if-not 2 #'consp numbers) => (2 2 2)
numbers => (2 2 2)

newitem can be any Symbolics Common Lisp object but must be a suitable element for the sequence.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

(nsubstitute-if-not 1 #'oddp '((1 1) (1 2) (4 3)) :key #'second)
=> ((1 1) 1 (4 3))

A non-nil :from-end specification matters only when the :count argument is provided; in that case only the rightmost :count elements satisfying the test are replaced.

For example:

(nsubstitute-if-not 'hi #'atom ('a 'b) :from-end t :count 1 )
=> ('A 'H)

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).
If both `:start` and `:end` are omitted, the entire sequence is processed by default.

For example:

```lisp
(nsubstitute-if-not 1 #'zerop '(3 0 2) :start 1 :end 3) => (3 0 1)
(nsubstitute-if-not 1 #'zerop '(3 0 2) :start 0 :end 2)  => (1 0 2)
(nsubstitute-if-not 1 #'zerop '(3 0 2) :end 1) => (1 0 2)
```

A non-nil `:count`, if supplied, limits the number of elements altered; if more than `:count` elements satisfy the test, then of these elements only the leftmost are replaced, as many as specified by `:count`. A negative `:count` argument is equivalent to a `:count` of 0.

For example:

```lisp
(nsubstitute-if-not 'see 'consp  '(b b a b b) :count 3)
=> (SEE SEE SEE B B)
```

```lisp
(alist (pairlis '(second third start end) '(11 21 13 43)))

(nsubstitute-if-not '(inner 24) #'(lambda(x)(member x '(start end middle)))
alist :key #'car))
```

```lisp
alist => ((END 43)(START 13)(INNER 24)(INNER 24))
```

`nsubstitute-if-not` is the destructive version of `substitute-if-not`.

For a table of related items: See the section "Sequence Modification".

**nsubstring string from &optional to (area nil)**

Function

Destructive form of the function `substring`. Instead of copying the substring, the system creates an indirect array that shares part of the argument `string`. See the section "Indirect Arrays". Modifying one string modifies the other.

`string` is a string or an object that can be coerced to a string. Since `nsubstring` is destructive, coercion should be used with care since a string internal to the object might be modified. See the function `string`.

Note that `nsubstring` does not necessarily use less storage than `substring`; an `nsubstring` of any length uses at least as much storage as a `substring` four characters long. So you should not use this just "for efficiency"; it is intended for uses in which it is important to have a substring that, if modified, causes the original string to be modified too.

Examples:

```lisp
(setq a "Aloysius") => "Aloysius"
a => "Aloysius"
(setq b (nsubstring a 2 4)) => "oy"
(nstring-upcase b) => "OY"
a => "Aloysius"
```

For a table of related items: See the section "String Access and Information".
nsymbolp arg  

Function

Returns nil if its argument is a symbol, otherwise t.

nth n list  

Function

Returns the nth element of list, where the zeroth element is the car of the list. Examples:

(nth 1 '(foo bar gack)) => bar
(nth 3 '(foo bar gack)) => nil

Returns nil if n is greater than the length of the list.

Note: this is not the same as the Interlisp function called nth, which is similar to, but not exactly the same as, the Symbolics Common Lisp function nthcdr.

nth could have been defined by:

(defun nth (n list)
  (do ((i n (1- i))
       (l list (cdr l)))
      ((zerop i) (car l))))

The relationship between nth and lists is similar to that of svref and simple vectors. However, references beyond the end of the vector are not considered errors by nth.

(nth 0 '(a b c)) = (first '(a b c)) => a
(nth 2 '(a b c)) = (third '(a b c)) => c
(nth 3 '(a b c)) = (fourth '(a b c)) => nil

This function allows selection beyond the cadddr, or even the zl-user:tenth element of a list.

For a table of related items: See the section "Functions for Extracting from Lists".

nthcdr n list  

Function

Performs n cdr operations on list, and returns the result. Examples:

(nthcdr 0 '(a b c)) => (a b c)
(nthcdr 2 '(a b c)) => (c)

In other words, it returns the nth cdr of the list. Returns nil if n is greater than the length of the list.

This is similar to Interlisp's function nth, except that the Interlisp function is one-based instead of zero-based; see the Interlisp manual for details. nthcdr could have been defined by:

(defun nthcdr (n list)
  (do ((i 0 (1+ i))
       (list list (cdr list)))
      ((= i n) list))

This selector function allows selection beyond the cddddr. Though the numeric ar-
gument is evaluated, it allows parameterization of the selected position. Compare the following two forms, and their results, in the following example.

\[
\text{(let ((foo joblist))}
  \text{(dotimes (i *times* foo) (setq foo (cdr foo))))}
\text{(nthcdr *times* joblist)}
\]

For a table of related items: See the section "Functions for Extracting from Lists".

**null**

**null** is the type specifier symbol for the predefined Lisp null data type.

The type null is a **subtype** of the type **symbol**; the only object of type **null** is **nil**.

The types **null** and **cons** form an **exhaustive partition** of the type **list**.

Examples:

\[
\begin{align*}
\text{(typep nil 'null)} & \Rightarrow T \\
\text{(null ())} & \Rightarrow T \\
\text{(subtypep 'null 't)} & \Rightarrow T \text{ and } T \\
\text{(subtypep 'null 'symbol)} & \Rightarrow T \text{ and } T \\
\text{(equal-typep (null ()) (not ()))} & \Rightarrow T \\
\text{(sys:type-arglist 'null)} & \Rightarrow NIL \text{ and } T
\end{align*}
\]

See the section "Data Types and Type Specifiers". See the section "Predicates".

**null x**

Returns **t** if **x** is **nil**, otherwise returns **nil**. **null** is the same as **not**; both functions are included for the sake of clarity. Use **null** to check whether something is **nil**; use **not** to invert the sense of a logical value. Even though Lisp uses the symbol **nil** to represent falseness, you should not make understanding of your program depend on this. For example, one often writes:

\[
\begin{align*}
\text{(cond ((not (null lst)) ... )} & \\
\text{ ( ... )))}
\end{align*}
\]

rather than

\[
\begin{align*}
\text{(cond (lst ... )} & \\
\text{ ( ... )))}
\end{align*}
\]

There is no loss of efficiency, since these compile into exactly the same instructions.

The following example searches a list:

\[
\begin{align*}
\text{(defun my-search(l key)} & \\
\text{ (if (null l)} & \\
\text{ nil} & \\
\text{ (or (equal (car l) key)} & \\
\text{ (search (cdr l) key)))))}
\end{align*}
\]
**sys:null-stream** op &rest args

*Function*

Can be used as a dummy stream object. As an input stream, it immediately reports end-of-file; as an output stream, it absorbs and discards arbitrary amounts of output. Note: **sys:null-stream** is not a variable; it is defined as a function. Use its definition (or the symbol itself) as a stream, not its value. Examples:

```
(stream-copy-until-eof a 'si:null-stream)
(stream-copy-until-eof a #'si:null-stream)
```

Either of the above two forms reads characters out of the stream that is the value of a and throws them away, until a reaches the end-of-file.

**number** &optional (low-limit *) (high-limit *)

*Type Specifier*

**number** is the type specifier symbol for the predefined Lisp data type, number.

The type **number** is a supertype of the following types, which are themselves pairwise disjoint:

- rational
- float
- complex

The types **number**, **cons**, **symbol**, **array**, and **character** are pairwise disjoint.

In addition to a symbol form, Symbolics Common Lisp provides a list form for **number**. Used in list form, **number** allows the declaration and creation of specialized numbers whose range is restricted to the limits specified in the arguments low-limit and high-limit. The list form might not work in other implementations of Common Lisp.

**low-limit** and **high-limit** must each be an integer, a list of an integer, or unspecified. If these limits are expressed as integers, they are inclusive; if they are expressed as a list of an integer, they are exclusive; * means that a limit does not exist, and so effectively denotes minus or plus infinity, respectively.

Examples:

```
(typep '1 'number) => T
(typep 1 '(number 1 3)) => T
(typep 0 '(number 1 3)) => NIL
(typep 4 '(number 5 *)) => NIL
(typep 5 '(number 5 *)) => T
(subtypep 'bit '(number 0 4)) => T and T
(commonp 3.14) => T
(numberp '16) => T
(numberp most-positive-long-float) => T
(subtypep 'rational 'number) => T and T
(subtypep 'float 'number) => T and T
```
(subtypep 'complex 'number) => T and T
(sys:type-arglist 'number)
=> (&OPTIONAL (LOW-LIMIT '*)) (HIGH-LIMIT '*)) and T

See the section "Data Types and Type Specifiers". See the section "Numbers".

sys:number-into-array array n &optional (radix zl:base) (at-index 0) (min-columns 0)

Function

Deposits the printed representation of n into array, which must be a string, which is an integer. sys:number-into-array is the inverse of zl:parse-number. It has three optional arguments:

radix
The radix to use when converting the number into its printed representation. It defaults *print-base*.

at-index
The character position in the array to start putting the number.

min-columns
The minimum number of characters required for the printed representation of the number. If the number contains fewer characters than min-columns, the number is right-justified within the array. If the number contains more characters than min-columns, min-columns is ignored. An error is signalled if the number contains more characters than the length of the array minus at-index. The default is the first position, position 0.

The following example puts 23453243 into string starting at character position 5. Since min-columns is 10, the number is preceded by two spaces.

(let ((string (make-array 20. :type 'art-string :initial-value #
\X))))
   (zl:number-into-array string 23453243. 10. 5. 10.)
string)

=> "XXXXX 23453243XXXXX"

For a table of related items: See the section "String Access and Information".

numberp object

Function

Returns t if its argument is any kind of number, otherwise nil.

The following code first tests whether a and b are numbers. If numbers, they are added, if strings, they are concatenated.

(if (and (numberp a) (numberp b))
 (+ a b)
 (if (and (stringp a) (stringp b))
      (concatenate 'string a b)
      (error "couldn't combine "a and "a b")))
For a table of related items, see the section "Numeric Type-checking Predicates".

**numerator**

If rational is a ratio, numerator returns the numerator of rational. If rational is an integer, numerator returns rational.

Examples:

```
(numerator 4/5) => 4
(numerator 3) => 3
(numerator 4/8) => 1
(numerator (/ 12 -17)) => -12
(numerator (rational 0.200)) => 13421773
```

Related Functions:

**denominator**

For a table of related items: See the section "Functions that Extract Components From a Rational Number".

**nunion** list1 list2 &key (test #eql) test-not (key #identity)

Destructive version of union. It takes two lists and returns a new list containing everything that is an element of either of the lists, and destroys the values of the list arguments. See the function union. The keywords are:

:test Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

For example:

```
(setq a-list 'a b c) => (A B C)
(setq b-list '(f a d)) => (F A D)
(nunion a-list b-list) => (A B C F D)
a-list => (A B C F D)
b-list => (F D)
```

In the following example, nunion updates the list of tenured professors by combin-
ing the list of tenured professors with the list of newly tenured professors.

(setq professors-with-tenure
  '(("Jones" CS101 CS242) ("smith" CS202 CS231)
      ("hunter" CS216 CS232)))
(setq new-tenured-professors
  '(("parks" CS221)))
(setq professors-with-tenure
  (nunion professors-with-tenure new-tenured-professors
    :test #'string-equal :key #'car))

professors-with-tenure =>
  (("Jones" CS201 CS242) ("smith" CS202 CS231)
      ("hunter" CS216 CS232) ("parks" CS221))

For a table of related items: See the section "Functions for Comparing Lists".

zl:nunion &rest lists

Function

Takes any number of lists that represent sets and returns a new list that is the union of all those sets. Destroys the arguments and reuses their conses. zl:nunion uses eq for its comparisons. You cannot change the function used for the comparison. Given no arguments, (nunion) returns nil.

For a table of related items: See the section "Functions for Comparing Lists".

oddp integer

Function

Returns t if integer is odd, otherwise nil. If integer is not an integer, oddp signals an error.

(oddp 1) => t
(oddp (* 2 (random n))) => nil

For a table of related items, see the section "Numeric Property-checking Predicates".

once-only (variable-name ... &environment environment) &body body

Macro

A once-only form looks like this:

(once-only (variable-name &environment environment)
  form1
  form2
  ...)

variable-name is a list of variables. once-only is usually used in macros where the variables are Lisp forms. &environment should be followed by a single variable that is bound to an environment representing the lexical environment in which the
macro is to be interpreted. Typically this comes from the &environment parameter of a macro. The forms are a Lisp program that presumably uses the values of the variables to construct a new form to be the value of the macro. When a call to the macro that includes the once-only form is macroexpanded, the form produced by that expansion will be evaluated.

The macro that includes the once-only form will be macroexpanded. The form produced by that expansion is then evaluated. In the process, the values of each of the variables in variable-name are first inspected. These variables should be bound to subforms, that probably originated as arguments to the defmacro or similar form, and will be incorporated in the macro expansion, possibly in more than one place.

Each variable is then rebound either to its current value, if the current value is a trivial form, or to a generated symbol. Next, once-only evaluates the forms, in this new binding environment, and when they have been evaluated it undoes the bindings. The result of the evaluation of the last form is presumed to be a Lisp form, typically the expansion of a macro. If all of the variables had been bound to trivial forms, then once-only just returns that result. Otherwise, once-only returns the result wrapped in a lambda-combination that binds the generated symbols to the result of evaluating the respective nontrivial forms.

The effect is that the program produced by evaluating the once-only form is coded in such a way that it only evaluates each of the forms that are the values of variables in variable-name once, unless evaluation of the form has no side effects. At the same time, no unnecessary lambda-binding appears in the program. The body of the once-only is not cluttered up with extraneous code to decide whether or not to introduce lambda-binding in the program it constructs.

Note well: once-only can be used only with an &environment keyword argument. If this argument is not present, a compiler warning will result.

For more information about using once-only with &environment: See the lambda list keyword &environment. Also, refer to the definitions of the macro defining forms: defmacro, macrolet, and defmacro-in-flavor.

(defmacro double (x &environment env)
  (once-only (x &environment env)
    '(+ ,x ,x)))
=> DOUBLE

(double 5)
=> (+ 5 5)

(double var)
=> (+ VAR VAR)

(double (compute-value var))
=> (LET ((#:ONCE-ONLY-X-3553 (COMPUTE-VALUE VAR)))
    (+ #:ONCE-ONLY-X-3553 #:ONCE-ONLY-X-3553))

Note that in the first three examples, when the argument is simple, it is duplicat-
ed. In the last example, when the argument is complicated and the duplication could cause a problem, it is not duplicated.

For information about avoiding problems with evaluation: See the section "Avoiding Multiple and Out-of-Order Evaluation".

**once-only** evaluates its subforms in the order they are presented. If it finds any form which is non-trivial, it rebinds the earlier variables to temporaries, and evaluates them first. In the following example, the order of evaluation is \texttt{x}, then \texttt{y}, even though the \texttt{y} appears before the \texttt{x} in the body of the \texttt{once-only}:

\begin{verbatim}
(defmacro my-progn (x y &environment env)
  (once-only (x y &environment env)
    ;; We willfully try to make it evaluate in the wrong order.
    '(progn ,y ,x))) => MY-PROGN

;;Macro expansion shows code that would be produced by the
;;once-only form in the macro.

(my-progn (print x) (setq x 'foo)) =>
(LET ((#:ONCE-ONLY-X-7614 (PRINT X)))
  (PROGN (VALUES (SETQ X 'FOO)) #:ONCE-ONLY-X-7614))
\end{verbatim}

In the next example, \texttt{once-only} evaluates \texttt{y}, then \texttt{x}, because \texttt{y} appears before \texttt{x} in \texttt{once-only}'s variable list. In actuality, this style is an example of poor programming practice as it is confusing. Always list variables in the order in which the forms they are bound to appear in the source that produced them. In a macro, this is normally the order they appear in the macro's argument list.

\begin{verbatim}
(defmacro backward-progn (x y &environment env)
  (once-only (y x &environment env)
    ;; We willfully try to make it evaluate in the wrong order.
    ;; But this time we tell once-only to evaluate y before x.
    '(progn ,y ,x))) => BACKWARD-PROGN

(backward-progn (print x) (setq x 'foo)) => FOO

(FOO
(FOO
(PROGN (VALUES (SETQ X 'FOO)) (VALUES (PRINT X))) => FOO)
\end{verbatim}

\texttt{sys:open-coroutine-stream function \&key (:direction :input) (:buffer-size 1000) (:element-type 'character)}

Function

Creates either input streams, output streams, or bidirectional streams, each with a shared buffer, depending on the argument given to \texttt{:direction}. For examples of coroutine streams, see the section "Coroutine Streams".

Using the functions \texttt{read-char} and \texttt{write-char} on the stream returned by \texttt{sys:open-coroutine-stream} cause the new stack group to be resumed and \texttt{function}
to be called from that stack group. The argument to function is the second stream created by *sys:open-coroutine-stream*. The first stream is the one returned. function should use *read-char* or *write-char* on the stream that is its argument. These functions resume the stack group in which *sys:open-coroutine-stream* was called. In this way function and the caller of *sys:open-coroutine-stream* communicate through the shared buffers; output from one function becomes input to the other.

function takes a single argument, stream, which is the "other end" of the stream returned to the caller by this function. (Note: If more than one argument to function is needed, use lexical scoping.)

:direction can be :input, :output, or :io. To create input coroutine streams use :input; to create output coroutine streams use :output; and to create bidirectional coroutine streams use :io. These are the values accepted by open as specified in Common Lisp: the Language.

:element-type can be any element type acceptable to open.

:buffer-size is the size of the buffer of the intermediate buffer. The value should usually be set to the default size.

• Creating input coroutine streams:

Give :direction the argument :input to create two coroutine streams, an input stream and an output stream, with a shared buffer. *sys:open-coroutine-stream* returns the input stream. The output stream is associated with a new stack group and the input stream with the stack group that is current when *sys:open-coroutine-stream* is called. :tyi messages to the input stream cause the new stack group to be resumed and function to be called from that stack group.

• Creating output coroutine streams:

Give :direction the argument :output to create two coroutine streams, an input stream and an output stream, with a shared buffer. *sys:open-coroutine-stream* returns the output stream. The input stream is associated with a new stack group and the output stream with the stack group that is current when *sys:open-coroutine-stream* is called. Using the cl:write-char function on the output stream causes the new stack group to be resumed and function to be called from that stack group.

• Creating two bidirectional coroutine streams:

Give :direction the argument :io to create two bidirectional coroutine streams. The input buffer of each stream is the output buffer of the other. One stream is associated with a new stack group and the other with the stack group that is current when *sys:open-coroutine-stream* is called. *sys:open-coroutine-stream* returns the stream associated with the current stack group.
**operation-handled-p object operation**  

*Function*  

Returns *non-nil* if the flavor associated with *object* has a method defined for *operation* and *nil* otherwise. *operation* is a message or the name of a generic function.

Note that *operation-handled-p* works by sending the :operation-handled-p message. You can customize the behavior of *operation-handled-p* by defining a method for the :operation-handled-p message.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**:operation-handled-p operation**  

*Message*  

*operation* is a message or the name of a generic function. The object should return *non-nil* if it has a handler for the operation, and *nil* if it does not.


Instead of sending this message, you can use the *operation-handled-p* function. See the function *operation-handled-p*.

Note that *operation-handled-p* works by sending the :operation-handled-p message. You can customize the behavior of *operation-handled-p* by defining a method for the :operation-handled-p message.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

**optimize (option1 value1) (option2 value2) ...**  

*Declaration*  

Advises the compiler to give attention to each *option* according to its associated *value*. *value* should be an integer between 0 and 3, where 0 means that *option* is totally unimportant and 3 means that it is extremely important. 1 and 2 are intermediate, with 1 being the usual or normal value. You may abbreviate (*option 3*) to *option*.

**compilation-speed**  

*Option*  

Speed of the compilation process.

**safety**  

*Option*  

Run time error-checking.

**space**  

*Option*  

Code size and run-time space.
speed

How fast the object code runs.
See the section "Declaration Specifiers".

lt:optimize-state name &optional env

Returns the value of the optimization quality name in the given environment. If env is omitted, the current environment is used.
See the section "Declarations".

&optional

If the lambda-list keyword &optional is present, all specifiers up to the next lambda-list keyword, or the end of the list, are optional parameter specifiers.

or

or &rest forms

Evaluates each form one by one, from left to right. If a form evaluates to nil, or proceeds to evaluate the next form. If there is no other form, or returns nil. But if a form evaluates to a non-nil value, or immediately returns that value without evaluating any other form.

As with and, or can be used either as a logical or function, or as a conditional.

Examples:

(or) => NIL
(or 'start 'finish 'middle) => START
(or (> 3 4)) => NIL
(or (numberp 'arg) "not a number") => "not a number"
(or it-is-fish
   it-is-fowl
   (print "It is neither fish nor fowl."))

In the following example, very-expensive-function is not evaluated because a prior form is true:

(setq foo 12 bar '(3 4 5))

(if (or (eql foo bar)
        (eql 12 foo)
        (very-expensive-function bar))
   bar
   foo)
Note: (or) => nil , the identity for this operation.

For a table of related items: See the section "Conditional Functions".

CLOE Note: This is a macro in CLOE.

**output-stream-p stream**

*Function*

Returns t if stream can handle output operations, and otherwise it returns nil.

```
(setq file-stream
     (open "foo" :direction :output :element-type 'character))

(output-stream-p file-stream) => T
```

**package**

*Type Specifier*

*package* is the type specifier symbol for the predefined Lisp data type of that name.

The types *package*, *hash-table*, *readtable*, *pathname*, *stream*, and *random-state* are pairwise disjoint.

Examples:

```
(typep *package* 'package) => T
(typep (in-package 'example) 'package) => T
(typep (in-package 'cl-user) 'package) => T
(typep (find-package 'cl-user) 'package) => T
(zl:typep *package*) => ZL:PACKAGE
(sys:type-arglist 'package) => NIL and T
```

See the section "Data Types and Type Specifiers". See the section "Packages".

**package**

*Variable*

The value is the current package; many functions that take packages as optional arguments default to the value of *package*, including *intern* and related functions. The reader and the printer deal with printed representations that depend on the value of *package*. Hence, under Genera, the current package is part of the user interface and is displayed in the status line at the bottom of the screen.

It is often useful to bind *package* to a package around some code that deals with that package. The operations of loading, compiling, and editing a file all bind *package* to the package associated with the file.

**zl:package**

*Variable*
See *package*.

**sys:package-cell-location** *symbol*  
*Function*  
Returns a locative pointer to *symbol*’s package cell. It is preferable to write the following, rather than calling this function explicitly.

```
(locf (symbol-package *symbol*))
```

See the section “The Package Cell of a Symbol”.

**sys:package-error**  
*Flavor*  
All package-related error conditions are built on **sys:package-error**.

**package-external-symbols** *package*  
*Function*  
A list of all the external symbols exported by *package*. *package* can be a package object or the name of a package (a symbol or a string).

**sys:package-locked**  
*Flavor*  
There was an attempt to intern a symbol in a locked package.

The :symbol message returns the symbol. The :package message returns the package.

The :no-action proceed type interns the symbol just as if the package had not been locked. Other proceed types are also available when interning the symbol would cause a name conflict.

**package-name** *pkg*  
*Function*  
Returns the name of *pkg* as a string. *pkg* must be a package object.

```
(find-package 'cl-user)  
  => #<Package USER (really COMMON-LISP-USER) 32720604>  
(find-package *) => "USER"  
  => (package-name (find-package "cloe"))  
  "cloe"
```

See the section “Mapping Between Names and Packages”.

**package-nicknames** *pkg*  
*Function*  
Returns the acceptable nickname strings for *pkg*. *pkg* must be a package object.

```
(find-package "common-lisp") => #<Package COMMON-LISP 35553744>  
(find-package-nicknames *) => ("COMMON-LISP-GLOBAL" "CL" "LISP")
```

In the following example, the name of a package is compared for length with the
nicknames of the package and the shortest name is returned.

(defun short-package-name (package)
  (let ((short-name (package-name package)))
    (dolist (nickname (package-nicknames package))
      (if (< (length nickname) (length short-name))
        (setq short-name nickname)))
    short-name))

sys:package-not-found

A package-name lookup did not find any package by the specified name.
The :name message returns the name. The :relative-to message returns nil if only absolute names are being searched, or else the package whose relative names are also searched.

The :no-action proceed type can be used to try again. The :new-name proceed type can be used to specify a different name or package. The :create-package proceed type creates the package with default characteristics.

package-shadowing-symbols package

The list of symbols that have been declared as shadowing symbols in this package by shadow or shadowing-import. All symbols on this list are present in the specified package. package can be a package object or the name of a package (a symbol or a string).

The following function checks if a list of symbols has already been made shadowing symbols of the indicated package, and if not, calls shadow.

(defun show-shadowed-symbols (package)
  (let ((shadowing-symbols (package-shadowing-symbols package)))
    (format t "~&The package ~A has ~D shadowing symbol~:P.~%"
            (package-name package) (length shadowing-symbols))
    (dolist (symbol shadowing-symbols)
      (let ((shadowed-symbols '())
            (name (symbol-name symbol)))
        (dolist (package (package-use-list package))
          (let ((shadowed-symbol (find-symbol name package)))
            (if (and shadowed-symbol (not (eq shadowed-symbol symbol)))
              (pushnew shadowed-symbol shadowed-symbols)))))
    (format t "~S shadows~:| no symbols~:|:*: ~(~"S", ~)~:|:"~%"
            symbol shadowed-symbols))))

package-use-list pkg

The list of other packages used by the argument package. pkg must be a package object. The elements of the list returned are package objects.
See the section "Interpackage Relations".

**package-used-by-list** \( pkg \)  
Function

The list of other packages that use the argument package. \( pkg \) can be a package object or the name of a package (a symbol or a string). The elements of the list returned are package objects.

The following example defines a function which prints information about the packages used by its argument package.

```lisp
(defun show-packages-using (package)
  (format t "The package ~A is used by: ~{~A~^, ~}~"
          (package-name package)
          (mapcar #'package-name (package-used-by-list package))))
```

See the section "Interpackage Relations".

**packagep** \( object \)  
Function

Returns \( t \) if \( object \) is a package. (packagep \( x \)) is equivalent to (typep \( x \) 'package).

```lisp
(setq foo (make-package 'turbine-package))
(packagep foo) => t
```

In the next example, the argument to packagep is a package name rather than a package object.

```lisp
(packagep (find-package 'turbine-package)) => t
(packagep "turbine-package") => nil
```

**sys:page-in-raster-array** \( raster \)  &optional from-x from-y to-x to-y (hang-p si:*default-page-in-hang-p*) (normalize-p t)
Function

Ensures that the storage that represents \( raster \) is in main memory. from-x and from-y can be specified as nil, meaning the lower limit for that item. to-x and to-y can be specified as nil, meaning the upper limit for that item.

This, rather than **sys:page-in-array**, should be used on rasters.

For a table of related items: See the section "Operations on Rasters".

**sys:page-in-table** \( table \)  &key :type :hang-p
Function

Brings back into main memory any swapped pages in \( table \) that have been swapped out to disk.

:type defaults to page-in-type.
If `hang-p` is `t`, the function waits for the disk reads to finish before returning. Otherwise, the function returns immediately after requesting the disk reads, which might still be in progress. Thus, `hang-p` causes the process to hang until the input/output is complete, that is, until all the requested pages are there. The default value, `page-in-hand-p` is `t` by default.

**sys:page-out-raster-array**

`array &optional from-x from-y to-x to-y (hang-p si:*default-page-in-hang-p*)`  
Function

Takes the pages that represent `raster` out of main memory. `from-x` and `from-y` can be specified as `nil`, meaning the lower value for that item. `to-x` and `to-y` can be specified as `nil`, meaning the upper limit for that item.

This, rather than `sys:page-out-array`, should be used on rasters.
For a table of related items: See the section "Operations on Rasters".

**sys:page-out-table**

`table &key :write-modified :reuse`  
Function

Takes all swapped pages in `table` out of main memory.

`:write-modified` defaults to `write-modified`.
`:reuse` defaults to `reuse`.

**pairlis**

`keys data &optional a-list`  
Function

Takes two lists and associates elements of the first list to corresponding elements of the second list, creating an association list. `pairlis` signals an error if the two lists, `keys` and `data`, are not of the same length. If the optional argument `a-list` is provided, then the new pairs are added to the front of `a-list`.

The new pairs can appear in the resulting association list in any order; in particular, either forward or backward order is permitted. Therefore, the result of the following call might be either of the two results.

```
(pairlis '(one two) '(1 2) '((three . 3) (four . 4))) =>
((TWO . 2) (ONE . 1) (THREE . 3) (FOUR . 4))
or
((ONE . 1) (TWO . 2) (THREE . 3) (FOUR . 4))
```

The following example demonstrates an association list consisting of pairs of keys and association lists.

```
(setq keys '(monthly-cash-on-hand monthly-expense monthly-revenue))
(setq data '((,(pairlis '(11 12) '(52 73))
             ,(pairlis '(10 11) '(20 21))
             ,(pairlis '(10 11) '(31 42))))
```
(setq financial-statement (pairlis keys data)) =>
((MONTHLY-CASH-ON-HAND ((11 . 52) (12 . 73)))
 (MONTHLY-EXPENSE ((10 . 20) (11 . 21)))
 (MONTHLY-EXPENSE ((10 . 31) (11 . 42))))

For a table of related items: See the section "Functions that Operate on Association Lists".

zl:pairlis vars vals

Function
Takes two lists and makes an association list which associates elements of the first list with corresponding elements of the second list. Example:

(zl:pairlis '(beef clams chicken) '(roast fried yu-hsiang))
=> ((beef . roast) (clams . fried) (chicken . yu-hsiang))

For a table of related items: See the section "Functions that Operate on Association Lists".

zl:parse-ferror format-string &rest format-args

Function
Signals an error of flavor zl:parse-ferror. format-string and format-args are passed as the :format-string and :format-args init options to the error object.

See the flavor zl:parse-ferror.

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

parse-integer string &key (start 0) :end (radix 10) :junk-allowed (sign-allowed t)

Function
Examines the substring of string delimited by :start and :end (which default to the beginning and end of the string). It skips over whitespace and then attempts to parse an integer. The :radix argument defaults to 10, and must be an integer between 2 and 36.

If :junk-allowed is nil (the default), then the entire substring is scanned. The returned value is the value of the number parsed as an integer. An error is signalled if the substring does not consist entirely of the representation of an integer, possibly surrounded on either side by whitespace characters.

If :junk-allowed is non-nil, the first value returned is the value of the number parsed as an integer, or nil if no syntactically correct integer was seen.

In either case, the second value returned is the index into the string of the delimiter that terminated the parse, or it is the index beyond the substring if the parse terminated at the end of the substring (as will be the case of :junk-allowed is nil).
Note that `parse-integer` does not recognize the syntactic radix-specifier prefixes `#o`, `#b`, `#x`, and `#nR`, nor does it recognize a trailing decimal point. It permits only an optional sign (`+` or `-`) followed by a non-empty sequence of digits in the specified radix. For example:

```lisp
(parse-integer "-1234567890" :start 3) => 234567890 and 13
(parse-integer "345")
=> 345

(parse-integer "345" :radix 8)
=> 229

(parse-integer "345a")
Error: Garbage character a seen while parsing integer in "345a"

(parse-integer "345a" :junk-allowed t)
=> 345

(parse-integer "345a" :radix 16)
=> 13402
```

For a table of related items: See the section "String Access and Information".

`zl:parse-number` `string` &optional (from 0) to radix fail-if-not-whole-string  

Function

Takes a string and "reads" a number from it. The function currently does not handle anything but integers.

`string` must be a string. It returns two values: the number found (or `nil`) and the character position of the next unparsed character in the string. It returns `nil` when the first character that it looks at cannot be part of a number. (`read-from-string` is a more general function that uses the Lisp reader; `prompt-and-read` reads a number from the keyboard.)

Four optional arguments:

- `from`  
The character position in the string to start parsing. The default is the first one, position 0.

- `to`  
The character position past the last one to consider. The default, `nil`, means the end of the string.

- `radix`  
The radix to read the string in. The default, `nil`, means base 10.

- `fail-if-not-whole-string`  
The default is `nil`. `nil` means to read up to the first character that is not a digit and stop there, returning the result of the parse so far. `t` means to stop at the first nondigit and to return `nil` and 0 length if that is not the end of the string.

Examples:
The Common Lisp equivalent of `zl:parse-number` is `parse-integer`.
For a table of related items: See the section "String Access and Information".

**pathname**

*Type Specifier*

`pathname` is the type specifier symbol for the predefined Lisp data type of that name.

The types `pathname`, `hash-table`, `readtable`, `package`, `stream`, and `random-state` are pairwise disjoint.

Examples:

```
(typep (pathname "apple") 'pathname) => T
(type-of (pathname "bubbles")) => FS:LMFS-PATHNAME
(sys:type-arglist 'pathname) => NIL
(pathnamep *default-pathname-defaults*) => T
```

See the section "Data Types and Type Specifiers". See the section "Files".

**:pathname**

*Message*

Returns the pathname that was opened to get this stream. This might not be identical to the argument to `open`, since missing components will have been filled in from defaults, and the pathname might have been replaced wholesale if an error occurred in the attempt to open the original pathname.

**phase number**

*Function*

Returns a single-precision result, unless `number` is a double-precision complex number. The phase of a number is the angle part, in radians, of its polar representation as a complex number. The phase of zero is arbitrarily defined to be zero.

`phase` could have been defined as:

```
(defun phase (number)
  (atan (imagpart number) (realpart number)))
```

Thus, the phase of any non-negative non-complex number is zero, and the phase of any non-complex negative `number` is \( \pi \). Complex values of `number` can result in other values in the range of -\( \pi \) to \( \pi \).

See the function `abs`. 
For a table of related items: See the section "Trigonometric and Related Functions".

**pi**  
**Constant**

The value of constant *pi* is the best possible approximation to \( \pi \) in double floating-point format.

To obtain an approximation to \( \pi \) in some other precision, use \((\text{float pi } x)\) where \( x \) is a floating-point number of the desired precision; or use \((\text{coerce pi type})\) where \( \text{type} \) is the name of a valid floating-point precision type.

Note that in CLOE, *pi* has single-float precision. Examples:

\[
\text{pi } => 3.141592653589793d0
\]
\[
(\text{float pi } 1.0) => 3.1415927
\]
\[
(\text{float pi } 1.0L0) => 3.141592653589793d0
\]
\[
(\text{coerce pi } '\text{single-float}) => 3.1415927
\]

**pkg-add-relative-name from-pkg name to-pkg**  
**Function**

Adds a relative name named *name*, a string or a symbol, that refers to *to-pkg*. From now on, qualified names using *name* as a prefix, when the current package is *from-package* or a package that uses *from-pkg*, refer to *to-pkg*.

*from-pkg* and *to-pkg* can be packages or names of packages.

It is an error if *from-pkg* already defines *name* as a relative name for a package different from *to-pkg*.

See the section "Interpackage Relations".

**zl:pkg-bind pkg body...**  
**Macro**

Evaluates the forms of the *body* with the variable *package* bound to the package named by *pkg*. Returns the values of the last form. *pkg* can be a package or a package name.

Example:

\[
(\text{zl:pkg-bind } "\text{zwei}\)"
\]
\[
(\text{read-from-string function-name}))
\]

The difference between **zl:pkg-bind** and a simple let of the variable *package* is that **zl:pkg-bind** ensures that the new value for *package* is actually a package; it coerces package names (strings or symbols) into actual package objects.

**pkg-delete-relative-name from-pkg name &optional to-pkg**  
**Function**
If `from-pkg` defines `name` as a relative name, it is removed. `from-pkg` can be a package or the name of a package. `name` can be a symbol or a string. It is not an error if `from-pkg` does not define `name` as a relative name.

See the section "Interpackage Relations".

**pkg-find-package**  
`thing` &optional `(create-p :error) relative-to syntax`  
**Function**

Tries to interpret `thing` as a package. Most of the functions whose descriptions say "... can be either a package or the name of a package" call `pkg-find-package` to interpret their package argument.

If `thing` is a package, `pkg-find-package` returns it.

If `thing` is a symbol or a string, it is interpreted as the name of a package. If `relative-to` is specified and non-nil, then it must be a package or the name of a package. If `relative-to` or one of the packages it uses has a relative name of `thing`, the package named by that relative name is used. If the relative name search fails, or if no relative name search is called for (that is, `relative-to` is `nil`, which is the default), then if a package with a primary name or nickname of `thing` exists it is returned.

If `thing` is a list, it is presumed to have come from a file attribute line. `pkg-find-package` is done on the car of the list. If that fails, a new package is created with that name, according to the specifications in the rest of the list. See the section "Specifying Packages in Programs".

If no package is found, the `create-p` argument controls what happens. Note that this can only happen if `thing` is a symbol or a string. The possible values for `create-p` are:

- **:error** or `nil`  
  A `sys:package-not-found` error is signalled. See the flavor `sys:package-not-found`. The error can be continued by defining the package manually, creating it automatically with default attributes, or using a different package name instead. **:error** is the default. `nil` is accepted as a synonym for **:error** for backwards compatibility.

- **:find**  
  Just returns `nil`.

- **:ask**  
  Asks the user whether to create it. Replying No to the **:ask** query is the same as **:error**, a `sys:package-not-found` error is signalled.

- **t**  
  Creates a package with the specified name with attributes determined by `relative-to` and `syntax`. If `relative-to` and `syntax` are omitted, the new package inherits from `global` but not from any other packages.

`relative-to` is a package object, or a string or symbol that names a package object  
`syntax` is a Lisp syntax object, obtained by using `si:lisp-syntax-from-keyword`. See the function `si:lisp-syntax-from-keyword`. 
(pkg-find-package "my-package" t "cl-user"
   (si:lisp-syntax-from-keyword :common-lisp))

The package name search is independent of alphabetic case. However, it is not considered good style to have two distinct packages whose names differ only in alphabetic case.

**ZL:pkg-global-package**

The *global* package.

**ZL:pkg-goto** &optional *pkg* *globally*  

*pkg* can be a package or the name of a package. *pkg* is made the current package; in other words, the variable *package* is set to the package named by *pkg*. *ZL:pkg-goto* can be useful to "put the keyboard inside" a package when you are debugging.

*pkg* defaults to the *user* package.

If *globally* is specified non-nil, *package* is set with *ZL:setq-globally* instead of *setq*. This is useful mainly in an init file, where you want to change the default package for user interaction, and a simple *setq* of *package* does not work because it is bound by *load* when it loads the init file.

*package* is equivalent to *ZL:package*.

**Sys:pkg-keyword-package**

The *keyword* package.

**PKG-kill**  

Kills *package* by removing it from all package system data structures. The name and nicknames of *package* cease to be recognized package names. If *package* is used by other packages, it is un-used, causing its external symbols to stop being accessible to those packages. If other packages have relative names for *package*, the names are deleted.

Any symbols in *package* still exist and their home package is not changed. If this is undesirable, evaluate *(ZL:mapatoms #ZL:remob *package* nil)* first.

*package* can be a package or the name of a package.

**ZL:pkg-name**  

Returns the (primary) name of *package* as a string. *package* should be a package object. However, *ZL:pkg-name* is a structure-accessing function and does not check that its argument is a package object, only that it is some kind of an array with a leader. If the argument is not a package object, the results are unpredictable.
The Common Lisp equivalent of \texttt{zl:pkg-name} is \texttt{package-name}. \texttt{package-name} does check that its argument is a package object. See the function \texttt{package-name}. See the section "Mapping Between Names and Packages".

\texttt{zl:pkg-system-package} \hspace{1cm} \textit{Variable}

The \texttt{system} package.

\texttt{plane-aref} \texttt{plane} \&rest \texttt{point} \hspace{1cm} \textit{Function}

Returns the contents of a specified element of a plane. \texttt{plane-aref} takes the subscripts as arguments. \texttt{setf} of \texttt{plane-aref} is allowed.

For a table of related items, see the section "Operations on Planes".

\texttt{zl:plane-aset} \texttt{datum} \texttt{plane} \&rest \texttt{point} \hspace{1cm} \textit{Function}

Stores \texttt{datum} into the specified element of a plane, extending it if necessary, and returns \texttt{datum}. \texttt{zl:plane-aset} differs from \texttt{zl:plane-store} in the way it takes its arguments; \texttt{zl:plane-aset} takes the subscripts as arguments, while \texttt{zl:plane-store} takes a list of subscripts. \texttt{setf} of \texttt{plane-aref} is preferred.

\texttt{plane-default} \texttt{plane} \hspace{1cm} \textit{Function}

Returns the contents of the infinite number of plane elements that are not actually stored.

For a table of related items, see the section "Operations on Planes".

\texttt{plane-extension} \texttt{plane} \hspace{1cm} \textit{Function}

Returns the amount to extend the plane by in any direction when \texttt{zl:plane-store} is done outside of the currently stored portion.

For a table of related items, see the section "Operations on Planes".

\texttt{zl:plane-origin} \texttt{plane} \hspace{1cm} \textit{Function}

Returns a list of numbers, giving the lowest coordinate values actually stored.

\texttt{zl:plane-ref} \texttt{plane} \texttt{point} \hspace{1cm} \textit{Function}

Returns the contents of a specified element of a plane. It differs from \texttt{plane-aref} in the way that it takes its arguments; \texttt{plane-aref} takes the subscripts as arguments, while \texttt{zl:plane-ref} takes a list of subscripts.
**zl:plane-store** *datum plane point*  
*Function*
Stores *datum* into the specified element of a plane, extending it if necessary, and returns *datum*. **zl:plane-store** differs from **zl:plane-aset** in the way it takes its arguments; **zl:plane-aset** takes the subscripts as arguments, while **zl:plane-store** takes a list of subscripts.

**zl:plist** *symbol*  
*Function*
Returns the list that represents the property list of *symbol*. Note that this is not the property list itself; you cannot do **get** on it.

The Common Lisp equivalent of this function is **symbol-plist**. See the section "Functions Relating to the Property List of a Symbol".

**zl:plus** &rest args  
*Function*
Returns the sum of its arguments. If there are no arguments, it returns 0, which is the identity for this operation.

The following functions are synonyms of **zl:plus**:
```
+  
zl:+$  
```

**plusp** *number*  
*Function*
Returns t if its argument is a positive number, strictly greater than zero. Otherwise it returns nil. If *number* is not a noncomplex number, **plusp** causes an error.

```
(plusp 1.0) => t  
(plusp 0) => nil  
(plusp -3) => nil  
(plusp least-negative-single-float) => nil  
(plusp least-positive-single-float) => t
```

For a table of related items, see the section "Numeric Property-checking Predicates".

**pop** *list*  
*Function*
Returns the car of the contents of *list*, and as a side effect, the cdr of contents is stored back into *list*. The form *list* can be any form acceptable as a generalized variable to **setf**. If *list* is viewed as a push-down stack, **pop** can be thought of as popping an element from the top of the stack and returning it. For example:

```
(setq stack '(a b c)) => (A B C)  
(pop stack) => A
```
For a table of related items: See the section "Functions for Extracting from Lists".

For a table of related items: See the section "Functions for Modifying Lists".

\textbf{zl:pop list &optional dest} \textit{Function}

Returns the car of the contents of \textit{list}, and as a side effect, the cdr of contents is stored back into \textit{list}. The form \textit{list} can be any form acceptable as a generalized variable to \texttt{setf}. If \textit{list} is viewed as a push-down stack, \texttt{pop} can be thought of as popping an element from the top of the stack and returning it. For example:

\begin{verbatim}
(setq stack '(a b c)) => (A B C)
(pop stack) => A
stack => (B C)
\end{verbatim}

The caveat that applies to \texttt{incf} also applies to \texttt{zl:pop} as well; \texttt{zl:pop} does not evaluate any part of the ref more than once.

\textbf{position item sequence &key (:test eql) :test-not (:key identity) :from-end (start 0) :end} \textit{Function}

If \textit{sequence} contains an element satisfying the predicate specified by the \texttt{:test} keyword, \texttt{position} returns the index within the sequence of the leftmost such element as a non-negative integer; otherwise \texttt{nil} is returned.

\textit{item} is matched against the elements specified by the \texttt{test} keyword. The \textit{item} can be any Symbolics Common Lisp object but must be a suitable element for the \textit{sequence}.

\textit{sequence} can be either a list or a vector (one-dimensional array). Note that \texttt{nil} is considered to be a sequence, of length zero.

\texttt{:test} specifies the test to be performed. An element of \textit{sequence} satisfies the test if \((\text{funcall testfun item (keyfn x)})\) is true. Where \texttt{testfun} is the test function specified by \texttt{:test}, \texttt{keyfn} is the function specified by \texttt{:key} and \texttt{x} is an element of the \textit{sequence}. The default test is \texttt{eql}.

For example:

\begin{verbatim}
(position 1 #(3 2 1 2) :test #'eq) => 2
\end{verbatim}

\texttt{:test-not} is similar to \texttt{:test}, except that the sense of the test is inverted. An element of \textit{sequence} satisfies the test if \((\text{funcall testfun item (keyfn x)})\) is false.

The value of the keyword argument \texttt{:key}, if non-\texttt{nil}, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.
For example:

(position 'c #(1 a) (2 b) (3 c) :key #'second) => 2

If the value of the :from-end argument is non-nil, the result is the index of the rightmost element that satisfies the predicate, however, the index is still computed from the left-hand end of the sequence.

For example:

(position 3 #(2 2 3 4 4 3) :from-end 'non-nil) => 5
(position 3 #(2 2 3 4 4 3) :from-end nil) => 2

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence). If :end is unspecified or nil, the length sequence is used.

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(position 'a #(b b a b b)) => 2
(position 'a #(b b a b b)) => 2
(position 2 #(2 3 3 2 3) :start 2) => 3
(position 3 #(2 1 1 2) :start 1 :end 4) => NIL
(setq vector-1 (vector 'foo 'bar 'baz 'boz)
  vector-2 (vector 3 2 4 5 1 7 6))
(replace vector-1 vector-2 :start2 (position 4 vector-2))
=> #(4 5 1 7)

For a table of related items: See the section "Searching for Sequence Items".

**position-if** predicate sequence &key :key :from-end (:start 0) :end

Function

If sequence contains an element satisfying predicate, then position returns the index within the sequence of the leftmost such element as a non-negative integer; otherwise nil is returned.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:
If the value of the :from-end argument is non-nil, then the result is the index of the rightmost element that satisfies the predicate, however, the index is still computed from the left-hand end of the sequence.

For example:

(position-if #'numberp '(1 a b c 3) :from-end 'non-nil) => 4
(position-if #'numberp '(a 1 b c 3) :from-end nil) => 1

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(position-if #'numberp '(2 a b c 3) :start 2) => 4
(position-if #'numberp '(2 a b c 2) :start 1 :end 4) => NIL

(setq text "It was the height, of folly; Was it not?"
(setq pos (position-if #'upper-case-p text :start 1)) => 29
(setf (elt text pos) (char-downcase (elt text pos))) => #\w
text => "It was the height, of folly; was it not?"

For a table of related items: See the section "Searching for Sequence Items".

(position-if-not predicate sequence &key :key :from-end (start 0) :end Function)

If sequence contains an element that does not satisfy predicate, position returns the index within the sequence of the leftmost such element as a non-negative integer; otherwise nil is returned.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:
(position-if-not #'zerop #((1 a) (0 b) (3 c)) :key #'car)
=> 0

If the value of the :from-end argument is non-nil, the result is the index of the rightmost element that satisfies the predicate, however, the index is still computed from the left-hand end of the sequence.

For example:

(position-if-not #'numberp #(1 a b c 3) :from-end 'non-nil)  => 3
(position-if-not #'numberp #(a 1 b c 3) :from-end nil) => 0

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

: start and :end must be non-negative integer indices into the sequence. : start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

: start indicates the start position for the operation within the sequence. : end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(position-if-not #'numberp #(2 a b c 3) :start 2) => 2
(position-if-not #'numberp #(a 1 2 3 a) :start 1 :end 4) => NIL

(setq text "It was the height, of folly; was it not?")

(setq pos (position-if-not
    #'(lambda(x)(or (alpha-char-p x)(char= x #\Space))) text))

(replace text text :start1 pos :start2 (+ pos 1))

=> "It was the height of folly; was it not??"

For a table of related items: See the section "Searching for Sequence Items".

pprint object &optional output-stream

Function

Writes the printed representation of object to the output-stream using the pretty printer. The printed representation is preceded by a newline and escape characters are used as appropriate. pprint returns no values. For example:

(pprint "A simple string") =>
"A simple string"

output-stream, which, if unspecified or nil, defaults to *standard-input*, and if t, defaults to *terminal-io*. 
(PPRINT (LOOP FOR I FROM 1 TO 5 COLLECT
   (LOOP FOR I FROM 1 TO 10 COLLECT #\X)))

might print something like
((#\X #\X #\X #\X #\X #\X #\X #\X #\X #\X)
 (#\X #\X #\X #\X #\X #\X #\X #\X #\X #\X)
 (#\X #\X #\X #\X #\X #\X #\X #\X #\X #\X)
 (#\X #\X #\X #\X #\X #\X #\X #\X #\X #\X)
 (#\X #\X #\X #\X #\X #\X #\X #\X #\X #\X))

prin1

Variable

The value of this variable is normally nil. If it is non-nil, then the read-eval-print loop uses its value instead of the definition of prin1 to print the values returned by functions. This hook lets you control how things are printed by all read-eval-print loops — the Lisp top level and any utility programs that include a read-eval-print loop. It does not affect output from programs that call the prin1 function or any of its relatives such as print and format; to do that, you need more information on customizing the printer. See the section "Output Functions". If you set prin1 to a new function, remember that the read-eval-print loop expects the function to print the value but not to output a Return character or any other delimiters.

**prin1 object &optional output-stream**

Function

Outputs the printed representation of object to stream, with slashification. Roughly speaking, the output from prin1 is suitable for input to the function zl:read. prin1 returns object.

output-stream, if unspecified or nil, defaults to *standard-input*, and if t, defaults to *terminal-io*.

See the section "What the Printer Produces".

For example:

```
(prin1 "A simple string") => "A simple string"
"A simple string"
(prin1 'foo) prints FOO
(prin1 "foo") prints "foo"
(prin1 #\c) prints #\c
```

zl:prin1-then-space object &optional output-stream

Function

Like prin1 except that output is followed by a space. zl:prin1-then-space returns object. For example:

```
(zl:prin1-then-space "A simple string") => "A simple string"
"A simple string"
```
## prin1-to-string object

The object is printed as if by **prin1**, and the characters that would be output are made into a string, which is returned. For example:

- `(prin1-to-string '|red|)` => `"\"|red|\"`
- `(prin1-to-string #'\A)` => `"\A"

```lisp
(let ((*print-escape* t))
  (list (prin1-to-string #'\A)
     (progn (setq *print-escape* nil) (prin1-to-string #'\A))))
=> ("\A" "\A")
```

## princ object &optional output-stream

Like **prin1** except that the output is not slashified. A symbol is printed as simply the characters of its print name, a string is printed without surrounding double quotes, and so on. The general rule is that output from **princ** is intended to look good to people, while output from **prin1** is intended to be acceptable to the function **read**. **princ** returns **object**.

- `(princ "A simple string")` => *A simple string*
- `"A simple string"

**output-stream**, which, if unspecified or **nil**, defaults to **standard-input**, and if **t**, is **terminal-io**.

- `(princ 'foo)` *prints* FOO
- `(princ "foo")` *prints* foo
- `(princ #'\c)` *prints* c

## princ-to-string object

The object is printed as if by **princ**, and the characters that would be output are made into a string, which is returned. For example:

- `(princ-to-string '|red|)` => `|red|`
- `(let ((*print-escape* t))
     (list (princ-to-string #'\A)
       (progn (setq *print-escape* nil) (princ-to-string #'\A))))
=> ("A" "A")

## zl:prinlength

Can be set to the maximum number of list elements to be printed before the printer just prints "...". If it is **nil**, which it is initially, a list of any length can be printed. Otherwise, the value of **zl:prinlength** must be an integer. This variable is superseded by **print-length**.
**z1:prinlevel**

Variable

Can be set to the maximum number of nested lists to be printed before the printer just prints "***". If it is nil, which it is initially, any number of nested lists can be printed. Otherwise, the value of **z1:prinlevel** must be an integer. This variable is superseded by **print-level**.

**print object &optional output-stream**

Function

Like **prin1** except that output is preceded by a Newline and followed by a space. **print** returns object. For example:

```lisp
(print "A simple string") =>
  "A simple string"
  "A simple string"
```

output-stream, which, if unspecified or nil, defaults to *standard-input*, and if t, defaults to *terminal-io*.

```lisp
(PRINT (LOOP FOR I FROM 1 TO 5 COLLECT
  (LOOP FOR I FROM 1 TO 10 COLLECT #\x)))
would print something like

((#\x #\x #\x #\x #\x #\x #\x #\x #\x #\x) (#\x #\x #\x #\x #\x #\x #\x #\x #\x #\x) (#\x #\x #\x #\x #\x #\x #\x #\x #\x #\x) (#\x #\x #\x #\x #\x #\x #\x #\x #\x #\x) (#\x #\x #\x #\x #\x #\x #\x #\x #\x #\x))

(PROGN (PRIN1 'A) (PRIN1 'B) (PRIN1 'C)
  (PRINT 'D) (PRINT 'E) (PRINT 'F))

prints

ABC
D
E
F
```

**print-abbreviate-quote**

Variable

Provides a way to print quoted forms in their short form. It is incorporated into **print-pretty**, so the value of **print-pretty** must be nil in order for **print-abbreviate-quote** to have any effect.

Examples:

```lisp
(let ((**print-abbreviate-quote** nil)
  (**print-pretty** nil))
  (print '(quote foo)) nil) => (QUOTE FO0) NIL
(let ((*print-abbreviate-quote* t)
      (*print-pretty* nil))
  (print '(quote foo)) nil) => 'FOO NIL

(let ((*print-abbreviate-quote* t)
      (*print-pretty* nil))
  (print '(function foo)) nil) => #'FOO NIL

(let ((*print-abbreviate-quote* t)
      (*print-pretty* nil))
  (print '(foo ,@bar ,baz)) nil)
  => '(FOO ,@BAR ,BAZ) NIL

**print-array***

Variable

A boolean which controls whether the contents of arrays other than strings are printed. If the value of **print-array** is nil, the array’s structure name is printed in a concise form, such as #<ART-Q-4-2 270017201>, that identifies the array and gives the dimensions. If the value is t, non-string arrays are printed using #(), #*, or #nA syntax.

This variable replaces si:prinarray, which is obsolete.

(let ((*print-array* t)
      (foo (vector 1 2 3 4 5)))
  (print foo)
  (setq *print-array* nil)
  (print foo)
  nil)

prints:
#(1 2 3 4 5)
#<ART-Q-5 104311373>

**print-array-length***

Variable

Controls the number of objects in the array that will be printed. Its value can be either nil (the default), or any positive integer up to \(2^{31}-1\).

The entire array prints if

- The value of **print-array-length** is nil
- The value of **print-array-length** is equal to or greater than the length of the array to be printed
This variable is dependent on the value of the variable *print-array*. If the value of *print-array* is nil, the array's structure name (which includes the array's length) is printed, no matter what the value of *print-array-length* is. The array's structure name is also printed when the array is longer than the integer value of *print-array-length*.

Examples:

```lisp
(setq array (make-array '(4 2) :initial-contents
  '((a b)
    (1 2)
    ("foo" "bar")
    (#\a #\b)))
=> #2A((A B) (1 2) ("foo" "bar") (#\a #\b))

(let ((*print-array-length* nil))
  (print array) nil)
=> #2A((A B) (1 2) ("foo" "bar") (#\a #\b)) NIL

(let ((*print-array-length* 2))
  (print array) nil)
=> #<ART-Q-4-2 10004306> NIL

(let ((*print-array-length* 8))
  (print array) nil)
=> #2A((A B) (1 2) ("foo" "bar") (#\a #\b)) NIL
```

*print-base*  

Variable

The value of this variable determines the radix in which the printer prints rational numbers (integers and ratios).

*print-base* can have any integer value from 2 to 36, inclusive; its default value is 10 (decimal radix). For values above 10, letters of the alphabet are used to represent digits above 9.

If no radix specifier is set (see *print-radix*), integers in base ten are printed without a trailing decimal point.

If the value of *print-base* is a symbol that has a si:princ-function property (such as :roman or :english), the value of the property is applied to two arguments:

- of the number to be printed
- the stream to which to print it

This allows output in roman numerals and the like.

Examples:
(setq *print-base* ':roman)
(* 5 5) ==> XXV

(setq *print-base* ':english)
(* 5 5) ==> twenty-five

(let ((*print-base* 8))
  (print (read-from-string "10"))
nil)

prints: 12

*print-bit-vector-length*  
Variable

Controls the number of objects in the bit vector that will be printed. Its value can be either nil (the default), or any positive integer up to \(2^{31}-1\).

When the value of *print-bit-vector-length* is nil, *print-bit-vector-length* interacts with *print-array*. Here is a table that shows the interactions:

<table>
<thead>
<tr>
<th><em>print-bit-vector-length</em></th>
<th><em>print-array</em></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>*</td>
<td>always prints the bit vector</td>
</tr>
<tr>
<td>integer</td>
<td>*</td>
<td>prints the bit vector if the value of</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>print-bit-vector-length</em> is equal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to or greater than the length of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit vector to be printed</td>
</tr>
<tr>
<td>nil</td>
<td>t</td>
<td>always prints the bit vector</td>
</tr>
<tr>
<td>nil</td>
<td>nil</td>
<td>never prints the bit vector</td>
</tr>
</tbody>
</table>

* means that the value of this variable does not affect the result

Examples:

(setq bit-vector (make-array 5 :element-type 'bit :initial-contents '(1 0 0 1 0)) => #*10010

(let ((*print-bit-vector-length* 2)
  (*print-array* t))
  (print bit-vector) nil)
=> #<ART-1B-5 10052423> NIL
\begin{verbatim}
(let ((*print-bit-vector-length* 5)
        (*print-array* t))
  (print bit-vector) nil)
=> #x10010 NIL
\end{verbatim}

**print-case**

Variable

Controls the case in which to print any uppercase characters in the names of symbols when vertical-bar syntax is not used. The `zli:read` function normally converts lowercase characters appearing in symbol names to their corresponding uppercase characters. This means that normally internal print names contain only uppercase letters. However, users might prefer to see output using lowercase or mixed case letters.

Lowercase characters in the internal print name are always printed in lowercase and are preceded by a single escape character or enclosed by multiple escape characters. Uppercase characters in the internal print name are printed in uppercase, lowercase, or in mixed case so as to capitalize words, according to the value of `*print-case*`. The convention for what constitutes a "word" is the same as for the function `string-capitalize`.

The value of `*print-case*` must be one of the keywords `:upcase` (the default), `:downcase`, or `:capitalize`. This variable replaces `si:princase`, which is obsolete.

\begin{verbatim}
(let ((*print-case* :capitalize))
  (print (read-from-string "foo"))
nil)
\end{verbatim}

prints: Foo

**print-circle**

Variable

Controls whether or not the printer tries to detect cycles in the structure to be printed. When the value of `*print-circle*` is `nil` (the default), the printing process proceeds by recursive descent. Attempts to print a circular structure can lead to looping behavior and failure to terminate.

When the value is non-nil, the printer tries to detect cycles in the structure to be printed, and uses `#n=` and `#n#` syntax to indicate the circularities.

\begin{verbatim}
(let* ((*print-circle* t)
       (foo (list 1 2 3))
       (foo (rplacd (cddr foo) foo)))
  (princ foo) nil)
\end{verbatim}

prints: #1=(3 1 2 . #1)
**sys:print-cl-structure** object stream depth

Function

Intended for use in a defstruct :print-function option. It prints the structure object to the specified stream using the standard #S syntax. It enables a print function to respect the variable *print-escape*.

```lisp
(defstruct (foo :print-function
    (lambda (object stream depth)
        (if *print-escape*
            (sys:print-cl-structure object stream depth)
            other-printing-strategy)))
```

For a table of related items: See the section “Functions Related to defstruct Structures”.

**print-escape**

Variable

Controls whether or not the printer outputs escape characters. When the value of *print-escape* is nil, escape characters are not output when an expression is printed. In particular, a symbol is printed by simply printing the characters of its print name. The function princ effectively binds *print-escape* to nil.

When the value is t (the default), an attempt is made to print an expression in such a way that it can be read again to produce an zl:equal structure. The function prin1 effectively binds *print-escape* to t.

The following example will print foo first with, then without quotation marks.

```lisp
(let ((*print-escape* t))
    (write "foo")
    (terpri)
    (setq *print-escape* nil)
    (write "foo")
    (terpri)
    nil)
```

“foo”

foo

**print-exact-float-value**

Variable

When set to t, prints the exact number represented by a floating-point number, not the rounded version, which is normally printed by the printer. The default is nil.

For information on floating-point numbers: See the section “Floating-Point Numbers”.
**flavor:print-flavor-compile-trace**  
**&key**  
  **flavor**  
  **generic**  
  **newest**  
  **oldest**  
  **newest-first**

Function

Enables you to view information on the compilation of combined methods that have been compiled into the run-time environment. You can supply keywords to filter the output and control the order of the combined methods displayed:

- **flavor**: Argument is a symbol that names a flavor of interest; all compilations of combined methods for that flavor are displayed. If the argument to `flavor` is `nil`, all flavors are displayed.
- **generic**: Argument is a generic function or message of interest; all compilations of combined methods for that generic function are displayed. If the argument to `generic` is `nil`, all generic functions are displayed.
- **newest**: Argument is an integer greater than or equal to 1, or `nil`. If an integer is given, it selects the number of compilations to display, starting from the most recent. If `nil` is given, all compilations are displayed. The order of combined methods displayed depends on the keyword `newest-first`.
- **oldest**: Argument is an integer greater than or equal to 1, or `nil`. If an integer is given, it selects the number of compilations to display, starting from the oldest. If `nil` is given, all compilations are displayed. The order of combined methods displayed depends on the keyword `newest-first`.
- **newest-first**: Argument is either non-`nil` or `nil`. `nil` causes the display to be ordered from oldest compilation to newest. A non-`nil` value causes the order to be from newest to oldest. By default, combined methods are displayed in oldest-first order.

The output of this function is mouse-sensitive. When you position the mouse over the name of a method or flavor, the menu offers several options that enable you to request more information. Pathnames are also mouse-sensitive.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section “Summary of Flavor Functions and Variables”.

**dbg:print-frame-locals**  
**frame local-start**  
**&optional**  
  **(indent 0)**  
  **n-args-and-locals**

Function

Prints the names and values of the local variables of `frame`. `local-start` is the first local slot number to print; the value returned by `dbg:print-function-and-args` is often suitable for this. `indent` is the number of spaces to indent each line; the default is no indentation.

**Caution**: Use this function only within the context of the `dbg:with-erring-frame` macro.
For a table of related items: See the section "Functions for Examining Stack Frames".

**dbg:print-function-and-args** frame &optional show-pc-p show-source-file-p show-local-if-different

Function

Prints the name of the function executing in `frame` and the names and values of its arguments, in the same format as the Debugger uses. If `show-pc-p` is true, the program counter value of the frame, relative to the beginning of the function, is printed in octal. `dbg:print-function-and-args` returns the number of local slots occupied by arguments.

Caution: Use this function only within the context of the `dbg:with-erring-frame` macro.

For a table of related items: See the section "Functions for Examining Stack Frames".

*print-gensym*

Variable

Controls whether the prefix `#:` is printed before symbols that have no home package. The prefix is printed if the value of `*print-gensym*` is non-nil. The initial value is `t`.

When not nil, causes the prefix `#:` to be printed before symbols with no home package.

```lisp
(let ((foo (gensym))
    (*print-gensym* t))
  (print foo)
  (setq *print-gensym* nil)
  (print foo)
  nil)
```

prints:
`#:G8063`

*print-integer-length*

Variable

Controls the printing of bignums. The default is to print every digit, but for very large bignums that can take prohibitively long. Setting `*print-integer-length*` to an integer, `n`, allows you to see the first `n/2` digits and the last `n/2` digits and the magnitude of the number without printing the entire number.

```lisp
(let ((*print-integer-length* 20))(print (expt 10 30))) =>
```
Variable

*print-length*

Controls how many elements at a given level are printed. Its value can be either nil (the default) or any positive integer up to $2^{31}$-1. This variable replaces *zl:prinlength*, which is obsolete.

The entire object prints if

- The value of *print-length* is nil
- The value of *print-length* is equal to or greater than the number of components in any given level of the object

If *print-length* is an integer, it indicates the maximum number of components to be printed. If the object to be printed has components at or greater than the value of *print-level*, then the object’s structure name is printed.

Examples:

```lisp
(setq list '(a b (c (d e f) g))) => (A B (C) (D (E F) G))

(let ([*print-length* nil])
  (print list) nil) => (A B (C) (D (E F) G)) NIL

(let ([*print-length* 2])
  (print list) nil) => (A B ...) NIL

(let ([*print-length* 4])
  (print list) nil) => (A B (C) (D (E F) G)) NIL

(let ((a '(1 (+ (+ 0 1) 2 3 4) 2 3 4 5 6 7 8))
  (*print-length* 0))
  (print a)
  (setq *print-length* 2)
  (print a)
  (setq *print-length* nil)
  (print a)
  nil)

prints:
(1 ...)
(1 (+ (+ 0 1) ...) ...)
(1 (+ (+ 0 1) 2 3 4) 2 3 4 5 6 7 8)
```
*print-level*

Variable

Controls how many levels of a nested data object will be printed. Its value can be either `nil` (the default), or any positive integer up to $2^{31}-1$. This variable replaces `zl:prinlevel`, which is obsolete.

The entire object prints if

- The value of *print-level* is `nil`
- The value of *print-level* is equal to or greater than the number of levels in the object

If *print-level* is an integer, it indicates the maximum level to be printed. The object itself is level 0; its components (as for a list or vector) are level 1; and so on. If any part the object to be printed has components at or greater than the value of *print-level*, that part of the object is printed as simply #.

Examples:

```lisp
(setq list '(a (b c) (d (e f) g))) => (A (B C) (D (E F) G))

(let ((*print-level* nil))
  (print list) nil) =>
(A (B C) (D (E F) G)) NIL

(let ((*print-level* 2))
  (print list) nil) =>
(A (B C) (D # G)) NIL

(let ((*print-level* 3))
  (print list) nil) =>
(A (B C) (D (E F) G)) NIL

(let ((a '(setq *print-level* (+ (+ 0 1) 2)))
  (*print-level* 0))
  (print a)
  (setq *print-level* 1)
  (print a)
  (setq *print-level* nil)
  (print a)
  nil)

prints:

#

(SETQ *PRINT-LEVEL* #)
(SETQ *PRINT-LEVEL* (+ (+ 0 1) 2))
format:print-list  destination  element-format-string  list  &optional  (separator-format-string  ",  ")  (start-line-format-string  "  ")  (tilde-brace-options  "")

Function

Provides a simpler interface for the specific purpose of printing comma-separated lists where no element from the list is broken at the end of a line.

The destination argument tells where to send the output, as with format; it can be t, nil, a string suitable for string-nconc, or a stream. See the function string-nconc.

element-format-string is a format control string that specifies how to print each element of list. It is used as the body of an iteration construction (as in "{element-format-string"}). See the section "{str}".

separator-format-string, which defaults to "," (comma, space), is a string that is placed after each element, except the last. format control directives are allowed in this string but should not take arguments from list.

start-line, which defaults to three spaces, is a format control string that is used as a prefix at the beginning of each line of output, except the first.

tilde-brace-options is a string inserted before the opening brace (\{) of the iteration construct. It defaults to the null string but allows you to insert a colon or at-sign. The line width of the stream is computed in the same way as with the "{str}" format directive. It is not possible to override the natural line width of the stream.

si:print-list  list  prindepth  slashify-p  stream  which-operations

Function

The part of the Lisp printer that prints lists. A stream's :print handler can call this function, passing along its own arguments and its own which-operations, to arrange for a list to be printed the normal way and the stream's :print hook to get a chance at each of the list's elements.

zl:print-notifications  &optional  (from  0)  (to  (1- (zl:length tv:notification-history)))

Function

Reprints any notifications that have been received. The difference between notifications and sends is that sends come from other users, while notifications are asynchronous messages from Genera itself. If from or to is specified, prints only part of the notifications list.

Example: (zl:print-notifications 0 4) prints the five most recent notifications.

This is the same as the "Show Notifications Command".

clos:print-object  object  stream

Generic Function

Provides a mechanism for users to control the printed representation of instances of a class. clos:print-object is called by the print system and should not be called by users.
clos:print-object returns the object.

object Any Lisp object.
stream A stream (this cannot be t or nil).

The default method uses the #<...> syntax.

Methods on clos:print-object must obey the print control special variables as follows:

- Each method must implement *print-escape* and *print-readably*.
- The *print-pretty* and *print-abbreviate-quote* control variables can be ignored by most methods other than the one for lists.
- The *print-circle* and *print-pretty-printer* control variables are handled by the printer and can be ignored by methods.
- Each method for clos:print-object is expected to handle exactly one level of structure, and should call write (or an equivalent function) recursively to handle any more structural levels. If this rule is followed, then the printer takes care of *print-level* automatically.
- Methods that produce output of indefinite length must obey *print-length*, but most methods other than the one for lists can ignore it.
- The following control variables apply to specific types of objects and are handled by the methods for those objects: *print-array*, *print-array-length*, *print-base*, *print-bit-vector-length*, *print-case*, *print-exact-float-value*, *print-gensym*, *print-integer-length*, *print-radix*, *print-string-length*, and *print-structure-contents*

The stream argument passed to clos:print-object is not necessarily the same as the original stream (it might be an intermediate stream that implements part of the printer). Therefore, methods for clos:print-object should not depend on the identity of the stream.

si:print-object object prindepth slashify-p stream &optional which-operations

Function

Outputs the printed representation of object to stream, as modified by prindepth and slashify-p. This is the guts of the Lisp printer. When a stream's :print handler calls this function, it should supply the list (:string-out) for which-operations, to prevent itself from being called recursively. It can supply nil if it does not want to receive :string-out messages.

Advising this function is the way to customize the behavior of all printing of Lisp objects. See the special form advise.
**Variables**

*print-pretty*

Controls the amount of whitespace output when printing an expression. When the value of *print-pretty* is **nil**, only a small amount of whitespace is output. When the value is non-nil, the output is adjusted to be more readable. Common Lisp uses only the values `t` and **nil**. Symbolics has added the values :code, :data, :plist and :alist.

The permissible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Disables pretty printing</td>
</tr>
<tr>
<td>t</td>
<td>Prints in the default format (the default is :code)</td>
</tr>
<tr>
<td>:code</td>
<td>Prints lists as if they were Lisp code (SCL extension)</td>
</tr>
<tr>
<td>:data</td>
<td>Prints lists with a format based on the first element (SCL extension)</td>
</tr>
<tr>
<td>:plist</td>
<td>Prints lists as property lists, with two elements per line (SCL extension)</td>
</tr>
<tr>
<td>:alist</td>
<td>Prints lists as association lists, giving a dotted cdr for each sublist, even when there is a proper list (SCL extension)</td>
</tr>
</tbody>
</table>

Examples:

```lisp
(write '(defun defvar defparameter defflavor) :pretty t)
=> (DEFUN DEFVAR DEFPARAMETER DEFFLAVOR)

(write '(defun defvar defparameter defflavor) :pretty :data)
=> (DEFUN DEFVAR DEFPARAMETER DEFFLAVOR)

(write '(((defun function)
    (defvar variable)
    (defflavor flavor))
  :pretty t)
=> '((DEFUN FUNCTION) (DEFVAR VARIABLE) (DEFFLAVOR FLAVOR))

(write '(((defun function)
    (defvar variable)
    (defflavor flavor))
  :pretty :alist)
=> '((DEFUN . (FUNCTION))
    (DEFVAR . (VARIABLE))
    (DEFFLAVOR . (FLAVOR)))
```

*print-pretty-printer*

Variable
Allows wholesale replacement of the pretty printer used by Common Lisp. Its value is a function, which will be called with three arguments: the object to be printed, the value of \texttt{*pretty-printer*}, and the output stream. The default is the Common Lisp pretty printer.

\texttt{*print-radix*}

\textit{Variable}

If set to \texttt{t}, rational numbers are printed with a radix specifier indicating what radix the printer is using. (The current radix is controlled by the value of variable \texttt{*print-base*}).

The default value of \texttt{*print-radix*} is \texttt{nil}.

The radix specifier has the general format 

\texttt{#nnrdddd}

where \texttt{n} is an unsigned decimal integer in the range 2 - 36 (inclusive) representing the radix, and \texttt{dddd} denotes the number in radix \texttt{n}.

When the value of \texttt{*print-base*} is 2, 8, or 16 (that is, binary, octal, or hexadecimal) the radix specifier is printed in the abbreviated form, \texttt{#b}, \texttt{#o}, \texttt{#x}, using lower case letters.

For printing integers, base ten is indicated by a trailing decimal instead of a leading radix specifier; for ratios, however, the specifier \texttt{#10r} is printed.

For example, the number ten (10) in radix eleven is 

\texttt{#11rA}

where the 11 indicates the radix, and the A indicates the digit whose base ten equivalent is the number 10. The lower case letters \texttt{#b}, \texttt{#o}, \texttt{#x} may be used as the radix specifier for a \texttt{*print-base*} of 2, 8, or 16. For example,

\texttt{#o10 = 8}

where integer radix 10 is indicated by the decimal point.

\begin{verbatim}
(let ((*print-base* 8)
     (*print-radix* t))
  (print (read-from-string "10"))
(nil))

d Prints: #o12
\end{verbatim}

\texttt{*print-readable*}

\textit{Variable}

A boolean that signals an error if the object to be printed is not in a form that the reader will accept. This is useful for objects such as arrays and flavor instances that are not forms the reader accepts.

\begin{verbatim}
(defflavor food () () => FOOD
\end{verbatim}
(setq apple (make-instance 'food)) => #<FOOD 10074402>

(let ((*print-readable* nil)) (print apple) nil) =>
#<FOOD 10074402> NIL

(let ((*print-readable* t)) (print apple) nil)

Rebinding the following specials:
use Show Standard Value Warnings for details:
*PRINT-PRETTY*, *PRINT-READABLY*, and GPRINT:*INSPECTING*

Error: Can’t print #<FOOD 10074402> readably

SYS:PRINT-NOT-READABLE:
  Arg 0 (SI:OBJECT): #<FOOD 43123626>
  s-A, :  Proceed without any special action
  s-B, :  Return to Lisp Top Level in Dynamic Lisp Listener 1
  → Abort Abort

Return to Lisp Top Level in Dynamic Lisp Listener 1
Back to Lisp Top Level in Dynamic Lisp Listener 1.

**si:print-readable**

A boolean that signals an error if the object to be printed is not in a form that
the reader will accept. The *print-readable* variable is preferred; it is the mod-
ern equivalent of this variable.

When **si:print-readable** is bound to t, the printer signals an error if there is an
attempt to print an object that cannot be interpreted by **zl:read**. When the printer
sends a :print-self or a print message, it assumes that this error checking is done
for it. Thus it is possible for these messages not to signal an error, if they see fit.

**sys:print-self object stream print-depth slashify-p**

The object should output its printed representation to the stream. **print-depth** is the
current depth in list-structure (for comparison with *print-level*). **slashify-p** indicates whether slashification is enabled (**print1** versus **princ**). The printer calls this
generic function when it encounters an instance.

The **sys:print-self** method of **flavor:vanilla** ignores the last two arguments and
prints something like #<**flavor-name** **octal-address**>. The **flavor-name** tells you the
type of object, and **octal-address** lets you tell different objects apart (provided the
garbage collector does not move them). For example:

#<CELL 1160762135>

The vast majority of objects that define **sys:print-self** methods have much in com-
mon. A macro is provided for convenience, so that users do not have to write out
that repetitious code: See the macro **sys:printing-random-object**.
The compatible message for **sys:print-self** is **:print-self**.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**:*print-string-length* Variable**

Controls the number of string characters that will print. Its value can be either **nil** (the default), or any positive integer up to \(2^{31} - 1\).

The entire string prints if

- The value of **:*print-string-length* is nil**
- The value of **:*print-string-length* is equal to or greater than the length of the string to be printed**

Only the structure name (which includes the string’s length) is printed when the string is longer than the integer value of **:*print-string-length*.

Examples:

```lisp
(let ((*print-string-length* nil))
  (print "This is a very long string") nil)
=> "This is a very long string" NIL

(let ((*print-string-length* 4))
  (print "This is a very long string") nil)
=> #<ART-STRING-26 36450275> NIL

(let ((*print-string-length* 4))
  (print "chip") nil) => "chip" NIL
```

**:*print-structure-contents* Variable**

Controls how structures are printed. The default is **t**, which uses the **#S** convention, printing the structure with all its slots filled in.

For example:

```lisp
(defstruct (cat :name 'Endor
                :age 17
                :sex nil
                :color 'black))
```
#S(CAT :NAME ENDOR
 :AGE 17
 :SEX NIL
 :COLOR BLACK)

If *print-structure-contents* is set to nil, the structure just prints as using the #< representation. For example:

#<CAT 40124014>

sys:printing-random-object (object stream &rest either of: :no-pointer or :typep)
&body body (object stream . keywords) body...

Macro

The vast majority of objects that define sys:print-self methods have much in common. See the generic function sys:print-self.

This macro is provided for convenience, so that users do not have to write out that repetitious code. It is also the preferred interface to *print-readably*. With no keywords, sys:printing-random-object checks the value of *print-readably* and signals an error if it is not nil. It then prints a number sign and a less-than sign, evaluates the forms in body, then prints a space, the octal machine address of the object, and a greater-than sign. A typical use of this macro might look like:

(sys:printing-random-object (ship stream)
 (princ (typep ship) stream)
 (tyo #\space stream)
 (prin1 (ship-name ship) stream))

This might print #<ship "ralph" 23655126>.

The following keywords can be used to modify the behavior of sys:printing-random-object:

:no-pointer This suppresses printing of the octal address of the object.
:typep This prints the result of (typep object) after the less-than sign.
In the example above, this option could have been used instead of the first two forms in the body.

sys:proceed condition proceed-type &rest args

Generic Function

Causes a program to continue execution after an error condition has been signalled.

To proceed from a condition, a handler function calls the sys:proceed generic function with one or more arguments. The first argument is the condition object. The second argument is the proceed type, and any remaining arguments are the arguments for that proceed type.
The condition flavor defined by the program signalling the error defines the proceed types that are available to `sys:proceed` for a particular condition. You can also define a method that creates a new proceed type.

The way to define a method that creates a new proceed type is somewhat unusual in that it uses a style of method combination called `:case` combination. Here's an example from the system:

```lisp
(defun (sys:proceed sys:subscript-out-of-bounds :new-subscript)
  (&optional (sub (prompt-and-read :number
     "Subscript to use instead: "))))
  "Supply a different subscript."
  (values :new-subscript sub))
```

This code fragment creates a proceed type called `:new-subscript` for the condition flavor `sys:subscript-out-of-bounds`. New proceed types are always defined by adding a `sys:proceed` method to the condition flavor, which is defined (in the `defflavor` for `condition`) to be combined using the `:case` method combination. The method must always return values rather than throwing.

In `:case` method combination, the first argument to the `sys:proceed` function is like a subsidiary message name, causing a further dispatch just as the original message name caused a primary dispatch. The method from the example is invoked whenever you call the `sys:proceed` generic function with a condition object:

```lisp
(sys:proceed obj :new-subscript new-sub)
```

The variables in the lambda list for the method come from the rest of the arguments of the `send`.

All of the arguments to a `sys:proceed` method must be optional arguments. The `sys:proceed` method should provide default values for all its arguments. One useful way of doing this is to prompt a user for the arguments using the `*query-io*` stream. The example uses `prompt-and-read`. If all the optional arguments were supplied, the `sys:proceed` method must not do any input or output using `*query-io*`.

This facility has been defined assuming that `condition-bind` handlers would supply all the arguments for the method themselves. The Debugger runs this method and does not supply arguments, relying on the method to prompt the user for the arguments.

As in the example, the method should have a documentation string as the first form in its body. The `dbg:document-proceed-type` generic function to a proceedable condition object displays the string. This string is used by the Debugger as a prompt to describe the proceed type. For example, the subscript example might result in the following Debugger prompt:

```
s-A: Supply a different subscript
```

The string should be phrased as a one-line description of the effects of proceeding from the condition. It should not have any leading or trailing newlines. (You can use the messages that the Debugger prints out to describe the effects of the `s-` commands as models if you are interested in stylistic consistency.)
Sometimes a simple fixed string is not adequate. You can provide a piece of Lisp code to compute the documentation text at run time by providing your own method for **sys:document-proceed-type**. This method definition takes the following form:

```
(defmethod (dbg:document-proceed-type condition-flavor proceed-type)
  (stream)
  body...)  
```

The body of the method should print documentation for `proceed-type` of `condition-flavor` onto `stream`.

The body of the **sys:proceed** method can do anything it wants. In general, it tries to repair the state of things so that execution can proceed past the point at which the condition was signalled. It can have side-effects on the state of the environment, it can return values so that the function that called **signal** can try to fix things up, or it can do both. Its operation is invisible to the handler; the signaller is free to divide the work between the function that calls **signal** and the **sys:proceed** method as it sees fit. When the **sys:proceed** method returns, **signal** returns all of those values to its caller. That caller can examine them and take action accordingly.

The meaning of these returned values is strictly a matter of convention between the **sys:proceed** method and the function calling **signal**. It is completely internal to the signaller and invisible to the handler. By convention, the first value is often the name of a proceed type. See the section "Signallers".

A **sys:proceed** method can return a first value of **nil** if it declines to proceed from the condition. If a **nil** returned by a **sys:proceed** method becomes the return value for a **condition-bind** handler, this signifies that the handler has declined to handle the condition, and the condition continues to be signalled. When the **sys:proceed** function is called by the Debugger, the Debugger prints a message saying that the condition was not proceeded, and it returns to its command level. This might be used by an interactive **sys:proceed** method that gives the user the opportunity either to proceed or to abort; if the user aborts, the method returns **nil**. Returning **nil** from a **sys:proceed** method should not be used as a substitute for detecting earlier (such as when the condition object is created) that the proceed type is inappropriate for that condition.

Condition objects created with **error** instead of **signal** do not have any proceed types.

See the section "Proceeding".

The compatible message for **sys:proceed** is:

```
:proceed
```

**dbg:proceed-type-p** condition proceed-type

*Generic Function*

Returns **t** if `proceed-type` is one of the valid proceed types of this condition object. Otherwise, returns **nil**.

The compatible message for **dbg:proceed-type-p** is:
:proceed-type-p

For a table of related items, see the section "Basic Condition Methods and Init Options".

**dbg:proceed-types**  condition  

*Generic Function*

Returns a list of all the valid proceed types for this condition.

The compatible message for **dbg:proceed-types** is:

:proceed-types

For a table of related items, see the section "Basic Condition Methods and Init Options".

**(flavor:method :proceed-types condition)**  

*Init Option*

Defines the set of proceed types to be handled by this instance. *proceed-types* is a list of proceed types (symbols); it must be a subset of the set of proceed types understood by this flavor. If this option is omitted, the instance is able to handle all of the proceed types understood by this flavor in general, but by passing this option explicitly, a subset of acceptable proceed types can be established. This is used by **signal-proceed-case**.

If only one way to proceed exists, *proceed-types* can be a single symbol instead of a list.

If you pass a symbol that is not an understood proceed type, it is ignored. It does not signal an error because the proceed type might become understood later when a new **defmethod** is evaluated; if not, the problem is caught later.

The order in which the proceed types occur in the list controls the order in which the Debugger displays them in its list. Sometimes you might want to select an order that makes more sense for the user, although usually this is not important. The most important thing is that the **RESUME** command in the Debugger is assigned to the first proceed type in the list.

For a table of related items, see the section "Basic Condition Methods and Init Options".

**dbg:*proceed-type-special-keys***  

*Variable*

The value should be an alist associating proceed types with characters. When an error supplies any of these proceed types, the Debugger assigns that proceed type to the specified key. For example, this is the mechanism by which the **:store-new-value** proceed type is offered on the s-sh-c keystroke.

For a table of related items, see the section "Debugger Special Key Variables".

**proclaim**  declaration  

*Function*
Puts the declaration specifier `declaration` into effect globally.

Each `declaration` is a list whose `car` is a symbol that indicates the kind of declaration and whose `cdr` is a list of objects to which the declaration applies.

```
(proclaim 'inline my-function)
```

Declarations made with `proclaim` are referred to as `proclamations` and are always global. Any variable mentioned in a proclamation refers to the dynamic binding of the variable and any function mentioned refers to its global function definition. A proclamation is always in force unless overridden locally. See the macro `locally`.

In addition to the declaration specifiers used with `declare`, the declaration specifier `declaration` can also be used with `proclaim`. The `declaration` declaration specifier is a list of the symbol `declaration` and one or more declaration specifier symbols. Any declarations that are not standard Common Lisp declarations must be listed in a proclamation of `declaration`. If any declarations are not recognized by the compiler and not so listed are encountered, an error will be signaled.

See the section "Operators for Making Declarations".

```
prog vars-and-vals &body body
```

`prog` provides temporary variables, sequential evaluation of forms, and a "goto" facility. A typical `prog` looks like:

```
(prog (var1 var2 (var3 init3) var4 (var5 init5))
tag1
  statement1
  statement2
tag2
  statement3
  
)
```

The first subform of a `prog` is a list of variables, each of which can optionally have an initialization form. The first thing evaluation of a `prog` form does is to evaluate all of the `init` forms. Then each variable that had an `init` form is bound to its value, and the variables that did not have an `init` form are bound to `nil`. Example:

```
(prog ((a t) b (c 5) (d (car '(zz . pp))))
  <body>
)
```

The initial value of `a` is `t`, that of `b` is `nil`, that of `c` is the integer 5, and that of `d` is the symbol `zz`. The binding and initialization of the variables is done in parallel; that is, all the initial values are computed before any of the variables are changed. `prog*` is the same as `prog` except that this initialization is sequential rather than parallel.

The part of a `prog` after the variable list is called the `body`. Each element of the body is either a symbol or an integer, in which case it is called a `tag`, or anything else (almost always a list), in which case it is called a `statement`. 
After **prog** binds the variables, it processes each form in its body sequentially. Anything that is a **tag** is skipped over. **statements** are evaluated, and their returned values discarded. If the end of the body is reached, the **prog** returns **nil**. However, two special forms can be used in **prog** bodies to alter the flow of control. If (**return** x) is evaluated, **prog** stops processing its body, evaluates x, and returns the result. If (**go tag**) is evaluated, **prog** jumps to the part of the body labelled with the **tag**, where processing of the body is continued. **tag** is not evaluated.

The compiler requires that **go** and **return** forms be *lexically* within the scope of the **prog**; it is not possible for a function called from inside a **prog** body to **return** to the **prog**. That is, the **return** or **go** must be inside the **prog** itself, not inside a function called by the **prog**.

See the special form **do**. That uses a body similar to **prog**. The **do**, **catch**, and **throw** special forms are included as an attempt to encourage goto-less programming style, which often leads to more readable, more easily maintained code. You should use these forms instead of **prog** wherever reasonable. Moreover, since **prog** is a combination of **block**, **tagbody**, and **let**, it is often better to use these constructs as needed. This is especially true in the case of macros with bodies where the unintended inclusion of a **block** might overshadow the user’s use of **block**.

If the first subform of a **prog** is a non-**nil** symbol (rather than a variable list), it is the name of the **prog**, and **return-from** can be used to return from it. In Zetalisp, see the special form **zl:do-named**.

Examples:

```lisp
(defun t-test (choice)
  (prog classic (pep coca) ; Initialize pep, coca to nil.
     (if (equal choice "left") (go left) )
     right
     (princ "pep is it")
     (terpri)
     (return t)
     left
     (princ "coca is it")
     (terpri)
     (return))) => T-TEST
(t-test "left")  => coca is it NIL
(t-test "right") => pep is it T
```

(prog (x y z) ; x, y, z are prog variables - temporaries.
    (setq y (car w) z (cdr w)) ; w is a free variable.
  loop
    (cond ((null y) (return x))
          ((null z) (go err)))
  rejoin
    (setq x (cons (cons (car y) (car z)) x))
    (setq y (cdr y) z (cdr z))
    (go loop)
  err
    (break "are-you-sure?")
    (setq z y)
    (go rejoin))

(defun factorial (x)
  "uses prog to implement an iterative factorial"
  (prog (i n)
        (if (minusp x) (error "Negative argument ~D to FACTORIAL" x))
        (setq n 1 i x)
      lp
        (if (zerop i) (return n))
        (setq n (* n i) i (- i 1))
        (go lp)))

prog, do, and their variants are effectively constructed out of let, block, and tagbody forms. prog could have been defined as the following macro (except for processing of local declare, which has been omitted for clarity):

(defun prog (&rest x)
  (let ((block-name (and (symbolp (car x))
                          (neq (car x) nil)
                          (pop x)))
        (variables (car x))
        (tagbody (cdr x)))
    (if block-name
        '(block ,block-name
          (block nil
            (let ,variables
              (tagbody ,@tagbody)))
        'block nil
            (let ,variables
              (tagbody ,@tagbody)))))

For a table of related items, see the section "Iteration Functions".

prog* vars-and-vals &body body

Special Form
The same as \texttt{prog}, except that the binding and initialization of the temporary variables is done \textit{sequentially}, so each one can depend on the previous ones.

For example:

\begin{verbatim}
(prog* ((y z) (x (car y)))
   (return x))
\end{verbatim}

returns the car of the value of \texttt{z}.

Examples:

\begin{verbatim}
(prog ( (x 1) (y (+ x 1)) z)
   (princ x)(princ " ")
   (princ y)(princ " ")
   (princ z)(princ " ")
   (terpri)) => Error: The variable X is unbound.
\end{verbatim}

\begin{verbatim}
(prog* ( (x 1) (y (+ x 1)) z)
   (princ x)(princ " ")
   (princ y)(princ " ")
   (princ z)(princ " ")
   (terpri)) => 1 2 NIL
\end{verbatim}

\texttt{prog*} is a synthesis of \texttt{let*}, \texttt{block} and \texttt{tagbody}. The \texttt{tagbody}, which is the body of \texttt{tags} and \texttt{statements}, is executed in the context of the variable bindings specified in the initial list argument to \texttt{prog*}. The specified bindings are computed sequentially, and the new bindings are in effect when computing values to the right in the binding list. This macro has an implicit block name of \texttt{nil}, and can be exited by \texttt{return} or \texttt{return-from}.

The implicit \texttt{tagbody} allows the successive evaluation of a number of forms in a context that permits the use of a \texttt{go} statement. Elements of the \texttt{tagbody} may be either \texttt{tags}, which are integers or symbols having lexical scope and dynamic extent, or they may be \texttt{statements}, which are lists. The \texttt{tags} are ignored except as targets of the (\texttt{go tag}) \texttt{statement}, which transfers control to the first list following the \texttt{tag}. The lists are evaluated, and \texttt{prog*} returns \texttt{nil}.

\begin{verbatim}
(prog* ((i 5)
   (list (reverse *data-list*))
   (item (car list)))
loop
   (when (or (endp list)(>= i (length *data-vector*)))
      (return t))
   (unless (= (aref *data-vector* i) item)
      (return nil))
   (setq i (+ i 1))
   (setq list (cdr list))
   (setq item (car list))
   (go loop))
\end{verbatim}

For a table of related items, see the section "Iteration Functions".
**prog1** *value &rest ignore*  
*Special Form*

Similar to **progn**, but returns *value* (its *first* form) rather than its last. It is most commonly used to evaluate an expression with side effects, and return a value that must be computed *before* the side effects happen.

Example:

```lisp
(setq x (prog1 y (setq y x)))
```

interchanges the values of the variables *x* and *y*.

Example:

```lisp
(setf (aref array index) 5)  
(prog1  
  (aref array index)  
  (incf (aref array index) 2)) => 5
```

```lisp
(aref array index) => 7
```

**prog1** never returns multiple values. See the special form **multiple-value-prog1**. See the section "Special Forms for Sequencing".

CLOE Note: **prog1** is a macro in CLOE.

**prog2** *ignore value &rest ignore*  
*Special Form*

Similar to **progn** and **prog1**, but returns its *second* form. It is included largely for compatibility with old programs. See the section "Special Forms for Sequencing".

Example:

In the following code, message printing brackets the second form.

```lisp
(prog2  
  (print prompt)  
  (read-and-evaluate)  
  (print pause-message))
```

CLOE Note: **prog2** is a macro in CLOE.

**progn** *&body body*  
*Special Form*

Evaluates the *body* forms in order from left to right and returns the value of the last one. **progn** is the primitive control structure construct for "compound statements". Although lambda-expressions, **cond** forms, **do** forms, and many other control structure forms use **progn** implicitly, that is, they allow multiple forms in their bodies, there are occasions when you need to evaluate a number of forms for their side effects and make them appear to be a single form. Example:
Note that in some cases where only a single form is allowed, semantically equivalent alternate forms accepting multiple forms are also available.

For example, the `if` form can be replaced by `cond`; thus, allowing multiple forms in each branch. The `cond` form, however, still allows only one form in the test part. The protected form in an `unwind-protect` and the init forms of a `do` or `let` are other examples of single forms lacking alternate multiple forms.

Here is another example involving `unwind-protect`.

See the section "Special Forms for Sequencing".

**progv**  
Progv provides the user with extra control over binding. It binds a list of special variables to a list of values, and then evaluates some forms. The lists of special variables and values are computed quantities; this is what makes `progv` different from `let`, `prog`, and `do`.

`progv` first evaluates `vars` and `vals`, and then binds each symbol to the corresponding value. If too few values are supplied, the remaining symbols are bound to `nil`. If too many values are supplied, the excess values are ignored.

After the symbols have been bound to the values, the `body` forms are evaluated, and finally the symbols' bindings are undone. The result returned is the value of the last form in the body. Example:

```
(setq a 'foo b 'bar)
```
During the evaluation of the body of this `progv`, `foo` is bound to `bar`, `bar` is bound to `nil`, `b` is bound to `nil`, and `a` retains its top-level value `foo`.

(setq win-list '(*current-window* *cursor-pos*))
(progv win-list
  (list (make-window)(cursor-reset)))
(initialize-pop-up *current-window*)
(process-user-input))

See the special form `progw`.

For other related functions, see the section "Special Forms for Sequencing".

---

### progw vars-and-vals &body body

A somewhat modified version of `progv`; like `progv`, it only works for special variables. First, `vars-and-vals-form` is evaluated. Its value should be a list that looks like the first subform of a `let*`:

```
  ((var1 val-form-1)
   (var2 val-form-2)
   ...
  )
```

Each element of this list is processed in turn, by evaluating the `val-form`, and binding the `var` to the resulting value. Finally, the `body` forms are evaluated sequentially, the bindings are undone, and the result of the last form is returned. Note that the bindings are sequential, not parallel.

This is a very unusual special form because of the way the evaluator is called on the result of an evaluation. Thus, `progw` is mainly useful for implementing special forms and for functions part of whose contract is that they call the interpreter. For an example of the latter, see `sys:*break-bindings*`; `break` implements this by using `progw`.

See the special form `progv`.

For other related functions, see the section "Special Forms for Sequencing".

---

### :properties

Returns two values:

- A list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a "disembodied" property list and `zl:get` can be used to access the file’s properties.
- A list of what properties of this file are "changeable".
**sys:property-cell-location** symbol

*Function*

Returns a locative pointer to the location of *sym*'s property-list cell. This locative pointer is as valid as *sym* itself as a handle on *sym*'s property list.

See the section "Functions Relating to the Property List of a Symbol".

**sys:property-list-mixin**

*Flavor*

Provides methods that perform the generic functions on property lists. **sys:property-list-mixin** provides methods for the following generic functions:

**:get indicator**

*Message*

Looks up the object's *indicator* property. If it finds such a property, it returns the value; otherwise it returns **nil**.

**:getl indicator-list**

*Message*

Like the **:get** message, except that the argument is a list of indicators. The **:getl** message searches down the property list for any of the indicators in *indicator-list* until it finds a property whose indicator is one of those elements. It returns the portion of the property list beginning with the first such property that it found. If it does not find any, it returns **nil**.

**:putprop property indicator**

*Message*

Gives the object an *indicator*-property of *property*.

**:remprop indicator**

*Message*

Removes the object's *indicator* property by splicing it out of the property list. It returns that portion of the list inside the object of which the former *indicator*-property was the **car**.

**:push-property value indicator**

*Message*

The *indicator*-property of the object should be a list (note that **nil** is a list and an absent property is **nil**). This message sets the *indicator*-property of the object to a list whose **car** is *value* and whose **cdr** is the former *indicator*-property of the list. Executing the form

```
(send object :push-property value indicator)
```

is analogous to doing

```
(zl:push value (send object :get indicator))
```

See the function **zl:push**.
:property-list

Returns the list of alternating indicators and values that implements the property list.

:set-property-list list

Sets the list of alternating indicators and values that implements the property list to list.

(flavor:method :property-list sys:property-list-mixin) list

Initializes the list of alternating indicators and values that implements the property list to list.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section “Summary of Flavor Functions and Variables”.

provide module-name

Adds module-name to the list in *modules* to indicate that the module has been loaded.

In the following code, the call to require loads the turbine-package module, and if turbine-speed were a constant in turbine-package, then its value would be available at this point. The following call to provide adds the name to the special variable *modules*. Generally, the call to provide would be made within the file containing the module to be loaded.

```lisp
0 => *modules*
 (GENERATOR-PACKAGE LISP)
 => (require 'turbine-package)
 TURBINE-PACKAGE
 => turbine-package:turbine-speed
 3600 => (provide 'turbine-package)
 TURBINE-PACKAGE
 => *modules*
 (TURBINE-PACKAGE GENERATOR-PACKAGE LISP)
```

psetf &rest pairs

Similar to setf, but performs all the assignments in parallel, that is, simultaneously, instead of from left to right. A generalization of parallel variable assignment, psetf is to setf what psetq is to setq. Allows the update of a wide variety of storage locations, such as structure components, vector elements, or elements of a list.
The &rest argument indicates that psetf expects 0 or more pairs on which to perform assignment operations. In each pair, a new value is assigned to a place. Evaluations are still performed from left to right, but assignments are parallel. psetf always returns the value nil.

```lisp
(setq a (cons 'foo 'bar))
(FOO . BAR)
(psetf (car a) (cdr a) (cdr a) (car a))
a => (BAR . FOO)
```

A large number of place forms are predefined, and additions can be made via defsetf or define-setf-method. See the macro setf.

**psetq &rest rest**

Macro

Similar tosetq, but performs all the assignments in parallel, that is, simultaneously, instead of from left to right. The &rest argument indicates that psetq expects 0 or more pairs which to perform assignment operations. In the arglist, these pairs are represented by rest. In each pair, a form is assigned to a variable. Evaluations are still performed from left to right, but assignments are parallel. psetq always returns the value nil.

Returns nil, and takes alternating variables and values as arguments. The even arguments are evaluated, and assigned as the value of the preceding variables. Because the evaluations are executed first, followed by the assignments, the assignments are effectively executed in parallel. This function is acceptable for both special and lexical variables.

```lisp
(setq a 3 b 4) => 4
(setq a b b a) => 4
a => 4
b => 4
```

```lisp
(setq a 3 b 4) => 4
(psetq a b b a) => NIL
a => 4
b => 3
```

**zl:psetq &rest rest**

Special Form

Just like a setq form, except that the variables are set "in parallel"; first all the value forms are evaluated, and then the variables are set to the resulting values.

Example:

```
(setq a 1)
(setq b 2)
(psetq a b b a)
a => 2
b => 1
```
**push item reference &key :area :localize**  

*Function*

If the list held in *reference* is viewed as a push-down stack, *push* can be thought of as pushing an element onto the top of the stack. *item* can be any Lisp object. *reference* can be the name of any generalized variable containing a list, that is, any form acceptable as a generalized variable to `setf`. *push* conses *item* onto the front of the list, and the augmented list is stored back into *reference* and returned.

**Compatibility Note:** The optional keyword arguments :area and :localize are Symbolics extensions to Common Lisp, and cannot be used in CLOE.

:area An integer that specifies the area in which to store the augmented list. See the section "Areas".

:localize Can be nil, t, or a positive integer:

- **nil** Does not change the behavior of *push*.
- **t** Localizes the top level of list structure, by calling `sys:localize-list` or `sys:localize-tree` on the list before returning it.
- **integer** Localizes integer levels of list structure, by calling `sys:localize-list` or `sys:localize-tree` on the list before returning it.

Examples:

```
(setq alist '(((A . B) (C . D))) => ((A . B) (C . D))

(push '(1 . 2) (cdr alist)) => ((1 . 2) (C . D))
alist => ((A . B) (1 . 2) (C . D))

(push '(3 . 4) alist :localize 2) =>
((3 . 4) (A . B) (1 . 2) (C . D))
alist => ((3 . 4) (A . B) (1 . 2) (C . D))
```

**Note:** If you try to push an item onto a list that is already a member of that list, with *push*, it adds that item to the list. This in contrast to *pushnew*, which does not add the item to the list. See the function *pushnew*.

```
(setq alist '(((9 . 10) (11 . 12)))

(pushnew '(9 . 10) alist) =>
((9 . 10) (9 . 10) (11 . 12))
alist => ((9 . 10) (9 . 10) (11 . 12))
```

For a table of related items: See the section "Functions for Constructing Lists and Conses".
**zl:push item list**

Function

Adds *item* to the front of a list, which should be stored in a generalized variable. `(zl:push item list)` creates a new cons whose car is the result of evaluating *item* and whose cdr is the contents of *list*, and stores the new cons into *list*.

The form:

```
(zl:push (new-function x y z) variable)
```

replaces the commonly used construct:

```
(setq variable (cons (new-function x y z) variable))
```

and is intended to be more explicit and aesthetic.

The caveat that applies to *incf* also applies to *zl:push*: this function does not evaluate any part of *ref* more than once.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**zl:push-in-area item list area**

Function

Adds *item* to the front of a list, which should be stored in a generalized variable. `(zl:push-in-area item list area)` creates a new cons in *area* whose car is the result of evaluating *item* and whose cdr is the contents of *list*, and stores the new cons into *list*. See the section "Areas".

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**pushnew item reference &key :test :test-not :key :area :localize :replace**

Function

If the list held in *reference* is viewed as a push-down stack, *pushnew* can be thought of as pushing *item* onto the top of the stack, unless it is already a member of the list. *item* can be any Lisp object. *reference* can be the name of any generalized variable containing a list, that is any form acceptable as a generalized variable to *setf*.

*item* is checked for membership in the list, as determined by the :test predicate, which defaults to *eql*. If *item* is not a member of the list, it is consed onto the front of the list, and the augmented list is stored back into *reference* and returned. If *item* is a member of the list, the unaugmented list is returned.

**Compatibility Note:** The optional keyword arguments :area, :localize, and :replace are Symbolics extensions to Common Lisp, and cannot be used in CLOE.

:area An integer that specifies the area in which to store the augmented list. See the section "Areas".

:localize Can be *nil*, *t*, or a positive integer, which specify the following:
Does not change the behavior of pushnew.

t  Localizes the top level of list structure, by calling sys:localize-list or sys:localize-tree on the list before returning it.

integer  Localizes integer levels of list structure, by calling sys:localize-list or sys:localize-tree on the list before returning it.

:replace  Destructively modifies the specified element (or elements) and replaces it with the value provided. :replace's value can be t or nil. For example:

\[
\begin{align*}
\text{(setq } l \ '(((a \ 1) \ (b \ 2) \ (c \ 3))) & \Rightarrow ((A \ 1) \ (B \ 2) \ (C \ 3)) \\
\text{(pushnew } '(a \ 10) \ l \ :key \ 'first \ :replace \ t) & \Rightarrow ((A \ 10) \ (B \ 2) \ (C \ 3))
\end{align*}
\]

Examples:

\[
\begin{align*}
\text{(setq } alist \ '(((a \ . \ b) \ (c \ . \ d))) & \Rightarrow ((A \ . \ B) \ (C \ . \ D)) \\
\text{(pushnew } '(1 \ . \ 2) \ (cdr \ \alist) \ :\text{localize} \ \text{nil}) & \Rightarrow ((1 \ . \ 2) \ (C \ . \ D)) \\
alist & \Rightarrow ((A \ . \ B) \ (1 \ . \ 2) \ (C \ . \ D)) \\
\text{(pushnew } '(C \ . \ D) \ (cdr \ \alist) \ :\text{test} \ '#\text{equal} \ :\text{localize} \ 2) & \Rightarrow ((1 \ . \ 2) \ (C \ . \ D)) \\
alist & \Rightarrow ((A \ . \ B) \ (1 \ . \ 2) \ (C \ . \ D))
\end{align*}
\]

Note: If you use pushnew to try to push an item onto a list that is already a member of that list, it has no effect on the list. This is in contrast to push, which pushes the item on the list. See the function push. For example:

\[
\begin{align*}
\text{(setq } \alist \ '(((5 \ . \ 6) \ (7 \ . \ 8))) & \\
\text{(pushnew } '(5 \ . \ 6) \ \alist \ :\text{test} \ '#\text{equal}) & \\
alist & \Rightarrow (\alist (5 \ . \ 6) (7 \ . \ 8))
\end{align*}
\]

CLOE users: one possible implementation of provide employs pushnew, as demonstrated in the following example.

\[
\begin{align*}
\text{(defun } \provide\text{(module-name)} & \\
& \quad \text{(pushnew (string module-name) *modules* :test '#\text{string=})}
\end{align*}
\]

For a table of related items: See the section "Functions for Constructing Lists and Conses".
:put-hash  key  value

Message

Creates an entry in the hash table associating key to value. If there is an existing entry for key, it replaces the value of that entry with value and returns value. The hash table automatically grows if necessary.

This message is obsolete; use setf in conjunction with the gethash function instead.

zl:puthash  key  value  hash-table

Function

Creates an entry in hash-table associating key to value. If there is an existing entry for key, it replaces the value of that entry with value and return value. hash-table grows automatically if necessary. This function is obsolete; use setf in conjunction with the gethash function instead.

zl:puthash-equal  key  value  hash-table

Function

Creates an entry in hash-table associating key to value. If there is an existing entry for key, it replaces the value of that entry with value and return value. hash-table grows automatically if necessary. This function is obsolete; use setf in conjunction with the gethash function instead.

zl:putprop  sym  value  indicator

Function

Gives sym an indicator-property of value. After this is done, (zl:get symbol indicator) returns value. zl:putprop uses its associated property list. zl:putprop returns its second argument. See the section "Property Lists".

Example:

(zl:putprop 'Nixon 'not 'crook) => NOT

For a table of related items: See the section "Functions That Operate on Property Lists".

*query-io*

Variable

The value is a stream to be used when asking questions of the user. The question should be output to this stream, and the answer read from it. When the normal input to a program comes from a file, questions such as "Do you really want to delete all of the files in your directory?" should be sent directly to the user and the answer should come from the user also, not from the data file. For these purposes, *query-io* should be used instead of *standard-input* and *standard-output*. *query-io* is used by such functions as yes-or-no-p.

In the following example, *standard-input* and *standard-output* are bound to files. Actions with severe consequences were requested, possibly by the input stream from the input file. In order to obtain confirmation and further information from the user, *query-io* is used instead of the file.
(with-open-file (outstream "myfile" :direction :output)
  (with-open-file (instream "infile.txt" :direction :input)
    (let ((*standard-output* outstream)
           (*standard-input* instream))
      ...
      (format *query-io* "You are requesting permanent destruction of data")
      (format *query-io* "Which records should be destroyed? ")
      (build-record-list (read *query-io*))
      ...
    )))

**zl:query-io**

In your new programs, we recommend that you use the variable *query-io*, which is the Common Lisp equivalent of **zl:query-io**.

The value of **zl:query-io** is a stream that should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. The reason for this is that when the normal input to a program might be coming from a file, questions such as "Do you really want to delete all of the files in your directory?" should be sent directly to the user, and the answer should come from the user, not from the data file. **zl:query-io** is used by **fquery** and related functions.

**quote object**

Returns object. It is useful specifically because object is not evaluated; the **quote** is how you make a form that returns an arbitrary Lisp object. **quote** is used to include constants in a form. Examples:

```
(quote x) => x
(setq x (quote (some list)))  x => (some list)
(setq foo (+ 1 2))
  foo => 3
(setq foo (quote (+ 1 2)))
  foo => (+ 1 2)
```

Since **quote** is so useful but somewhat cumbersome to type, the reader normally converts any form preceded by a single quote (') character into a **quote** form. Example:

```
(setq x '(some list))
```

is converted by **read** into

```
(setq x (quote (some list)))
```

See the section "Functions and Special Forms for Constant Values".

**zl:quotient number &rest more-numbers**

Function
Returns the first argument divided by all of the rest of its arguments.

With more than one argument, zl:quotient is the same as zl:/.  

With integer arguments, zl:quotient acts like truncate, except that it returns only a single value, the quotient.

For a table of related items, see the section "Arithmetic Functions".

 random number &optional (state *random-state*)  
 Function  
 Generates numbers from a uniform distribution over [0, number) (meaning the interval including 0, and up to but excluding number.)

 random generates and returns an integer if number is an integer, returns a single-precision floating-point number if number is single-precision, and returns a double-precision number if number is double-precision.

 number must be positive and can either be an integer or a floating-point number. If number is an integer, each of the possible results occurs with probability very close to 1/number.

The optional argument state must be an object of type random-state. It defaults to the current value of the variable *random-state* which is used to maintain the state of the pseudorandom number generator between calls. The value of *random-state* changes as a side effect of the random operation.

For example:

(defun executive-decision-maker (question &optional (choices '(yes no)))  
(declare (ignore question))  
(sleep 5)  
(nth (random (length choices)) choices))

(executive-decision-maker "Should I buy a new car?") => YES

(executive-decision-maker "Where should we eat lunch?"  
 '(deli woven-hose mary-chung)) => MARY-CHUNG

(list (random 25) (random 25) (random 25)) => (16 5 8)

For a table of related items, see the section "Random Number Functions" and see CLtL 228.

 zl:random &optional arg random-array  
 Function  
 Returns a random integer, positive or negative. If arg is present, an integer between 0 and arg minus 1 inclusive is returned. If random-array is present, the given array is used instead of the default one. Otherwise, the default random-array is used (and is created if it does not already exist). The algorithm is executed inside a without-interrupts so two processes can use the same random-array without colliding.
For a table of related items, see the section "Random Number Functions".

**si:random-create-array** length offset seed &optional (area nil)

Function

Creates, initializes, and returns a random-array. length is the length of the array. offset is the distance between the pointers and should be an integer less than length. seed is the initial value of the seed, and should be an integer. This calls **si:random-initialize** on the random array before returning it.

For a table of related items, see the section "Random Number Functions".

**si:random-initialize** array &optional new-seed

Function

Reinitializes the contents of the array from the seed (calling **zl:random** changes the contents of the array and the pointers, but not the seed).

array must be a random-array, such as is created by **si:random-create-array**. If new-seed is provided, it should be an integer, and the seed is set to it.

For a table of related items, see the section "Random Number Functions".

**random-normal** &optional (mean 0.0) (standard-deviation 1.0) (state *random-state*)

Function

Generates random numbers from the normal (Gaussian) distribution with mean mean and standard deviation standard-deviation.

Returns a double-precision floating-point answer if either mean or standard-deviation is double-precision. Otherwise, it returns a single-precision floating-point answer.

The optional argument state must be an object of type **random-state**. It defaults to the current value of the variable *random-state* which is used to maintain the state of the pseudorandom number generator between calls. The value of *random-state* changes as a side effect of the **random-normal** operation.

You can use this function on items that are normally distributed, such as heights and weights. For example, to assign grades for a group of students:

```lisp
(defun assign-grade (student class-average class-standard-deviation)
  (declare (ignore student))
  ;; pick grades from a "bell curve", or normal distribution
  (let ((raw-grade (random-normal class-average class-standard-deviation)))
    ;; make sure the output falls in the range for grades
    (max 0 (min (round raw-grade) 100)))))

(loop for student in (sort ’("Ron" "Dave" "Sue" "Jackie" "Fred" "Mary")
                                             ’#’string-lessp)
      do (format t "~A~6.2D~\n" student (assign-grade student 75 10)))
```
For a table of related items, see the section "Random Number Functions".

*random-state*  
Variable
This variable holds a data structure, an object of type random-state which the function random uses by default to encode the internal state of the random-number generator.

This data structure can be printed and successfully read back in. Each call to random performs a side effect on *random-state*. *random-state* can be lambda-bound to a different random-number state object to save and restore the old state object.

(random-state-p *random-state*) => t

random-state-p object  
Function
This predicate is true if the argument is an object of type random-state; it is false otherwise.

Examples:
(setq x (make-random-state)) => #.(RANDOM-STATE 71 1695406379...)  
(setq copy-x (make-random-state x)) => #.(RANDOM-STATE 71...)  
(random-state-p x) => T  
(random-state-p copy-x) => T  
(random-state-p *random-state*) => T ;always true  
(random-state-p (random 10)) => NIL

For a table of related items, see the section "Random Number Functions".

zl:rass pred item in-list  
Function
Looks up item in the association list in-list. The value is the first cons whose cdr matches item, according to pred or nil if none matches. You can use noncommutative predicates; the first argument to the predicate is pred, the second is item, and the third is the cdr of the element of in-list.

For a table of related items: See the section "Functions that Operate on Association Lists".

rassoc item a-list &key (test #eql) test-not (key #identity)  
Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dave</td>
<td>66</td>
</tr>
<tr>
<td>Fred</td>
<td>88</td>
</tr>
<tr>
<td>Jackie</td>
<td>90</td>
</tr>
<tr>
<td>Mary</td>
<td>85</td>
</tr>
<tr>
<td>Ron</td>
<td>91</td>
</tr>
<tr>
<td>Sue</td>
<td>72</td>
</tr>
</tbody>
</table>
Searches the association list \textit{a-list}. Returns the first pair in \textit{a-list} whose cdr satisfies the predicate specified by \texttt{:test}. Returns \texttt{nil} if none does. \texttt{rassoc} is the reverse form of \texttt{assoc}. The keywords are:

\begin{itemize}
  \item \texttt{:test} \hspace{1cm} Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The \textit{item} matches the specification only if the predicate returns \texttt{t}. If \texttt{:test} is not supplied, the default operation is \texttt{eql}.
  \item \texttt{:test-not} \hspace{1cm} Similar to \texttt{:test}, except that \textit{item} matches the specification only if there is an element of the list for which the predicate returns \texttt{nil}.
  \item \texttt{:key} \hspace{1cm} If not \texttt{nil}, should be a function of one argument that will extract the part to be tested from the whole element.
\end{itemize}

If \textit{a-list} is considered to be a mapping, \texttt{rassoc} treats the \textit{a-list} as representing the inverse mapping. For example:

\begin{verbatim}
(rassoc 'diver '(((eagle . raptor) (loon . diver)))) =>
  (LOON . DIVER)
(rassoc 'loon '(((eagle . raptor) (loon . diver)))) => NIL
\end{verbatim}

The two expressions

\begin{verbatim}
(rassoc item a-list :test pred)
\end{verbatim}

and

\begin{verbatim}
(find item a-list :test pred :key #'cdr)
\end{verbatim}

are almost equivalent in meaning. The difference occurs when \texttt{nil} appears in \textit{a-list} in place of a pair, and the item being searched for is \texttt{nil}. In these cases, \texttt{find} computes the cdr of the \texttt{nil} in \textit{a-list}, finds that it is equal to \textit{item}, and returns \texttt{nil}, while \texttt{assoc} ignores the \texttt{nil} in \textit{a-list} and continues to search for an actual cons whose cdr is \texttt{nil}.

\begin{verbatim}
(setq family-list
  '(((name . "Larry") (spouse . "Mary")
     (children . ("larry" "fred" "sue")))))
\end{verbatim}

We can then use \texttt{rassoc} to find the pair whose datum is \texttt{string-equal} to \texttt{larry}:

\begin{verbatim}
(rassoc "larry" family-list :test #'string-equal)
=> (name . "Larry")
\end{verbatim}

Or, we could add a \texttt{:key} function, as follows:

\begin{verbatim}
(rassoc "larry" family-list :test #'string-equal
  :key #'car)
=> (children ("larry" "fred" "sue"))
\end{verbatim}

See the function \texttt{assoc}.

For a table of related items: See the section "Functions that Operate on Association Lists".
**zl:rassoc** item in-list  

*Function*

Looks up item in the association list in-list. Returns the first cons whose cdr is zl:equal to item, or nil if there is none such.

For a table of related items: See the section "Functions that Operate on Association Lists".

**rassoc-if** predicate a-list &key :key  

*Function*

Searches the association list a-list and returns the first pair in a-list whose cdr satisfies predicate, nil if none does. The keyword is:

:key  

If not nil, should be a function of one argument that will extract the part to be tested from the whole element. :key is a Symbolics extension to Common Lisp.

Example:

```
(rassoc-if #'integerp '(((eagle . raptor) (1 . 2))) => (1 . 2))
(rassoc-if #'symbolp '(((eagle . raptor) (1 . 2))) =>
(EAGLE   RAPTOR))
(rassoc-if #'floatp '(((eagle . raptor) (1 . 2))) => NIL
```

The function in the following example finds the largest numeric value in an association list by repeating rassoc-if with a test for a datum greater than the greatest datum found so far.

```
(defun find-largest-datum( a-list, &optional (start 0) )
  (if (setq pair
          (rassoc-if #'(lambda(x) (> x start))
                        a-list))
      (find-largest-datum a-list (car pair)))))
```

In the following example, we have an association list consisting of pairs of keys and association lists.

```
(setq financial-statement
  '((MONTHLY-CASH-ON-HAND ((10 . 41)(11 . 52)(12 . 73)))
   (MONTHLY-EXPENSE ((9 . 22)(10 . 20)(11 . 21)))
   (MONTHLY-REVENUE ((9 . 34)(10 . 31)(11 . 42)))
   (setq monthly-cash-on-hand (assoc 'monthly-cash-on-hand financial-statement)))
```

We can then use rassoc-if to find the first pair whose cash-on-hand is greater than 50:

```
(rassoc-if #'(lambda(x) (> x 50)) monthly-cash-on-hand) =>
(11 . 52)
```

For a table of related items: See the section "Functions that Operate on Association Lists".
Compatibility Note: :key is a Symbolics extension to Common Lisp not available in CLOE.

rassoc-if-not predicate a-list &key :key

Function

Searches the association list a-list, and returns the first pair in a-list whose cdr does not satisfy predicate, nil if predicate is satisfied. The keyword is:

: key

If not nil, should be a function of one argument that will extract the part to be tested from the whole element. :key is a Symbolics extension to Common Lisp.

Example:

(rassoc-if-not #'integerp '(((eagle . raptor) (1 . 2))) =>
 (EAGLE . RAPTOR)

(rassoc-if-not #'symbolp '(((eagle . raptor) (1 . 2))) => (1 . 2)

(rassoc-if-not #'symbolp '(((eagle . raptor) (loon . diver)))) => NIL

In the following example, rassoc-if-not finds the first pair in a-list whose datum is not a list:

(setq foo '(((a . (1 4 7))(b . (3 5 8))(c . 100))))

(rassoc-if-not #'listp foo) => (c . 100)

Compatibility Note: :key is a Symbolics extension to Common Lisp and is not available in CLOE. For a table of related items: See the section "Functions that Operate on Association Lists".

zl:rassq item in-list

Function

Looks up item in the association list in-list. Returns the first cons whose cdr is eq to item. Returns nil if none does. zl:rassq means "reverse assq". zl:rassq could have been defined by:

(defun zl:rassq (item in-list)
  (do l in-list (cdr l) (null l)
      (and (eq item (cdar l))
           (return (car l)))))

For a table of related items: See the section "Functions that Operate on Association Lists".

raster-aref raster x y

Function

Accesses the (x,y) graphics coordinate of raster. Use this instead of aref when accessing rasters.

For a table of related items: See the section "Operations on Rasters".
raster-index-offset \texttt{raster x y} \quad \textit{Function}

Returns a linear index of the array element referenced by the (x,y) coordinate of the raster. This can be used as the index to \texttt{sys:*id-aref} or as the :displaced-index-offset argument to \texttt{make-array}.

\texttt{raster-index-offset} is preferred over manual computation and over \texttt{array-row-major-index} when the array is conceptually a raster.

For information about rasters: See the section "Rasters".

For a table of related items: See the section "Operations on Rasters".

\begin{description}
\item[raster-width-and-height-to-make-array-dimensions \texttt{width height} \quad \textit{Function}]
Creates an argument that can be used to call \texttt{make-array}. You would use this in circumstances in which it is not possible to call \texttt{zl:make-raster-array}, for example from the :make-array option to \texttt{defstruct} constructors.

For a table of related items: See the section "Operations on Rasters".
\end{description}

\texttt{ratio} \quad \texttt{&optional (low '*') (high '*')} \quad \textit{Type Specifier}

\texttt{ratio} is the type specifier symbol for the predefined Lisp ratio number type.

The types \texttt{ratio} and \texttt{integer} are disjoint subtypes of the type \texttt{rational}.

In addition to a symbol form, Symbolics Common Lisp provides a list form for \texttt{ratio}. Used in list form, \texttt{ratio} allows the declaration and creation of a specialized set of ratios whose range is restricted to the limits specified in the arguments \texttt{low} and \texttt{high}. \texttt{low} and \texttt{high} must each be an integer, a list of an integer, or unspeci-

fied. If these limits are expressed as integers, they are inclusive; if they are ex-

pressed as a list of an integer, they are exclusive; * means that a limit does not exist, and so effectively denotes minus or plus infinity, respectively. The list form might not work in other implementations of Common Lisp.

Examples:

\begin{verbatim}
(typep -5/2 'ratio) => T
(typep 4/5 '(ratio 0 1)) => T
(typep 2/1 '(ratio 0 1)) => NIL
(typep 2 '(ratio 3 *)) => NIL
(subtypep 'ratio 'rational) => T and T ; subtype and certain
(subtypep '(ratio 2 9) 'rational) => T and T
(subtypep '(ratio 3.2/3 *) 'rational) => T and T
(commonp 15/5) => T
(zl:realp #3r120/21) => T
(sys:type-arglist 'ratio) => (&OPTIONAL (LOW '*') (HIGH '*')) and T
(subtypep '(ratio 0 9) '(rational 0 9)) => T and T
\end{verbatim}

See the section "Data Types and Type Specifiers". See the section "Numbers".
**rational number**

Function

Accepts any non-complex number and converts it to a rational number in canonical form. If the argument is already rational, it is returned. If *number* is in floating-point form, it is assumed to be completely accurate, and **rational** returns a rational number mathematically equal to the precise value of the floating-point number. Note that:

\[(\text{float} \ (\text{rational} \ x)) \equiv x\]

Examples:

- \((\text{rational} \ 0.2) \Rightarrow 13421773/67108864\)
- \((\text{rational} \ 3.95) \Rightarrow 16567501/4194304\)
- \((\text{rational} \ 6/2) \Rightarrow 3\)
- \((\text{rational} \ 0.203) \Rightarrow 13623099/67108864\)
- \((\text{rational} \ 0.000015) \Rightarrow 8246337/549755813888\)

For a table of related items, see the section "Functions that Convert Noncomplex to Rational Numbers".

**rational &optional (low '*') (high '*')**

Type Specifier

**rational** is the type specifier symbol for the predefined Lisp rational number type.

The types **rational**, **float**, and **complex** are pairwise disjoint subtypes of the type **number**.

The type **rational** is a supertype of the following types which are an exhaustive partition of it:

- **integer**
- **ratio**

This type specifier can be used in either symbol or list form. Used in list form, **rational** allows the declaration and creation of specialized rational numbers, whose range is restricted to *low* and *high*.

*low* and *high* must each be a rational, a list of rational numbers, or unspecified. If these limits are expressed as rationals, they are inclusive; if they are expressed as a list of rationals, they are exclusive; * means that a limit does not exist, and so effectively denotes minus or plus infinity, respectively.

Examples:

- \((\text{typep} \ #3r102/21 \ \text{`rational}) \Rightarrow T\)
- \((\text{typep} \ 4 \ ((\text{rational} \ 3 \ 4)) \Rightarrow T\)
- \((\text{typep} \ 5 \ ((\text{rational} \ 3 \ 4)) \Rightarrow \text{NIL}\)
- \((\text{typep} \ 2354 \ ((\text{rational} *)) \Rightarrow T\)
- \((\text{zl:typep} \ 2/3) \Rightarrow :\text{RATIONAL}\)
- \((\text{subtypep} \ \text{rational} \ \text{number}) \Rightarrow T \text{ and T} \ ;\text{subtype and certain}\)
- \((\text{subtypep} \ \text{integer} \ \text{rational}) \Rightarrow T \text{ and T}\)
- \((\text{subtypep} \ \text{ratio} \ \text{rational}) \Rightarrow T \text{ and T}\)
(subtypep '(rational -4 98) '(rational *)) => T and T
(typep 17/89 'common) => T
(rationalp 6/3) => T
(rationalp (+ #2r101 #2r11)) => T
(sys:type-arglist ':rational) => NIL
(sys:type-arglist 'rational) => (&OPTIONAL (LOW '*)) and T
(subtypep '(rational 0 9) 'rational) => T and T

See the section "Data Types and Type Specifiers". See the section "Numbers".

**zr:rational** number

*Function*

In your new programs, we recommend that you use the function *rationalize*, which is the Common Lisp equivalent of the function *zr:rational*.

Converts any noncomplex number to an equivalent rational number. If *number* is a floating-point number, *zr:rational* returns the rational number of least denominator, which when converted back to the same floating-point precision, is equal to *number*.

For a table of related items: See the section "Functions that Convert Noncomplex to Rational Numbers".

**rationalize** number

*Function*

Accepts any non-complex *number* and converts it to a rational number in canonical form. If the argument is already rational, it is returned. If *number* is in floating-point form, *rationalize* assumes that it is accurate only to the precision of the floating-point representation. Hence the returned value can be any rational number for which the floating-point argument is the best available approximation. The aim is to keep both numerator and denominator as small as possible. *rationalize* is guaranteed to return the number with the smallest denominator, such that the following expression is true:

\[(\text{float} (\text{rationalize} \ x)) \equiv x\]

Examples:

- \(\text{rationalize} \ 0.2\) => 1/5
- \(\text{rationalize} \ 3.95\) => 79/20
- \(\text{rationalize} \ 0.203\) => 203/1000
- \(\text{rationalize} \ 0.000015\) => 3/200000

For a table of related items, see the section "Functions that Convert Noncomplex to Rational Numbers".

**rationalp** object

*Function*

This predicate is true if *object* is a rational number (a ratio or an integer) after conversion to canonical form; it is false otherwise.
Examples:

(rationalp 3.0) => NIL
(rationalp 2) => T
(rationalp #c(3 4)) => NIL
(rationalp (/ 22 7)) => T
(rationalp #c(4 0)) => T ;complex canonicalization

The following code tests whether \( a \) and \( b \) are numbers. If they are numbers, they are added. Otherwise, we attempt to extract rationals that are then tested by \texttt{rationalp}.

\[
\begin{align*}
\text{(if (and (numberp a) (numberp b))}
\quad (+ a b) \\
\text{(if (and (consp a)} \\
\quad (rationalp (car a)) \\
\text{(consp b) \\
\text{(rationalp (car b))}} \\
\quad (+ (car a) (car b)) \\
\quad \text{(error "couldn't extract rationals from ~a and ~a" a b)))}
\end{align*}
\]

For a table of related items, see the section "Numeric Type-checking Predicates".

\texttt{zl:rationalp object}$^*$

\textit{Function}

Returns \texttt{t} if \texttt{object} is a ratio. Returns \texttt{nil} if \texttt{object} is an integer or other type of object.

Examples:

(zl:rationalp (/ 8 7)) => T
(zl:rationalp 9/16) => T
(zl:rationalp 4) => NIL
(zl:rationalp (/ 9 3)) => NIL
(zl:rationalp 16/4) => NIL

For a table of related items, see the section "Numeric Type-checking Predicates".

\texttt{read &optional input-stream (eof-error-p t) eof-value recursive-p}$^*$

\textit{Function}

Reads in the printed representation of a Lisp object from \texttt{stream}, builds a corresponding Lisp object, and returns the object.

The optional arguments \texttt{input-stream}, \texttt{eof-error-p}, \texttt{eof-value} and \texttt{recursive-p} affect how \texttt{read} reads and interprets the incoming information. \texttt{input-stream} is the stream from which to obtain input. If unsupplied or \texttt{nil}, it defaults to the value of the special variable \texttt{*standard-input*}. If \texttt{t}, it becomes the value of the special variable \texttt{*terminal-io*}. 
eof-error-p controls what happens if input is from a file (or any other input source that has a definite end) and the end of file is reached. If eof-error-p is t (the default), an error is signalled at the end of file (EOF). If it is nil, then no error is signalled, and instead read returns eof-value.

Because read reads the representation of an object rather than a single character, it always signals an error, regardless of eof-error-p, if the file ends in the middle of an object representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, read will complain. If a file ends in a symbol or a number, immediately followed by EOF, read will read the symbol or number successfully and when called again will see the EOF and only then act according to eof-error-p. If a file contains ignorable text at the end, such as blank lines and comments, read will not consider it to end in the middle of an object. Thus an eof-error-p argument controls what happens when the file ends between objects.

If recursive-p is specified and non-nil, this argument specifies that this call is not a top-level call to read, but an imbedded call. This typically happens from the function for a macro character.

```lisp
(+ (READ) (READ)) => 79
```

```lisp
(with-input-from-string (S "(+ 3 2) (* 5 4)")
  (load-a-stream S))
Q: (+ 3 2)
  A: 5
Q: (* 5 4)
  A: 20
=> NIL
```

For more information on how recursive-p affects input functions: See the section "Input Functions".

The corresponding output function is write.

zl:read &optional (stream zl:standard-input) eof-option Function

Reads in the printed representation of a Lisp object from stream, builds a corresponding Lisp object, and returns the object. For details, see the section "Input Functions".
(This function can take its arguments in the other order, for Maclisp compatibility only.)

**zl:read-and-eval** &optional stream (catch-errors t)  
*Function*

Calls **zl:read-expression** to read a form, without completion. It then evaluates the form and returns the result. If *catch-errors* is not **nil**, it calls **zl:parse-ferror** if an error occurs during the evaluation (but not the reading) so that the input editor catches the error.  
*stream* defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

**read-base**  
*Variable*

The value of **read-base** is a number controlling the radix in which integers and ratios are read. Valid values are between 2 and 36, inclusive; the default is 10 (decimal radix).

The value of **read-base** does not affect rational numbers whose radix is explicitly indicated by a radix specifier, or by a trailing decimal point. See the section "Radix Specifier Format".

The reader uses letters to represent digits greater than 10. Thus, when **read-base** is greater than 10 and no radix specifier is present, some tokens could be read as either integers, floating-point numbers, or symbols. Under Genera the reader's action on such tokens is determined by the value of **si:read-extended-ibase-unsigned-number** and **si:read-extended-ibase-signed-number**. Setting these variables to **t** causes the tokens to be always interpreted as numbers.

**Compatibility Note:** This is an incompatible difference from the language specification in Steele's *Common Lisp* manual. CLOE, however, is compatible with CLtL.

```lisp
(setq foo "23")

(let ((*read-base* 8))
  (values (read-from-string foo)))
=> 19

(let ((*read-base* 10))
  (values (read-from-string foo)))
=> 23
```

For documentation of related variables, see the section "Control Variables for Reading Numbers".

**read-byte** binary-input-stream &optional (eof-error-p t) eof-value  
*Function*

Reads one byte from **binary-input-stream** and returns it in the form of an integer. The corresponding output function is **write-byte**.
(with-open-file (s "data.file"
              :direction :output
              :element-type '(unsigned-byte 2))
  (write-byte 1 s)
  (write-byte 3 s)
  (write-byte 2 s))
=> 2

(with-open-file (s "data.file"
              :direction :input
              :element-type '(unsigned-byte 2))
  (list (read-byte s) (read-byte s) (read-byte s)))
=> (1 3 2)

:read-bytes n-bytes file-position

Sent to a direct access input or bidirectional file stream, requests the transfer of n-bytes bytes from position file-position of the file. The message itself does not return any data to the caller. It causes the stream to be positioned to that point in the file, and the transfer of n-bytes bytes to begin. An EOF is sent following the requested bytes. The bytes can then be read using :tyi, :string-in, or any of the standard input messages or functions. An EOF is sent following the requested bytes.

The stream enforces the byte limit, and presents an EOF if you attempt to read bytes beyond that limit. You must actually read all the bytes and read past (that is, consume from the stream) the EOF.

It is also possible, before all the bytes have been read, to perform stream operations other than reading bytes. For example, an application might read several records at a time, to optimize transfer and buffering, and decide, after reading the first record, to position somewhere else. Direct access file streams handle this properly. Nevertheless, network and buffering resources allocated to the stream (both on the local machine and server machine) are not freed unless all the requested bytes (of the last :read-bytes request) and the EOF following them are read.

If you request more bytes than remain in the file, you receive the remaining bytes followed by EOF.

read-char &optional input-stream (eof-errorp t) eof-value recursive-p Function

Reads and returns a character from input-stream, or if unspecified or nil, *standard-input*. A value of t for input-stream indicates *terminal-io*.

The arguments eof-error-p and eof-value control what happens when the function is called at the end of input-source. If the first argument, eof-error-p is nil, then nothing is done, otherwise an end-of-file error is signalled, and the value returned is eof-value.
The recursive-p argument is used to signal that the call to read-char is not at the top level.

```
(list (read-char) (read-char) (read-char) (read))abcdef
=> (#\a #\b #\c DEF)
```

Note that the character objects produced in response to keyboard input differ between Genera and CLOE 386. Specifying CTRL-A in response to a read-char produces #\c-a under Genera, and \s\ch under CLOE. char-int of these is different also. In the CLOE Developer, specifying control characters in response to read-char results in translation to the char with the appropriate ASCII code, where in the context of :run-program or :run-expression, the variable that controls this behavior is zl:::si*translate-input-to-ascii*. It can assume values of :never, t (meaning always), or the default nil.

```
read-char-no-hang &optional input-stream (eof-error-p t) eof-value recursive-p
```

Function

Performs the same operation as read-char, but if it would be necessary to wait in order to get a character (as from a keyboard), it returns nil immediately, without waiting. This allows you to check for input availability and get the input, if it is available, in the same operation. This is different from the listen operation in two ways. First, read-char-no-hang potentially reads a character, whereas listen never inputs a character. Second, listen does not distinguish between end-of-file (EOF) and no input being available, whereas read-char-no-hang does make that distinction. read-char-no-hang returns eof-value at EOF (or signalling an error of no eof-error-p is true), and always returns nil if no input is available.

A value of t for input-stream indicates *terminal-io*. If input-stream is unspecified or nil, *standard-input* is used. After reading in the printed representation, read-char-no-hang constructs the Lisp object, and returns it. If unable to complete parsing an entire Lisp object, because of end of file or any other reason, read-char-no-hang generates an error.

The arguments eof-error-p and eof-value control what happens when read-char-no-hang is called at the end of input-source. If the first argument, eof-error-p is nil, then nothing is done, otherwise an end-of-file error is signalled, and the value returned is eof-value.

The recursive-p argument signals that the call to read-char-no-hang is not at the top level, and is used to provide the correct behavior in such cases as recursive calls to read-char-no-hang to evaluate read macros.

```
(let ((c (read-char)))
  (list c
    (read-char-no-hang)
    (progn (unread-char c) (read-char-no-hang))))x
=> (#\x NIL #\x)
```

Note that under CLOE, read-char-no-hang does hang, unless it is in the scope of a with-input-editing form.
sys:read-character &optional stream &key (fresh-line t) (any-tyi nil) (eof nil) (notification t) (prompt nil) (help nil) (refresh t) (suspend t) (abort t) (status nil) presentation-context

Function

Reads and returns a single character from stream. This function displays notifications and help messages and reprompts at appropriate times. It is used by fquery and the :character option for prompt-and-read.

stream must be interactive. It defaults to zl:query-io.

Following are the permissible keywords:

:fresh-line If not nil, the function sends the stream a :fresh-line message before displaying the prompt. If nil, it does not send a :fresh-line message. The default is t.

:any-tyi If not nil, the function returns blips. If nil, blips are treated as the :tyi message to an interactive stream treats them. The default is nil.

:eof If not nil and the function encounters end-of-file, it returns nil. If nil and the function encounters end-of-file, it beeps and waits for more input. The default is nil.

:notification If not nil and a notification is received, the function displays the notification and reprompts. If nil and a notification is received, the notification is ignored. The default is t.

:prompt If nil, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor". The default is nil.

:help If not nil, the value should be a help option. See the section "Displaying Help Messages in the Input Editor". Then, when the user presses HELP, the function displays the help option and reprompts. If nil and the user presses HELP, the function just returns #\help. The default is nil.

:refresh If not nil and the user presses REFRESH, the function sends the stream a :clear-window message and reprompts. If nil and the user presses REFRESH, the function just returns #\refresh. The default is t.

:suspend If not nil and the user types one of the sys:kbd-standard-suspend-characters, a zl:break loop is entered. If nil and the user types a suspend character, the function just returns the character. The default is t.

:abort If not nil and the user types one of the sys:kbd-standard-abort-characters, sys:abort is signalled. If nil and the user types an abort character, the function just returns the character. The default is t.
:status

This option takes effect only if the stream is a window. If the value is :selected and the window is no longer selected, the function returns :status. If the value is :exposed and the window is no longer exposed or selected, the function returns :status. If the value is nil, the function continues to wait for input when the window is deexposed or deselected. The default is nil.

:presentation-context

If this is not nil, the presentation system is enabled, that is, presentations that are targets of existing mouse handlers will be sensitive.

:read-cursorpos &optional (units :pixel)

This operation is supported by windows. It returns two values, the current x and y coordinates of the cursor. It takes one optional argument, which is a symbol indicating in what units x and y should be; the symbols :pixel and :character are understood. :pixel means that the coordinates are measured in display pixels (bits), while :character means that the coordinates are measured in characters horizontally and lines vertically.

This operation and :set-cursorpos are used by the zl:format \T request, which is why \T does not work on all streams. Any stream that supports this operation must support :set-cursorpos as well.

*read-default-float-format*

Variable

Controls the printing and reading of floating-point numbers. This variable takes on one of four possible values, namely short-float, single-float, long-float, or double-float.

For printing floating-point numbers:

The printer checks the value of *read-default-float-format* and applies the following rules to decide whether to print an exponent character with the number, and if so, which character.

<table>
<thead>
<tr>
<th>Notation used</th>
<th>Does number’s format match current value of <em>read-default-float-format</em></th>
<th>Exponent marker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordinary</td>
<td>Yes</td>
<td>Don’t print marker</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Print marker and zero</td>
</tr>
<tr>
<td>Exponential</td>
<td>Yes</td>
<td>Print e</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Print marker</td>
</tr>
</tbody>
</table>
See the section “Printed Representation of Floating-point Numbers”.

For reading floating-point numbers:

*read-default-float-format* controls how floating-point numbers with no exponent or an exponent preceded by "E" or "e" are read. Following is a summary of the way possible values cause these numbers to be read.

<table>
<thead>
<tr>
<th>Value</th>
<th>Floating-point precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-float</td>
<td>single-precision</td>
</tr>
<tr>
<td>short-float</td>
<td>single-precision</td>
</tr>
<tr>
<td>double-float</td>
<td>double-precision</td>
</tr>
<tr>
<td>long-float</td>
<td>double-precision</td>
</tr>
</tbody>
</table>

The default value is single-float.

See the section “How the Reader Recognizes Floating-Point Numbers”.

**read-delimited-list** [char] &optional [stream] recursive-p Function

Reads objects from [stream] until the next character after an object’s representation (ignoring whitespace characters and comments) is [char]. read-delimited-list returns a list of the objects read.

To be more precise, read-delimited-list looks ahead at each step for the next non-whitespace character and peeks at it as if with peek-char. If it is [char], the character is consumed, and the list of objects is returned. If it is a constituent or escape character, read is used to read an object, which is added to the end of the list. If it is a macro character, the associated macro function is called, and if that function returns a value, the returned value is added to the list. Then, the peek-ahead process is repeated.

This function is particularly useful for defining new macro characters. Usually it is desirable for the terminating character [char] to be a terminating macro character, so that it may be used to delimit tokens. However, read-delimited-list makes no attempt to alter the syntax specified for [char] by the current readtable. You must make any necessary changes to the readtable syntax explicitly. The following example illustrates this.

Suppose you wanted #{a b c ... z} to read as a list of all pairs of the elements a, b, c, ... z. For example:

#{p q z a} reads as ((p q) (p z) (p a) (q z) (q a) (z a))

This can be done by specifying a macro-character definition for #{ that does two things: reads in all of the items up to the }, and constructs the pairs. read-delimited-list performs the first task.
(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  (mapcon #'(lambda (x)
                (mapcar #'(lambda (y) (list (car x) y)) (cdr x)))
            (read-delimited-list #\} stream t)))

(set-dispatch-macro-character #\# ( #\|#{-reader|)

(set-macro-character #\ (get-macro-character #\) nil)

It is necessary to give a macro definition to the character } as well, to prevent it from being a constituent, as discussed above. Without the definition, the } in the input expression would be considered a constituent character; part of the symbol named a}. You could correct for this by putting a space before the }, but it is cleaner to simply use the call to set-macro-character.

Giving } the same definition as the standard definition of the character ) has the twin benefit of making it terminate tokens for use with read-delimited-list, and also making it illegal for use in any other context. This means that attempting to read a stray } will signal an error.

---

read-delimited-string  delimiters &optional stream eof-error-p eof-value &rest make-array-args

Function

delimiters is either a character or a list of characters. Characters are read from stream until one of the delimiter characters is encountered. The characters read up to the delimiter are returned as a string. This function can be invoked from inside or outside the input editor. If invoked from outside the input editor, the delimiter characters are set up as activation characters. make-array-args are arguments to be passed to make-array when constructing the string to return.

eof-error-p controls what happens if input is from a file (or any other input source that has a definite end) and the end of file is reached. If eof-error-p is t (the default), an error is signalled at the end of file (EOF). If it is nil, no error is signalled, and instead read returns eof-value.

read-delimited-string returns four values:

- The string
- An eof-value, if the eof-error-p parameter was nil
- The character that delimited the string
- Any numeric argument given the delimiter character

This function is used by readline and the :delimited-string option for prompt-and-read.

Examples:

The following reads characters until END is pressed and returns a string at least 200 characters long with a leader-length of 3:
(read-delimited-string #\end *standard-input* nil nil 200. :leader-length 3)

The following is the same as (readline), except that it does not echo a Newline after the string is activated:

(ready-delimited-string '#\return #\line #\end))

A simple word parser:

(ready-delimited-string '#\space #/., #/. #/?))

zl:read-delimited-string &optional (delimiters #end) (stream standard-input) (eof nil) (input-editor-options nil) &rest (make-array-args '(100 :type sys:art-string))

Function
delimiters can be either a character or a list of characters. Characters are read from stream until one of the delimiter characters is encountered. The characters read up to the delimiter are returned as a string. This function can be invoked from inside or outside the input editor. If invoked from outside the input editor, the delimiter characters are set up as activation characters. The eof argument is treated the same way as the eof argument to the :tyi message to non-interactive streams. input-editor-options are passed on as the first argument to the :input-editor message, after having an :activation entry prepended. make-array-args are arguments to be passed to zl:make-array when constructing the string to return.

zl:read-delimited-string returns four values:

• The string
• An eof flag, if the eof parameter was nil
• The character that delimited the string
• Any numeric argument given the delimiter character

This function is used by readline, zl:qsend, and the :delimited-string option for prompt-and-read.

Examples:
The following reads characters until END is pressed and returns a string at least 200. characters long with a leader-length of 3:

(ready-delimited-string #\end standard-input nil nil 200. :leader-length 3)

The following is the same as (readline), except that it does not echo a Newline after the string is activated:

(zl:read-delimited-string '#\return #\line #\end))

A simple word parser:

(zl:read-delimited-string '#\space #/., #/. #/?)

For a more complex example of a sentence parser that uses zl:read-delimited-string: See the section "Examples of Use of the Input Editor".
\textbf{zll:read-expression}  \&optional  \texttt{stream}  \&key  \texttt{(completion-alist  \texttt{nil})  (completion-delimiters  \texttt{nil})}  \texttt{(Function)}

Like \texttt{sys:read-for-top-level} except that if it encounters a top-level end-of-file, it just beeps and waits for more input. This function is used by the \texttt{:expression} option for \texttt{prompt-and-read}.

\texttt{stream}  defaults to  \texttt{zll:standard-input}. This function is intended to read only from interactive streams.

If \texttt{completion-alist} is not \texttt{nil}, this function also sets up \texttt{COMPLETE} and \texttt{c-\texttt{-}}? as input editor commands. When the user presses \texttt{COMPLETE}, the input editor tries to complete the current symbol over the set of possibilities defined by \texttt{completion-alist}. When the user presses \texttt{c-\texttt{-}?), the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. \texttt{completion-alist} can be \texttt{nil}, an alist, an \texttt{sys:art-q-list} array, or a keyword:

\begin{itemize}
  \item \texttt{nil}  \text{No completion is offered.}
  \item \texttt{alist}  \text{The car of each alist element is a string representing one possible completion.}
  \item \texttt{array}  \text{Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.}
  \item \texttt{keyword}  \text{If the symbol is :zmacs, completion is offered over the definitions in Zmacs buffers. If the symbol is :flavors, completion is offered over all flavor names. If the symbol is :documentation, completion is offered over all documentation topics available to Document Examiner.}
\end{itemize}

The default for \texttt{completion-alist} is \texttt{nil}.

\texttt{completion-delimiters}  is \texttt{nil} or a list of characters that delimit "chunks" for completion. As in Zwei, completion works by matching initial substrings of "chunks" of text. If \texttt{completion-delimiters} is \texttt{nil}, the entire text of the current symbol is a single "chunk". The default is \texttt{nil}.

\textbf{si:*read-extended-ibase-signed-number*}  \texttt{(Variable)}

Controls how a token that could be an integer, floating-point number, or symbol and starts with a + or - sign, is interpreted when \texttt{*read-base*} (or \texttt{zll:ibase}) is greater than ten. Here are the possible values of this variable and their effect on the token read.

\begin{itemize}
  \item \texttt{nil}  \text{It is never an integer.}
  \item \texttt{t}  \text{It is always an integer.}
\end{itemize}
:*sharpsign* It is a symbol or floating-point number at top level, but an integer after \#x or \#nR.

:*single* It is a symbol or floating-point number except immediately after \#x or \#nR.

The default value is :sharpsign.

In the table below, the token FACE for each case could be a symbol or a hexadecimal number. :single makes it an integer on the second line, but a symbol on the first and third lines. :sharpsign makes it an integer on both the second and third lines.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>t</td>
<td>:single</td>
<td>:sharpsign</td>
<td></td>
</tr>
<tr>
<td>+FACE</td>
<td>symbol</td>
<td>integer</td>
<td>symbol</td>
<td>symbol</td>
</tr>
<tr>
<td>#x+FACE</td>
<td>symbol</td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>#x(+FACE +FF 1234 +5C00)</td>
<td>symbol</td>
<td>integer</td>
<td>symbol</td>
<td>integer</td>
</tr>
<tr>
<td>+1d0</td>
<td>float</td>
<td>integer</td>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>

Related Topics:

si:*read-extended-ibase-unsigned-number*

si:*read-extended-ibase-unsigned-number* Variable

Controls how a token that could be an integer, floating-point number, or symbol and does not start with a + or - sign, is interpreted when *read-base* (or zl:ibase) is greater than ten. Here are the possible values of this variable and the their effect on the token read.

nil It is never an integer.

:sharpsign It is a symbol or floating-point number at top level, but an integer after \#X or \#nR.

:single It is a symbol or floating-point number except immediately after \#X or \#nR.

The default value is :single.

In the table below, the token FACE for each case could be a symbol or a hexadecimal number. :single makes it an integer on the second line, but a symbol on the first and third lines. :sharpsign makes it an integer on both the second and third lines.
Related Topics:
   \( \text{si:*read-extended-ibase-signed-number}* \)

\textbf{sys:read-for-top-level} &optional (stream \texttt{zl:standard-input}) \texttt{eof-option} Function

Like \texttt{zl:read} but ignores close parentheses seen at top level, and it returns the symbol \texttt{si:eof} if the stream reaches end-of-file if you have not supplied an \texttt{eof-option} (instead of signalling an error as \texttt{zl:read} would). This version of \texttt{zl:read} is used in the system's "read-eval-print" loops.

\textbf{zl:read-form} &optional \texttt{stream} \&key (edit-trivial-errors-p \texttt{zl:*read-form-edit-trivial-errors-p*}) (completion-alist \texttt{zl:*read-form-completion-alist*}) (completion-delimiters \texttt{zl:*read-form-completion-delimiters*}) Function

Like \texttt{zl:read-expression}, but assumes that the returned value will be given immediately to \texttt{eval}. This function is used by the Lisp command loop and by the \texttt{:eval-form} and \texttt{:eval-form-or-end} options for \texttt{prompt-and-read}.

\texttt{stream} defaults to \texttt{zl:standard-input}. This function is intended to read only from interactive streams.

If \texttt{edit-trivial-errors-p} is not \texttt{nil}, the function checks for two kinds of errors. If a symbol is read, it checks whether the symbol is bound. If a list whose first element is a symbol is read, it checks whether the symbol has a function definition. If it finds an unbound symbol or undefined function, it offers to use a lookalike symbol in another package or calls \texttt{zl:parse-ferror} to let the user correct the input. \texttt{edit-trivial-errors-p} defaults to the value of \texttt{zl:*read-form-edit-trivial-errors-p*}. The default value is \texttt{t}.

If \texttt{completion-alist} is not \texttt{nil}, this function also sets up \texttt{COMPLETE} and \texttt{c-?} as input editor commands. When the user presses \texttt{COMPLETE}, the input editor tries to complete the current symbol over the set of possibilities defined by \texttt{completion-alist}. When the user presses \texttt{c-?}, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. \texttt{completion-alist} can be \texttt{nil}, an alist, an \texttt{sys:art-q-list} array, or a keyword:

\texttt{nil} No completion is offered.
alist

The car of each alist element is a string representing one possible completion.

array

Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.

keyword

If the symbol is :zmacs, completion is offered over the definitions in Zmacs buffers. If the symbol is :flavors, completion is offered over all flavor names. If the symbol is :documentation, completion is offered over all documentation topics available to Document Examiner.

The default for completion-alist is the value of zl:*read-form-completion-alist*. The default value is :zmacs.

completion-delimiters is nil or a list of characters that delimit "chunks" for completion. As in Zwei, completion works by matching initial substrings of "chunks" of text. If completion-delimiters is nil, the entire text of the current symbol is a single "chunk". The default is the value of zl:*read-form-completion-delimiters*. The default value is (#- #: #space).

zl:*read-form-completion-alist*

Variable

If not nil, zl:read-form sets up COMPLETE and c-? as input editor commands. When the user presses COMPLETE, the input editor tries to complete the current symbol over the set of possibilities defined by completion-alist. When the user presses c-?, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. zl:*read-form-completion-alist* can be nil, an alist, an sys:art-q-list array, or a keyword:

nil

No completion is offered.

alist

The car of each alist element is a string representing one possible completion.

array

Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.

keyword

If the symbol is :zmacs, completion is offered over the definitions in Zmacs buffers. If the symbol is :flavors, completion is offered over all flavor names. If the symbol is :documentation, completion is offered over all documentation topics available to Document Examiner.

The default value is :zmacs.

zl:*read-form-completion-delimiters*

Variable
The value is \texttt{nil} or a list of characters that delimit "chunks" for completion in \texttt{znl:read-form}. As in Zwei, completion works by matching initial substrings of "chunks" of text. If \texttt{znl:*read-form-completion-delimiters*} is \texttt{nil}, the entire text of the current symbol is a single "chunk". The default value is \texttt{(#\# #: #space)}.

\texttt{znl:*read-form-edit-trivial-errors-p*} \hspace{1cm} \textbf{Variable}

If not \texttt{nil}, \texttt{znl:read-form} checks for two kinds of errors. If a symbol is read, it checks whether the symbol is bound. If a list whose first element is a symbol is read, it checks whether the symbol has a function definition. If it finds an unbound symbol or undefined function, it offers to use a lookalike symbol in another package or calls \texttt{znl:parse-ferror} to let the user correct the input. The default is \texttt{t}.

\texttt{read-from-string} \texttt{string} \&optional \texttt{(eof-errorp \texttt{t})} \texttt{eof-value} \&key \texttt{(:start 0)} \texttt{:end} \texttt{:preserve-whitespace} 

\textbf{Function}

\texttt{read-from-string} gives the characters of \texttt{string} successively to the reader, until a Lisp object can be built.

\texttt{read-from-string} returns two values: The first is the object that was read and the second is the index of the first character in \texttt{string} not read. If the entire string is read, the second object is the length of the string. Macro characters and so on all take effect. If \texttt{string} has a fill-pointer it controls how much can be read.

\textbf{Note:} The \texttt{eof-error-p} and \texttt{eof-value} arguments are optional and must be passed values if the keyword parameters are to be used.

The optional arguments are:

\texttt{eof-error-p} Indicates whether or not to signal an error at the end of a file. A value of \texttt{t} causes the error to be signalled. The default is \texttt{t}.

\texttt{eof-value} Value to be returned if \texttt{eof-error-p} is \texttt{nil} and the end of a file is encountered. The default is \texttt{nil}.

The keywords are:

\texttt{:start} Index of first character to be read from \texttt{string}. The default is 0.

\texttt{:end} Index of first character not to be read from \texttt{string}.

\texttt{:preserve-whitespace} If \texttt{t}, flags the reader to preserve whitespace. The default is \texttt{nil}.

For example:

\begin{verbatim}
(read-from-string "a b c" t nil :preserve-whitespace nil)
\end{verbatim}

\begin{verbatim}
=> A and 2
\end{verbatim}

The expression above returned a value of 2 as an index of the first character not read. The whitespace was ignored.
(read-from-string "a b c" t nil :preserve-whitespace t)
  => A and 1

This expression returned a value of 1 as an index. The whitespace was not ignored.

Example:

(read-from-string "(a b c)")  => (A B C) and 7
(read-from-string "(A B)(C D)"
  => (A B) 5

(read-from-string "(A B)(C D)" nil nil :start 5)
  => (C D) 10

zl:read-from-string string &optional (eof-option 'si:no-eof-option) (start 0) end (preserve-whitespace zl:read-preserving-delimiters) Function

The characters of string are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect. If string has a fill-pointer it controls how much can be read.

eof-option is what to return if the end of the string is reached, as with other reading functions. start is the index in the string of the first character to be read. end, if given, is used instead of (zl:array-active-length string) as the integer that is one greater than the index of the last character to be read.

The flag :preserve-whitespace, if provided and non-nil, indicates that the operation should preserve whitespace as for read-preserving-whitespace. It defaults to nil.

zl:read-from-string returns two values: The first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this is the length of the string.

Example:

(read-from-string "(a b c)")  => (A B C) and 7

:read-input-buffer &optional eof no-hang-p Message

Returns three values: a buffer array, the index in that array of the next input byte, and the index in that array just past the last available input byte. These values are similar to the string, start, end arguments taken by many functions and stream operations.

If the end of the file has been reached and no input bytes are available, the stream returns nil or signals an error, based on the eof argument, just like the :tyi message. If the argument no-hang-p is t and no input is available, the call returns nil and nil.
After reading as many bytes from the array as you care to, you must send the `:advance-input-buffer` message. The data in the buffer is valid only until the `:advance-input-buffer` message is given. At that point, the stream may reuse the buffer for other storage.

**read-line** &optional input-stream (eof-error-p t) eof-value recursive-p

Function

Reads in a line of text. This function is usually used to get a line of input from the user. It returns the line of input and some other values according to the following rules.

read-line and read-line-trim return from one to four values, depending on the kind of input and the values of the `eof-errorp`, `eof-value`, and `recursive-p` arguments:

**Compatibility Note:** The `read-line` function is an extension of the Common Lisp function `read-line`. The Symbolics version of `read-line` returns up to four values; the version as described in CLtL returns two values.

See the section "CLtL Compatibility: Input from Character Streams".

1. A string representing the input. When `eof-errorp` is `nil` and an empty line is terminated by end-of-file, the first value is `eof-value`.

2. A flag indicating whether or not end-of-file occurs while reading the line. No second value is returned when an empty line is terminated by end-of-file.

3. The character that terminates the line, or `nil` if a nonempty line is terminated by end-of-file. This is meaningful only when reading from interactive streams. No third value is returned when an empty line is terminated by end-of-file.

4. Any numeric argument given to the termination character, or `nil` if no argument is given or if a nonempty line is terminated by end-of-file. This is meaningful only when reading from interactive streams. No fourth value is returned when an empty line is terminated by end-of-file.

**Input**

A (possibly empty) line terminated by a character

**Values Returned**

1. The line as a (possibly empty) string without the termination character. `read-line-trim` trims leading and trailing whitespace.

2. `nil`

3. The character that terminates the line

4. Any numeric argument given to the termination character; `nil` if no numeric argument
A nonempty line terminated by end-of-file

1. The line as a string.
2. t
3. nil
4. nil

An empty line terminated by end-of-file

If `eof-errorp` is not `nil`, an error is signalled. The error is interpreted as occurring at top level if `recursive-p` is `nil` and as occurring in the middle of an expression if `recursive-p` is not `nil`.

If `eof-errorp` is `nil`, the only value returned is `eof-value`.

In the following examples, executed in a Lisp Listener, the terminator character, such as `RETURN`, is explicitly shown. Likewise, the end-of-file is inserted by means of `FUNCTION END` and the numeric argument to the termination character is inserted by means of `CONTROL` and a number and also explicitly shown.

Examples:

```lisp
(read-line) fuel consumption way too fast RETURN
"fuel consumption way too fast"
NIL
\Return
NIL
(read-line) Morgan Le Fay FUNCTION END
"Morgan Le Fay"
T
NIL
NIL
(read-line) RETURN
"
NIL
\Return
NIL
(read-line nil nil 365.25) 20,000 Leagues Under the Sea FUNCTION END
"20,000 Leagues Under the Sea"
T
NIL
NIL
```
(read-line)Captain Nemo
"Captain Nemo"
NIL
#\Return
3

See the function \texttt{read-line-trim}.

See the section "Input Functions".

\texttt{read-line-no-echo} &optional \texttt{stream} \texttt{&rest} \texttt{keywords} \texttt{&key} \texttt{(:terminators '#(\Return \Line \End)) \texttt{:full-rubout} \texttt{:(notification} \texttt{t}) \texttt{:prompt} \texttt{:help} \texttt{Function}

Reads a line of input from \texttt{stream} without echoing the input, and returns the input as a string, without the terminating character. This function is used to read passwords and encryption keys. It does not use the input editor but does allow input to be edited using RUBOUT.

\texttt{stream} must be interactive. It defaults to \texttt{zl:query-\textit{io}}.

Following are the permissible keywords:

\begin{description}
\item[\texttt{:terminators}] A list of characters that terminate the input. If the user types \#\texttt{:\Return}, \#\texttt{:\Line}, or \#\texttt{:\End} as a terminator, the function echoes a Newline. If the user types any other character as a terminator, the function echoes that character. The default is \#\texttt{:(\Return \Line \End)}.
\item[\texttt{:full-rubout}] If not \texttt{nil} and the user rubs out all characters on the line, the function returns \texttt{nil}. If \texttt{nil} and the user rubs out all characters on the line, the function waits for more input. The default is \texttt{nil}.
\item[\texttt{:notification}] If not \texttt{nil} and a notification is received, the function displays the notification and reprompts. If \texttt{nil} and a notification is received, the notification is ignored. The default is \texttt{t}.
\item[\texttt{:prompt}] If \texttt{nil}, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor". The default is \texttt{nil}.
\item[\texttt{:help}] If not \texttt{nil}, the value should be a help option. See the section "Displaying Help Messages in the Input Editor". Then, when the user presses \texttt{HELP}, the function displays the help option and reprompts. If \texttt{nil} and the user presses \texttt{HELP}, the function just returns \#\texttt{:#\help}. The default is \texttt{nil}.
\end{description}

\texttt{read-line-trim} &optional \texttt{input-stream} (\texttt{eof-errorp} \texttt{t}) \texttt{eof-value} \texttt{recursive-p} \texttt{Function}
Reads in a line of text and returns it as the first value after trimming leading and trailing whitespace, that is, spaces and tabs. **read-line-trim** takes the same arguments as **read-line** and returns the same values. For a discussion of these values: See the function **read-line**.

```
(read-line-trim)  itchy thumb and fingers  RETURN
  "itchy thumb and fingers"
NIL
#\Return
NIL
```

See the function **read-line-trim**.

See the section "Input Functions".

See the function **zl:readline-trim**.

See the function **zl:readline**.

**si:*read-multi-dot-tokens-as-symbols***

Variable

In Zetalisp, when this function is set to t, it reads tokens containing more than one dot (but no other characters) as symbols. In Common Lisp, when this function is set to nil, it signals an error when it reads tokens containing more than one dot (but no other characters).

**zl:read-or-character** &optional delimiters stream reader

Function

Like **zl:read-expression**, except that if it is reading from an interactive stream and the user types one of the delimiters as the first character or the first character after only whitespace characters, it returns four values: nil, :character, the character code of the delimiter, and any numeric argument to the delimiter. If it encounters any nonwhitespace characters, it calls the reader function with an argument of stream to read the input.

delimiters is a character, a list of characters, or nil. The default is nil. reader defaults to **zl:read-expression**. stream defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

**read-or-end** &optional (stream zl:standard-input) reader

Function

Like **zl:read-expression** except that if it is reading from an interactive stream and the user presses END as the first character or the first character after only whitespace characters, it returns two values, nil and :end. If it encounters any nonwhitespace characters, it calls the reader function with an argument of stream to read the input. reader defaults to **zl:read-expression**. stream defaults to **zl:standard-input**.

The :expression-or-end and :eval-form-or-end options for **prompt-and-read** invoke **read-or-end**.
This function is intended to read only from interactive streams.

:read-pointer

Returns the current position within the file, in characters (bytes in fixnum mode). For text files on PDP-10 file servers, this is the number of Symbolics characters, not PDP-10 characters. The numbers are different because of character-set translation.

z:read-preserve-delimiters

Certain printed representations given to z:read, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the matching close parenthesis serves to mark the end of the list.) Normally z:read throws away the delimiting character if it is "whitespace", but preserves it (with a :unt yi stream operation) if the character is syntactically meaningful, since it might be the start of the next expression.

If z:read-preserve-delimiters is bound to t around a call to z:read, no delimiting characters are thrown away, even if they are whitespace. This might be useful for certain reader macros or special syntaxes.

read-preserving-whitespace &optional input-stream (eof-error-p t) eof-value recursive-p

Certain printed representations given to read, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the matching close parenthesis serves to mark the end of the list.) Normally, read will throw away the delimiting character if it is a whitespace character, but will preserve the character of the next expression.

read-preserving-whitespace is provided for some specialized situations where it is desirable to determine precisely what character terminated the extended token. For example, consider this macro-character definition:

(defun slash-reader (stream char)
  (declare (ignore char))
  (do ((path (list (read-preserving-whitespace stream)))
       (cons (progn (read-char stream nil nil t)
                     (read-preserving-whitespace stream))
       path))
  ((not (char= (peek-char nil stream nil nil t) #\/)))
    (cons 'path (nreverse path))))

(set-macro-character #\/ #'slash-reader)

Consider calling read now on this expression:

(zyedh /usr/games/zork /usr/games/boggle)
The / macro reads objects separated by more / characters, thus /usr/games/zork is intended to read as (path usr games zork). The entire example expression should therefore be read as:

(zyedh (path usr games zork) (path usr games boggle))

However, if read had been used instead of read-preserving-whitespace, after reading the symbol zork the following space would have been discarded, and the next call to peek-char would see the following /. Since the / had already been read, the loop would continue, producing the expression:

(zyedh (path usr games zork usr games boggle))

Note that read-preserving-whitespace behaves exactly like read when the recursive-p argument is non-nil. The distinction is established only by calls with recursive-p equal to nil or omitted.

Note also that this is actually a rather dangerous definition to make, because expressions such as (/ x 3) will no longer read properly. The ability to reprogram the reader syntax is very powerful, and must be used with caution. This redefinition of / is shown here purely for the sake of example.

(list (read) (read-char) (read))foo bar
=> (FOO #\b AR)

(list (read-preserving-whitespace) (read-char) (read))foo bar
=> (FOO #\Space BAR)

si:read-recursive stream

Function
Should be called by reader macros that need to call a function to read. It is important to call this function instead of zl:read in macros that are written in Zetalisp but used by the Common Lisp readtable. In particular, this function must be called by macros used in conjunction with the Common Lisp #n= and #n# syntaxes.

stream is the stream from which to read. This function can be called only from inside a zl:read.

For example, this is the reader macro called when the reader sees a quote ('):

si:(defun xr-quote-macro (list-so-far stream)
   list-so-far ;not used
   (values (list-in-area read-area
            'quote (read-recursive stream))
             'list))

*read-suppress*

Variable
When the value is nil, the Lisp reader operates normally. When it is non-nil, most of the interesting operations of the reader are suppressed; input characters are parsed, but much of what is read is not interpreted.
The primary purpose of *read-suppress* is to support the operation of the read-
time conditional constructs #+ and #-. See the section "Sharp-sign Reader Macros". It is important for these constructs to be able to skip over the printed representa-
tion of a Lisp expression despite the possibility that the syntax of the skipped ex-
pression may not be legal for the current implementation. This is especially useful because a primary application of #+ and #- is to allow the same program to be shared among several Lisp implementations despite small incompatibilities of syn-
tax.

A non-nil value of *read-suppress* has the following specific effects on the Lisp reader:

- All extended tokens are completely uninterpreted; they are discarded and treated as if they were nil. It does not matter whether a token looks like a valid num-
ber or whether the package markers are correct. One consequence of this is that the error concerning improper dotted-list syntax will not be signalled.

- Any standard # macro-character construction that requires, permits, or disallows an infix numerical argument, such as #n, will not enforce any constraint on the presence, absence, or value of such an argument.

- The # construction always produces the value nil. It will not signal an error even if an unknown character name is seen.

- Each of the #b, #o, #x, and #r constructions always scans over a following token and produces the value nil. It will not signal an error even if the token does not have the syntax of a rational number.

- The #* construction always scans over a following token and produces the value nil. It will not signal an error even if the token does not consist solely of the characters 0 and 1.

- Each of the #, and #, constructions reads the following form in suppressed mode but does not evaluate it. The form is discarded and nil is produced.

- Each of the #a, #s, and #: constructions reads the following form in suppressed mode but does not interpret it in any way. It need not be a list in the case of #s, or a symbol in the case of #:. The form is discarded and nil is produced.

- The #= construction is totally ignored. It does not read a following form. It pro-
duces no object, but is treated as whitespace.

- The ## construction always produces nil.

Note that, no matter what the value of *read-suppress* is, parentheses continue to delimit (and construct) lists, the # construction continues to delimit vectors, and comments, strings, and the quote and backquote constructions continue to be interpreted properly. Furthermore, such illegal constructions as '), #<, #), and #<space> continue to signal errors.
In some cases, it may be appropriate for a user-written macro-character definition
to check the value of \texttt{*read-suppress*} and avoid certain computations or side ef-
facts if its value is not \texttt{nil}.

\begin{verbatim}
(setq foo "23")

(let ((*read-suppress* t))
  (read-from-string "foo")
=> nil 3
\end{verbatim}

\texttt{zl:readch} \texttt{&optional stream \texttt{eof-option}} \textit{Function}

Provided for Maclisp compatibility only. \texttt{zl:readch} is just like \texttt{zl:tyi}, except that in-
stead of returning a character object, it returns a symbol whose print name is the
character read in. The symbol is interned in the current package. This is just like a
Maclisp "character object". (This function can take its arguments in the other
order, for Maclisp compatibility only.)

\texttt{zl:readline} \texttt{&optional (stream \texttt{zl:standard-input}) \texttt{eof-option}} \textit{Function}

Reads in a line of text. This function is usually used to get a line of input from
the user. The line of text is normally terminated by \texttt{RETURN}, \texttt{LINE}, or \texttt{END}. If the
line of text is being read from a file stream, it is terminated by a Newline charac-
ter — a Return, or Carriage-Return/Line-Feed, for example — or by end-of-file.
\texttt{zl:readline}, \texttt{zl:readline-trim}, and \texttt{zl:readline-or-nil} return four values, which de-
pend on the kind of input and whether or not the \texttt{eof-option} argument is supplied:

1. A string representing the input. When \texttt{eof-option} is supplied and an empty
line is terminated by end-of-file, the first value is \texttt{eof-option}. When an empty
line is terminated by a character, \texttt{zl:readline-or-nil} returns \texttt{nil}.

2. A flag indicating whether or not end-of-file occurred while reading the line.

3. The character that terminates the line, or \texttt{nil} if the line is terminated by
end-of-file. This is meaningful only when reading from interactive streams.

4. Any numeric argument given to the termination character, or \texttt{nil} if no argu-
ment is given or if the line is terminated by end-of-file. This is meaningful
only when reading from interactive streams.

\begin{tabular}{ll}
\textit{Input} & \textit{Values Returned} \\
A nonempty line terminated by a character & 1. The line as a string without the
termination character. \\
& \texttt{zl:readline-trim} and \\
& \texttt{zl:readline-or-nil} trim leading and trailing whitespace
\end{tabular}
2. \texttt{nil}
3. The character that terminates the line
4. Any numeric argument given to the termination character; \texttt{nil} if no numeric argument is given

An empty line terminated by a character

1. \texttt{zl:readline} and \texttt{zl:readline-trim} return the empty string.
   \texttt{zl:readline-or-nil} returns \texttt{nil}.
2. \texttt{nil}
3. The character that terminates the line
4. Any numeric argument given to the termination character; \texttt{nil} if no numeric argument is given

A nonempty line terminated by end-of-file

1. The line as a string.
   \texttt{zl:readline-trim} and \texttt{zl:readline-or-nil} trim leading and trailing whitespace.
2. \texttt{t}
3. \texttt{nil}
4. \texttt{nil}

An empty line terminated by end-of-file

If \texttt{eof-option} is supplied:
1. \texttt{eof-option}
2. \texttt{t}
3. \texttt{nil}
4. \texttt{nil}

If no \texttt{eof-option} is supplied, an error is signalled.

In the following examples, executed in a Lisp Listener, the terminator character, such as \texttt{RETURN}, is explicitly shown. Likewise, the end-of-file is inserted by means of \texttt{FUNCTION END} and the numeric argument to the termination character is inserted by means of \texttt{CONTROL} and a number and also explicitly shown.

Examples:

\begin{verbatim}
(zl:readline)Bo Diddley caught a bear cat\texttt{RETURN}
"Bo Diddley caught a bear cat"
NIL
#\texttt{Return}
NIL
\end{verbatim}
(zl:readline)To make his pretty baby a Sunday hatCONTROL-3 RETURN
"To make his pretty baby a Sunday hat"
NIL
#\Return
3
(zl:readline)Warren G. HardingEND
"Warren G. Harding"
NIL
#\End
NIL
(zl:readline)FUNCTION END
Error: READLINE encountered an EOF in #:TERMINAL-IO-SYN-STREAM

SI:READLINE-EOF:
   Arg 0 (SI:STREAM): #:TERMINAL-IO-SYN-STREAM
   Arg 1 (SI:EOF-OPTION): SI:NO-EOF-OPTION
s-A, ABORT: Return to Lisp Top Level in Dynamic Lisp Listener 2
s-B: Restart process Dynamic Lisp Listener 2

For more information on the handling of end-of-line characters, such as the
Carriage-Return/Line-Feed combination, see the section "The Character Set".

See the section "Input Functions".

See the function **zl:read-delimited-string**.

See the function **zl:readline-or-nil**.

See the function **zl:readline-trim**.

**zl:readline-no-echo** &optional stream &key (terminators ’(#\return #\line #\end))
(full-rubout nil) (notification t) (prompt nil) (help nil) Function

Reads a line of input from stream without echoing the input, and returns the input
as a string, without the terminating character. This function is used to read pass-
words and encryption keys. It does not use the input editor but does allow input to
be edited using RUBOUT.
stream must be interactive. It defaults to `zl:query-io`.

Following are the permissible keywords:

:terminators A list of characters that terminate the input. If the user types `#\return`, `#\line`, or `#\end` as a terminator, the function echoes a Newline. If the user types any other character as a terminator, the function echoes that character. The default is `(#\return #\line #\end)`.

:full-rubout If not `nil` and the user rubs out all characters on the line, the function returns `nil`. If `nil` and the user rubs out all characters on the line, the function waits for more input. The default is `nil`.

:notification If not `nil` and a notification is received, the function displays the notification and reprompts. If `nil` and a notification is received, the notification is ignored. The default is `t`.

:prompt If `nil`, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor". The default is `nil`.

:help If not `nil`, the value should be a help option. See the section "Displaying Help Messages in the Input Editor". Then, when the user presses HELP, the function displays the help option and reprompts. If `nil` and the user presses HELP, the function just returns `#\help`. The default is `nil`.

======

**zl:readline-or-nil** &optional (stream zl:standard-input) eof-option

Function

Reads in a line of text. It is like `zl:readline` except that `zl:readline-or-nil` returns a first value of `nil` instead of the empty string if the input string is empty. In other respects, it is like `zl:readline-trim` in that it trims leading and trailing whitespace — spaces and tabs — from string input. It takes the same arguments as `zl:readline` and `zl:readline-trim` and returns the same four values. For a discussion of these values: See the function `zl:readline`.

Example:

```
(zl:readline-or-nil)RETURN
NIL
NIL
#\Return
NIL
```

For more examples: See the function `zl:readline`.

See the section "Input Functions".
The :string-or-nil option for prompt-and-read and the :string-or-nil tv:choose-variable-values keyword use zl:readline-or-nil.

See the function zl:readline-trim.

**zl:readline-trim** &optional (stream zl:standard-input) eof-option  

Function

Reads in a line of text. It is like zl:readline except that zl:readline-trim trims leading and trailing whitespace — spaces and tabs — from string input. It takes the same arguments as zl:readline and zl:readline-or-nil and returns the same four values. For a discussion of these values, see the function zl:readline.

Example:

```
(zl:readline-trim)  exciting option  RETURN
"exciting option"
NIL
#\Return
NIL
```

For more examples, see the function zl:readline.


See the section "Input Functions".

See the function zl:readline-or-nil.

**zl:readlist** char-list  

Function

Provided mainly for Maclisp compatibility. char-list is a list of characters. The characters can be represented by anything that the function character accepts: integers, strings, or symbols. The characters are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect.

If there are more characters in char-list beyond those needed to define an object, the extra characters are ignored. If there are not enough characters, an "eof in middle of object" error is signalled.

***readtable***  

Variable

The value is the current readtable. The initial value of this is a readtable set up for standard Common Lisp syntax. You can bind this variable to temporarily change which readtable is being used.

**readtable**  

Type Specifier

A datastructure called a readtable is the type specifier symbol for the predefined Lisp data structure of that name.
The types **readtable**, **hash-table**, **package**, **pathname**, **stream** and **random-state** are **pairwise disjoint**.

Examples:

```
(typep *readtable* 'readtable) => T
(zl:typep *readtable*) => ZL:READTABLE
(subtypep 'readtable 'common) => T and T
(sys:type-arglist 'readtable) => NIL and T
(readtablep *readtable*) => T
```

See the section “Data Types and Type Specifiers”. See the section "The Readtable".

**zl:readtable**

In your new programs, we recommend that you use the variable *readtable*, which is the Common Lisp equivalent of **zl:readtable**.

The value of **zl:readtable** is the current readtable. This starts out as a copy of **si:initial-readtable**. You can bind this variable to temporarily change the readtable being used.

**readtablep object**

Returns **t** if object is a readtable, otherwise returns **nil**.

```
(readtablep (copy-readtable)) => t
```

**realpart number**

If number is a complex number, returns the real part of number. If number is a noncomplex number, returns number.

Examples:

```
(realpart #c(3 4)) => 3
(realpart 4) => 4
```

Related Functions:

  complex
  imagpart

For a table of related items: See the section "Functions that Decompose and Construct Complex Numbers".

**recompile-flavor flavor &key generic ignore-existing-methods (do-dependents t)**

Updates the internal data of flavor and any flavors that depend on it, such as regenerating inherited information about methods. Normally the Flavors system does the equivalent of **recompile-flavor** whenever it is needed.
recompile-flavor is provided so you can recover from unusual situations where the Flavors system does not automatically update the inherited information. These situations include: redefining a function called as part of expanding a wrapper, and recovering from a bug in a method combination routine. If for any reason you suspect that the inherited methods have not been calculated and combined properly, you can use recompile-flavor.

If you supply a non-nil value to generic, only the methods for that generic function are changed. The system does this when you define a new method or redefine a wrapper (when the new definition is not equal to the old). Otherwise, all generic functions are updated.

do-dependents controls whether flavors that depend on the given flavor are also recompiled. By default, all flavors that depend on it are recompiled. You can specify nil for do-dependents to prevent the dependent flavors from being recompiled.

If you supply a non-nil value to ignore-existing-methods, all combined methods are regenerated. Otherwise, new combined methods are generated only if the set of methods to be called has changed. This is the default.

One example of the need for supplying t to ignoring-existing-methods is when you change the way a defwrapper expands, but there is no visible change to the body of the defwrapper. Typically this happens when the wrapper expansion invokes a macro or a subst whose definition has been changed. The same situation can happen for defwhopper-subst, and defmethod and defwhopper when the inline-methods option to defgeneric is used. The Flavors system does not know that anything has changed, and recompiling the wrapper (or whopper or method) does not recompile any combined methods that exists. However, if you supply t to ignore-existing-methods, all combined methods are regenerated.

recompile-flavor affects only flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, and does not affect mixins.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section “Summary of Flavor Functions and Variables”.

record-source-file-name function-spec &optional (type 'defun) (no-query (eq sys:inhibit-fdefine-warnings t))

Function

Associates the definition of a function with its source files, so that tools such as Edit Definition (n-. ) can find the source file of a function. It also detects when two different files both try to define the same function, and warns the user.

record-source-file-name is called automatically by defun, defmacro, defstruct, defflavor, and other such defining special forms. Normally you do not invoke it explicitly. If you have your own defining macro, however, that does not expand into one of the above, you can make its expansion include a record-source-file-name form.

Normally, record-source-file-name returns t. If a definition of the same name and type was already made by another file, the user is asked whether the definition
should be performed. If the user answers "no", \texttt{record-source-file-name} returns \texttt{nil}. When \texttt{nil} is returned the caller should not perform the definition.

\textit{function-spec} The function spec for the entity being defined.

\textit{type} The type of entity being defined, with \texttt{defun} as the default. \textit{type} can be any symbol, typically the name of the corresponding special form for defining the entity. Some standard examples:

\begin{verbatim}
defun
defvar
defflavor
defstruct
\end{verbatim}

Both macros and subs are subsumed under the type \texttt{defun}, because you cannot have a function named \texttt{x} in one file and a macro named \texttt{x} in another file.

\textit{no-query} Controls queries about redefinitions. \texttt{t} means to suppress queries about redefining. The default value of \texttt{no-query} depends on the value of \texttt{sys:inhibit-fdefine-warnings}. When \texttt{sys:inhibit-fdefine-warnings} is \texttt{t}, \texttt{no-query} is \texttt{t}; otherwise it is \texttt{nil}. Regardless of the value for \texttt{no-query}, queries are suppressed when the definition is happening in a patch file.

You cannot specify the source file name with this function. The function is always associated with the pathname for the file being loaded (\texttt{sys:fdefine-file-pathname}).

When redefining functions, some users try to avoid redefinition warnings and queries by using the form \texttt{(remprop symbol :source-file-name)}. The preferred way to do this is to use the form \texttt{(record-source-file-name 'function-spec 'defun t)}. The former method causes the system to forget both the original definition and other definitions for the same symbol (as a variable, flavor, structure, and so forth). \texttt{record-source-file-name} lets the system know that the function is defined in two places, and it avoids redefinition warnings and queries.

Of course, if you are redefining something other than a function, use the appropriate definition type symbol instead of \texttt{defun} as the second argument to \texttt{record-source-file-name}. For example, if you are redefining a flavor, use \texttt{defflavor} as the second argument. See the section "How Programs Manipulate Definitions".

\texttt{reduce} function sequence &key :from-end (:start 0) :end :initial-value (key #'identity) Function

Combines all of the elements of a sequence using a binary operation, for example, using + to sum all of the elements.

\texttt{sequence} is combined or "reduced" using \texttt{function}, which must accept two arguments. The reduction is left-associative, unless the value of the \texttt{:from-end} keyword argument is \texttt{t}, in which case it is right-associative. The first two elements of the indicated subsequence of \texttt{sequence} are combined by using \texttt{function}. The result is
combined with the next element of the subsequence, and so forth, until the sub-
sequence is exhausted, and the result is returned. If the :initial-value argument is
specified, it is logically placed before sequence (or after, if the value of the :from-
end argument is t) and it is included in the reduction operation.

If the specified subsequence contains exactly one element and no :initial-value ar-
gument is specified, that element is returned and function is not called. If the :
start and :end arguments are specified and the subsequence is empty, and the :
initial-value argument is specified, the :initial-value is returned and function is
not called. If the subsequence is empty and no :initial-value is specified, function
is called with zero arguments, and reduce returns whatever the function returns.
(This is the only case where function is called with other than two arguments.)

If a :key argument is supplied, its value must be a function of one argument
which will be used to extract the values to reduce. The :key function will be ap-
plied exactly once to each element of the sequence in the order implied by the re-
duction order but not to the value of the :initial-value argument, if any.

Example:
Using reduce to obtain the total of the ages of the possibly empty sequence of as-
tronauts astros:

(reduce #'+ astros :key #'person-age)

sequence can be either a list or a vector (one-dimensional array). Note that nil is
considered to be a sequence, of length zero.

For example:

(reduce #'+ '(1 2 3 4)) => 10

(reduce #'- '(1 2 3 4) :from-end t) => -2

(reduce #'+ '()) => 0

(reduce #'+ #(1 1 1 1) :start 2 :end 5) => 3

(reduce #'list '(1 2 3 4)) => (((1 2) 3) 4)

(reduce #'list '(1 2 3 4) :initial-value 'foo :from-end t) =>
(1 (2 (3 (4 FOO)))))

In the previous example, + accepts an arbitrary number of arguments; thus, apply
could be used instead of reduce. However, apply can not be used in the following
examples because oddadd accepts exactly two arguments.

(defun oddadd (x y)
  (if (and (oddp x) (oddp y))
      (+ x y) 1))
(reduce #'oddadd '(1 2 3 4 5))
= (oddadd (oddadd (oddadd (oddadd 1 2) 3) 4) 5)
=> 6

(reduce #'oddadd '(1 2 3 4 5) :from-end t)
= (oddadd 1 (oddadd 2 (oddadd 3 (oddadd 4 5)))))
=> 2

The following example illustrates the difference between apply and reduce. Because < is an arbitrary function, apply returns true. However, reduce returns an error because the result of an application of < is not of a suitable type for an argument to <.

(reduce #'< '(1 2 3 4 5)) is erroneous

(apply #'< '(1 2 3 4 5)) => t

For a table of related items: See the section "Mapping Sequences".

clos:reinitialize-instance instance &rest initargs

Reinitializes an existing instance according to initargs (by calling clos:shared-initialize) and returns the initialized instance. This generic function is intended both to be called by users, and to be specialized by users.

instance

The instance to initialize.

initargs

Alternating initialization argument names and values. The set of valid initialization argument names includes:

- Symbols declared by the :initarg slot option to clos:defclass, which are used to initialize the value of a slot.

- Keyword arguments accepted by any applicable methods for clos:reinitialize-instance or clos:shared-initialize.

- The keyword :allow-other-keys. The default value for :allow-other-keys is nil. If you provide t as its value, then all keyword arguments are valid.

The default primary method for clos:reinitialize-instance does the following:

1. Checks the validity of the initargs and signals an error if an invalid initialization argument name is detected.

2. Calls the clos:shared-initialize generic function with the instance, nil, and the initialization arguments provided to clos:reinitialize-instance. The second argument is nil to indicate that no slots are to be initialized from their init-forms.
Note that the usual way for users to customize the reinitialization behavior is to specialize `clos:reinitialize-instance` by writing after-methods. A user-defined primary method would override the default method, and thus could prevent the usual slot-filling behavior.

See the section "Reinitializing a CLOS Instance".

### rem number divisor

Divides `number` by `divisor`, truncating the quotient toward zero, and returns the remainder. This is the same as the second value of `(truncate number divisor)`. If `q` and `r` denote, respectively, the quotient and remainder, then: `q * divisor + r = number`.

The arguments can be rational or floating-point numbers. The returned value, `r`, is rational if both arguments are rational; it is floating-point if either argument is floating-point.

Examples:

```
(rem 3 2) => 1
(rem 3 -2) => 1
(rem -3 2) => -1
(rem -3 -2) => -1
(rem 4 2) => 0
(rem 3.8 2) => 1.8
(rem -3.8 2) => -1.8
(rem 19/5 2) => 9/5
```

When using Genera, the following functions are synonyms of `rem`:

```
zl:\nzl:remainder
```

Related Functions:

```
truncate
mod
```

For a table of related items, see the section "Arithmetic Functions".

### zl:rem pred item list &optional (times most-positive-fixnum)

Returns a copy of `list` with all occurrences of `item` removed. `pred` is used to match the elements of `list` against `item`. `(zl:rem 'eq a b)` is the same as `(zl:remq a b)`.

```
(rem 25 12) → 1
(rem -25 12) → -1
(rem 25 -12) → 1
(rem 4.5 2.2) → 0.1
```
For a table of related items: See the section "Functions for Modifying Lists" and see CLtL 217.

:rem-hash key

Message
Removes any entry for key in the hash table. Returns t if there was an entry or nil if there was not. This message is obsolete; use remhash instead.

zl:rem-if pred list &rest extra-lists

Function
Removes from list those elements that satisfy pred. A new list is made by applying pred to all the elements of list and removing the ones that satisfy it. zl:rem-if does the same thing, but is used if list does not represent a mathematical set.

zl:subset-not and zl:rem-if do the same thing, but they are used in different contexts. zl:subset-not refers to the function's action if list is considered to represent a mathematical set.

pred should be a function of one argument, if there are no extra-lists arguments. If extra-lists is present, each element of extra-lists (that is, each further argument to zl:subset-not or zl:rem-if) is a list of objects to be passed to pred as pred's second argument, third argument, and so on. The reason for this is that pred might be a function of many arguments; extra-lists lets you control what values are passed as additional arguments to pred. However, the list returned by zl:subset-not or zl:rem-if is still a "subset" of the first argument in the various calls to pred.

For a table of related items: See the section "Functions for Modifying Lists".

zl:rem-if-not pred list &rest extra-lists

Function
Removes from list those elements that do not satisfy pred. That is, it keeps the elements for which pred is true. zl:subset does the same thing, but is used if list does not represent a mathematical set.

pred should be a function of one argument, if there are no extra-lists arguments. If extra-lists is present, each element of extra-lists (that is, each further argument to zl:rem-if-not) is a list of objects to be passed to pred as pred's second argument, third argument, and so on. The reason for this is that pred might be a function of many arguments; extra-lists lets you control what values are passed as additional arguments to pred. However, the list returned by zl:rem-if-not is still a "subset" of the first argument in the various calls to pred.

zl:remainder x y

Function
Returns the remainder of x divided by y. x and y must be integers. The exact rules for the meaning of the quotient and remainder of two integers in Zetalisp are given in another section. See the section "Integer Division in Zetalisp".
Examples:

(\texttt{zl:remainder 3 2}) \Rightarrow 1
(\texttt{zl:remainder -3 2}) \Rightarrow -1
(\texttt{zl:remainder 3 -2}) \Rightarrow 1
(\texttt{zl:remainder -3 -2}) \Rightarrow -1

The following functions are synonyms of \texttt{zl:remainder}:

\texttt{rem}
\texttt{zl:\}

\texttt{remf place indicator} \hspace{1em} \textit{Macro}

Searches property list \texttt{place} for a property with an indicator \texttt{eq} to \texttt{indicator}, removes indicator and its value from the property list via splicing, and returns a \texttt{non-nil} value. Otherwise, \texttt{nil} is returned. This macro differs from function \texttt{remprop} in that it takes a place rather than a symbol to indicate the appropriate property list.

In the following example, assume that \texttt{symbol-plist} returns the indicated property list:

\begin{verbatim}
(defvar *some-symbol* (list 'COLOR 'RED 'SPEED 'MYSTICAL 'HIT-POINTS '60))
\end{verbatim}

Then the following calls to \texttt{remprop} give the indicated results:

\begin{verbatim}
(remf *some-symbol* 'speed)
(getff *some-symbol* 'speed 'default-val) \Rightarrow DEFAULT-VAL
(remf *some-symbol* 'magic-user) \Rightarrow nil
\end{verbatim}

See the section "Functions Relating to the Property List of a Symbol".

\texttt{remhash key table} \hspace{1em} \textit{Function}

Removes any entry for \texttt{key} in \texttt{table}. Returns \texttt{t} if there was an entry or \texttt{nil} if there was not.

\begin{verbatim}
(setq company (pop recent-payments))
(unless (remhash company payment-overdue-hash-table)
(setf (gethash company slow-payers-hash-table)
'\texttt{\textbackslash max-days-late-unknown})
\end{verbatim}

For a table of related items: See the section "Table Functions".

\texttt{zl:remhash-equal key hash-table} \hspace{1em} \textit{Function}

Removes any entry for \texttt{key} in \texttt{hash-table}. Returns \texttt{t} if there was an entry or \texttt{nil} if there was not. This function is obsolete; use \texttt{remhash} instead.
In your new programs, we recommend that you use the function `unintern` which is the Common Lisp equivalent of the function `zl:remob`.

`zl:remob` removes `symbol` from `package` (the name is historical and means "RE-Move from OBlist"). `symbol` itself is unaffected, but `intern` no longer finds it in `package`. Removing a symbol from its own package sets its home package to `nil`; removing a symbol from a package different from its home package leaves the symbol's home package unchanged.

`zl:remob` returns `t` if the symbol was found and removed, or `nil` if it was not found.

`zl:remob` is always "local", in that it removes only from the specified package and not from any other packages. Thus `zl:remob` has no effect unless the symbol is present in the specified package, even if it is accessible from that package via inheritance.

If `package` is unspecified it defaults to the symbol's home package. Note this exception well: the default value of `zl:remob`'s `package` argument is *not* the current package.

```lisp
remove item sequence &key (test #'eql) :test-not (#'identity) :from-end (:start 0) :end :count
```

Returns a sequence of the same type as `sequence` that has the same elements, except that those in the subsequence delimited by `:start` and `:end` and satisfying the predicate specified by the `:test` keyword have been removed. This is a non-destructive operation. The returned sequence is a copy of `sequence`, save that some elements are not copied. Elements that are not removed occur in the same order in the result as they did in `sequence`.

For example:

```lisp
(setq nums '(1 2 3)) => (1 2 3)
(remove 1 nums) => (2 3)
nums => (1 2 3)

(remove 2 nums) => (1 3)
nums => (1 2 3)
```

`item` is matched against the elements specified by the `test` keyword. The `item` can be any Symbolics Common Lisp object.

`sequence` can be either a list or a vector (one-dimensional array). Note that `nil` is considered to be a sequence, of length zero. Here is an example of `remove` used with a list:
(setq list '(a b c)) => (A B C)
(remove b list) => (A C)
list => (A B C)

(remove c list) => (A B)
list => (A B C)

:test specifies the test to be performed. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is true. Where testfun is the test function specified by :test, keyfn is the function specified by :key and x is an element of the sequence. The default test is eql.

For example:

(remove 4 #(6 1 6 4) :test #'>) => #(6 6 4)

:test-not is similar to :test, except that the sense of the test is inverted. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is false.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

(remove 0 '((0 1) (0 1) (1 0)) :key #'second) => ((0 1) (0 1))

If the value of the :from-end argument is non-nil, it only affects the result when the :count argument is specified. In that case only the rightmost :count elements that satisfy the predicate are removed.

For example:

(remove 4 '(4 2 4 1) :count 1) => (2 4 1)

(remove 4 #(4 2 4 1) :count 1 :from-end t) => #(4 2 1)

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(remove 'a #(b a a c)) => #(B C)

(remove 4 '(4 4 1)) => (1)
The `:count` argument, if supplied, limits the number of elements removed. If more than `:count` elements of `sequence` satisfy the predicate, then only the leftmost `:count` of those elements are deleted. A negative `:count` argument is equivalent to a `:count` of 0.

For example:

```lisp
(remove 4 '(4 1 4) :count 1) => (2 4 1)
```

`remove` is the non-destructive version of `delete`. The following example uses the `key` function to obtain a value for comparison with `item` by adding one to each element of the sequence. The `item` 3 is passed as the `x` parameter of the anonymous comparison function, and one plus the current sequence element is passed as the `y` parameter. After `count` elements are removed, the value is returned.

Additional examples:

```lisp
(setq a #(1 2 3 4 5 6 7))
(remove 3 a :test #'=) => #(1 2 4 5 6 7)
(remove 3 a :start 1 :key #'1+ :count 3 :test #'(lambda (x y) (x y))) => #(1 2 6 7)
```

For a table of related items: See the section "Functions for Modifying Lists".

---

**zl:remove item list &optional (times most-positive-fixnum)**

Function

Returns a copy of `list` with all occurrences of `item` removed. `zl:equal` is used to match elements of `list` against `item`. `zl:remove` is the non-destructive version of `zl:delete`.

For a table of related items: See the section "Functions for Modifying Lists".

---

**remove-duplicates**

Function

Compares the elements of `sequence` pairwise, and if any two match, discards the one occurring earlier in the sequence. The returned form is `sequence`, with enough elements removed such that no two of the remaining elements match. `remove-duplicates` is a non-destructive function.
sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The function normally processes the sequence in the forward direction, but if a non-nil value is specified for :from-end, processing starts from the reverse direction. If the :from-end argument is true, then the one later in the sequence is discarded.

:test specifies the test to be performed. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is true. Where testfun is the test function specified by :test, keyfn is the function specified by :key and x is an element of the sequence. The default test is eql.

For example:

(remove-duplicates '(1 1 2 2 3 3 3) :test #'>) => (1 1 2 2 3 3 3)
(remove-duplicates '(1 1 2 2 3 3 3) :test #'=) => (1 2 3)

:test-not is similar to :test, except that the sense of the test is inverted. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is false.

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(remove-duplicates '(a a b b)) => (A B)
(remove-duplicates #((1 1 1 1 1)) => #(1)
(remove-duplicates #((1 1 2 2 2) :start 3) => #(1 1 2)
(remove-duplicates #((1 1 2 2 2) :start 2 :end 4) => #(1 1 2 2 2)

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

(remove-duplicates '((Smith S) (Jones J) (Taylor T) (Smith S)) :key #'second) => ((Jones J) (Taylor T) (Smith S))

The value returned by remove-duplicates can share elements with sequence. A list can share a tail with an input list, and the result can be eq to the input sequence if no elements are removed.

In the following example, the key function defines duplicates as a number with the same square as another, or as any other object eql to another. The eql function is the default test. Note that 7 is not removed because it is not duplicated within the
subsequence delimited by \textit{start} and \textit{end}.
\begin{verbatim}
(setq set-a '#(1 2 1 -2 7 4 5 6 7))

(remove-duplicates set-a :end 4 :key #'(lambda(x)(if (numberp x) x 0)) :from-end t)
=> #(1 2 7 4 5 6 7)
\end{verbatim}

\texttt{remove-duplicates} is the non-destructive version of \texttt{delete-duplicates}.
For a table of related items: See the section "Sequence Modification".

\texttt{flavor:remove-flavor} \texttt{flavor-name} \texttt{Function}

Removes the definition of the flavor named by \texttt{flavor-name}. Any accessor functions are also removed from the world.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

\texttt{remove-if} \texttt{predicate sequence} \&\texttt{key :key :from-end (:start 0) :end :count} \texttt{Function}

Returns a sequence of the same type as \texttt{sequence} that has the same elements, except that those in the subsequence delimited by \texttt{:start} and \texttt{:end} and satisfying \texttt{predicate} have been removed. This is a non-destructive operation. The returned sequence is a copy of \texttt{sequence}, save that some elements are not copied. Elements that are not removed occur in the same order in the result as they did in \texttt{sequence}.

For example:
\begin{verbatim}
(setq a-list '(1 a b c)) => (1 A B C)
(remove-if #'numberp a-list) => (A B C)
a-list => (1 A B C)

(setq my-list '(0 1 0)) => (0 1 0)
(remove-if #'zerop my-list) => (1)
my-list => (0 1 0)
\end{verbatim}

\texttt{predicate} is the test to be performed on each element.

\texttt{sequence} can be either a list or a vector (one-dimensional array). Note that \texttt{nil} is considered to be a sequence, of length zero.

The value of the keyword argument \texttt{:key}, if non-\texttt{nil}, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:
\begin{verbatim}
(remove-if #'atom '((book 1) (math (room c)) (text 3)) :key #'second)
=> ((MATH (ROOM C)))
\end{verbatim}
If the value of the :from-end argument is non-nil, it only affects the result when the :count argument is specified. In that case only the rightmost :count elements that satisfy the predicate are deleted.

For example:

(remove-if #'numberp '(4 2 4 1) :count 1)  => (2 4 1)

(remvoue-if #'numberp '(4 2 4 1) :count 1 :from-end t) => (4 2 4)

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(remvoue-if #'atom ('a 1 "list") ) => ('A)

(remvoue-if #'numberp '(4 1 4) :start 1 :end 2)  => (4 4)

(remvoue-if #'evenp '(4 1 4) :start 0 :end 3)  => (1)

The :count argument, if supplied, limits the number of elements deleted. If more than :count elements of sequence satisfy the predicate, then only the leftmost :count of those elements are deleted. A negative :count argument is equivalent to a :count of 0.

For example:

(remvoue-if #'oddp '(1 1 2 2) :count 1)  => (1 2 2)

In the following example, vector elements lists are removed from the result vector if their second element is an odd number:

(setq sequence '(((A 1 2) (B 2 5) (C 3 10) (D 4 17)))

(remvoue-if #'oddp sequence :key #'second)

=> #((B 2 5) (D 4 17))

remove-if is the non-destructive version of delete-if.

For a table of related items: See the section "Sequence Modification".
Returns a sequence of the same type as sequence that has the same elements, except that those in the subsequence delimited by :start and :end which do not satisfy predicate have been removed. The returned sequence is a copy of sequence, save that some elements are not copied. Elements that are not removed occur in the same order in the result as they did in sequence. This is a non-destructive operation.

For example:

\[
\begin{align*}
\text{(setq a-list '(1 a b c)) } &= (1 A B C) \\
\text{(remove-if-not #'numberp a-list) } &= (1) \\
a-list &= (1 A B C) \\
\text{(setq my-list '(0 1 0)) } &= (0 1 0) \\
\text{(remove-if-not #'zerop my-list) } &= (0 0) \\
my-list &= (0 1 0)
\end{align*}
\]

\textit{predicate} is the test to be performed on each element.

\textit{sequence} can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

\[
\begin{align*}
\text{(remove-if-not #'atom '((book 1) (math (room c)) (text 3)) :key #'second) } &= ((BOOK 1) (TEXT 3))
\end{align*}
\]

If the value of the :from-end argument is non-nil, it affects the result only when the :count argument is specified. In that case only the rightmost :count elements that satisfy the predicate are removed.

For example:

\[
\begin{align*}
\text{(remove-if-not #'numberp '(4 'a 'b 1) :count 1) } &= (4 'B 1) \\
\text{(remove-if-not #'numberp ('c 4 2 4 'a) :count 1 :from-end t) } &= ('c 4 2 4)
\end{align*}
\]

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

: \texttt{:start} and : \texttt{:end} must be non-negative integer indices into the sequence. : \texttt{:start} must be less than or equal to : \texttt{:end}, else an error is signalled. It defaults to zero (the start of the sequence).

: \texttt{:start} indicates the start position for the operation within the sequence. : \texttt{:end} indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both : \texttt{:start} and : \texttt{:end} are omitted, the entire sequence is processed by default.
For example:

\[
\begin{align*}
&\text{(remove-if-not #'atom ('a 1 "list"))} \Rightarrow (1 "list") \\
&\text{(remove-if-not #'numberp ('a 'b 'c) :start 1 :end 2) } \Rightarrow ('A 'C) \\
&\text{(remove-if-not #'evenp (1 2 3 5) :start 0 :end 3) } \Rightarrow (2 5)
\end{align*}
\]

The :count argument, if supplied, limits the number of elements deleted. If more than :count elements of sequence satisfy the predicate, then only the leftmost :count of those elements are deleted. A negative :count argument is equivalent to a :count of 0.

For example:

\[
\begin{align*}
&\text{(remove-if-not #'oddp (1 1 2 2) :count 1 ) } \Rightarrow (1 1 2)
\end{align*}
\]

\textit{remove-if-not} is the non-destructive version of \textit{delete-if-not}.

For a table of related items: See the section "Sequence Modification".

**clos:remove-method**

\textit{generic-function method}  
\textit{Generic Function}

Removes a method from a generic function and returns the modified generic function.

\textit{generic-function}  
A generic function object.

\textit{method}  
A method object.

If the method is not one of the methods on the generic function, no action is taken and no error is signaled.

**remove-proclaims**

\textit{fspec}  
\textit{Function}

Removes any proclamations associated with \textit{fspec}. This function is a Symbolics extension to Common Lisp.

See the function \textit{proclaim}.

**remprop**

\textit{symbol indicator}  
\textit{Function}

Removes from the property list in \textit{symbol} a property with an indicator \textit{eq} to \textit{indicator}. For example, if the property list of \textit{foo} was:

\[
\text{(color blue height six-three near-to bar)}
\]

then:

\[
\text{(remprop 'foo 'height) } \Rightarrow \text{(six-three near-to bar)}
\]

and \textit{foo}'s property list would be:

\[
\text{(color blue near-to bar)}
\]
If the property list has no *indicator*-property, then **remprop** has no side-effect and returns **nil**.

See the section "Functions Relating to the Property List of a Symbol".

For a table of related items: See the section "Functions That Operate on Property Lists".

**zl:remprop** *sym* *indicator*

Removes *sym*’s *indicator* property, by splicing it out of the property list. It returns that portion of the list inside *sym* of which the former *indicator*-property was the car. The car of what **zl:remprop** returns is what **zl:get** would have returned with the same arguments. **zl:remprop** uses the property lists associated with the symbol. For example, if the property list of **foo** was:

```
(color blue height six-three near-to bar)
```

then:

```
(zl:remprop 'foo 'height) => (six-three near-to bar)
```

and **foo**’s property list would be:

```
(color blue near-to bar)
```

If *sym* has no *indicator*-property, then **zl:remprop** has no side-effect and returns **nil**.

For a table of related items: See the section "Functions That Operate on Property Lists".

Searches the property list of symbol for a property with an indicator **eq** to *indicator*, removes the indicator value pair from the property list via splicing, and returns a non-nil value. Otherwise, **nil** is returned.

In the following example, assume that **symbol-plist** returns the indicated property list:

```
(setf (get 'some-symbol 'hit-points) '60)
(setf (get 'some-symbol 'speed) 'mystical)
(setf (get 'some-symbol 'size) 'large)
(setf (get 'some-symbol 'color) 'red)
```

```
(symbol-plist 'some-symbol)
→ (COLOR RED SIZE LARGE SPEED MYSTICAL HIT-POINTS 60)
```

The following calls to **remprop** produce the results as indicated:

```
(get 'some-symbol 'size) → LARGE

(remprop 'some-symbol 'size)

(get 'some-symbol 'size) → NIL
```
(remprop 'some-symbol 'speed)

(get 'some-symbol 'speed) → NIL

(symbol-plist 'some-symbol)

→ (COLOR RED HIT-POINTS 60)

See Also: CLtL 166, get

zl:remq item list &optional (times most-positive-fixnum) Function

Returns a copy of list with all occurrences of item removed. eq is used for the comparison. zl:remq is the non-destructive version of zl:delq. Examples:

(setq x '(a b c d e f))
(zl:remq 'b x) => (a c d e f)
x => (a b c d e f)
(zl:remq 'b '(a b c b a b) 2) => (a c a b)

For a table of related items: See the section "Functions for Modifying Lists".

rename new-name Message

 Renames the file open on this stream. You should not use :rename. Instead, use rename-file.

flavor:rename-instance-variable flavor-name old new Function

 Renames an instance variable old to a new name new for the given flavor-name. When this is done, the value of the old instance variable is carried over to the new instance variable. Any old instances are updated to reflect the new name of the instance variable. Often you use flavor:rename-instance-variable first, which ensures that the value of the instance variable is carried over. You might then use defflavor to add options such as :readable-instance-variables, or change the default initial value.

(flavor:rename-instance-variable 'ship 'captain 'skipper)

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

rename-package pkg new-name &optional new-nicknames Function

 Replaces the old name and all old nicknames of pkg with new-name and new-nicknames. new-name is a string or a symbol. new-nicknames is a list of strings or symbols. new-nicknames defaults to nil.
In the following example, \texttt{package-nicknames} is used to retrieve the current list of nicknames for an existing package and then \texttt{rename-package} is used to add a new nickname to that package.

\begin{verbatim}
(defun add-nickname (package new-nickname)
  (rename-package package (package-name package)
     :nicknames (cons new-nickname (package-nicknames package))))
\end{verbatim}

See the section "Mapping Between Names and Packages".

\textbf{si:rename-within-new-definition-maybe} \textit{function definition}

Given \texttt{new-structure} that is going to become a part of the definition of \texttt{function-spec}, performs on it the replacements described by the \texttt{si:rename-within} encapsulation in the definition of \texttt{function-spec}, if there is one. The altered (copied) list structure is returned.

It is not necessary to call this function yourself when you replace the basic definition because \texttt{fdefine} with carefully supplied as \texttt{t} does it for you. \texttt{si:encapsulate} does this to the body of the new encapsulation. So you only need to call \texttt{si:rename-within-new-definition-maybe} yourself if you are replacing part of the definition.

For proper results, \texttt{function-spec} must be the outer-level function spec. That is, the value returned by \texttt{si:unencapsulate-function-spec} is not the right thing to use. It has had one or more encapsulations stripped off, including the \texttt{si:rename-within} encapsulation if any, and so no renamings are done.

\textbf{repeat} Keyword for loop

Repeat is one of the iteration-driving clauses for \texttt{loop}.

\textbf{repeat} \textit{expression}

Evaluates \textit{expression} (during the variable-binding phase), and causes the \texttt{loop} to iterate that many times. \textit{expression} is expected to evaluate to an integer. If \textit{expression} evaluates to a 0 or negative result, the body code is not executed.

Examples:

\begin{verbatim}
(defun loop1 (how-far)
  (loop repeat how-far
       for x from 1 to 1000 by 2
       do
         (princ x) (princ " ")))
\end{verbatim}

\begin{verbatim}
(loop1 5) => LOOP1
(loop1 9) => 1 3 5 7 9 NIL
\end{verbatim}

See the section "Iteration-Driving Clauses".
replace sequence1 sequence2 &key (start1 0) (end1 0) (start2 0) (end2 0)  Function

Destructively modifies sequence1 by copying into it successive elements from sequence2.

sequences can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero. The elements of sequence2 must be of a type that can be stored into sequence1.

The keyword arguments :start1, :end1, :start2, and :end2 are used to specify subsequences of sequence1 and sequence2.

:start1 and :end1 must be non-negative integer indices into the sequence. :start1 must be less than or equal to :end1, else an error is signalled. It defaults to zero (the start of the sequence).

:start1 indicates the start position for the operation within the sequence. :end1 indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence). If both :start1 and :end1 are omitted, the entire sequence is processed by default.

:start2 and :end2 operate the same as :start1 and :end1.

If the subsequences delimited by :start1, :start2, :end1 and :end2 are not of the same length, the shorter length determines how many elements are copied. The extra elements near the end of the longer subsequence are not involved in the operation. The number of elements copied can be expressed as:

\[
\text{min}(-\text{end1 start1}) (-\text{end2 start2})
\]

If sequence1 and sequence2 are the same (eq) object and the region being modified overlaps the region being copied from, it is as if the entire source region were copied to another place, and only then copied back into the target region. However, if sequence1 and sequence2 are not the same, but the region begin modified overlaps the region being copied from, after the replace operation the subsequence of sequence1 being modified will have unpredictable contents.

For example:

(setq bird-list '(heron flamingo loon owl)) => (HERON FLAMINGO LOON OWL)
(replace bird-list bird-list :start2 2 :end2 3) => (LOON FLAMINGO LOON OWL)
bird-list => (LOON FLAMINGO LOON OWL)
(setq bird-list '(heron flamingo loon owl)) => (HERON FLAMINGO LOON OWL)
(replace bird-list '(hawk turkey) :start1 1 :end1 3) => (HERON HAWK TURKEY OWL)
(setq a #(1 2 3 4 5) b #*1001010100110)
(replace a b :start1 1 :end1 3 :start2 3 :end2 9)
=> #(1 1 0 4 5)

In the previous example, only the second and third vector elements are replaced because
(< (- end1 start1) (- end2 start2))

For a table of related items: See the section "Sequence Modification". Also: See the section "Copying an Array".

\textbf{dbg:report} condition stream \textit{Generic Function}

Prints the text message associated with this object onto stream. The \textit{condition} flavor does not support this itself, but you must provide a handler, and any flavor built on \textit{condition} that is instantiated must support this function.

The compatible message for \textbf{dbg:report} is:

:report

For a table of related items, see the section "Basic Condition Methods and Init Options".

\textbf{dbg:report-string} condition \textit{Generic Function}

Returns a string containing the report message associated with this object. It works by sending :report to the object.

The compatible message for \textbf{dbg:report-string} is:

:report-string

For a table of related items: see the section "Basic Condition Methods and Init Options".

\textbf{require} module-name &optional pathname \textit{Function}

Checks the list in *modules* to see if module-name is already loaded; if it is not, \textbf{require} loads the appropriate file or set of files. module-name can be a string or a symbol representing a module. pathname can be a single pathname or a list of pathnames to be loaded in order, left to right.

In the following code, the call to \textbf{require} loads the \textbf{turbine-package} module, and if \textbf{turbine-speed} were a constant in \textbf{turbine-package}, then its value would be available at this point.
=> *modules*
(GENERATOR-PACKAGE LISP)
=> (require 'turbine-package)
TURBINE-PACKAGE
=> turbine-package:turbine-speed
3600

si:resource-error

All resource-related error conditions are built on si:resource-error. Used primarily for zl:typep.

si:resource-extra-deallocation

Detects situations where there is extra deallocation, and enters the Debugger. Extra deallocation occurs when deallocate-resource is called more than one time on an object.

Use the no-action message to ignore this error. The :object message returns the object. The :resource message returns the resource.

si:resource-object-not-found

Signifies an error in the client and gives the error message "Object not found in resource". This occurs when a deallocated object was not found in the resource.

This situation can be created in two ways:

- Not creating the object on the resource with the following:
  (si:allocate-resource <resource name>...)

- Executing the following form between the original allocation, and the deallocation:
  (si:clear-resource <resource name>)

Use the :no-action proceed type to ignore this error. The :object message returns the object. The :resource message returns the resource.

rest x

Returns the tail (cdr) of list or cons x, and mnemonically complements the function first. setf can be used with rest to replace the cdr of a list with a new value. For example:

(setq item-list '(loon eagle)) => (LOON EAGLE)
(setf (rest item-list) 'heron) => HERON

item-list => (LOON . HERON)

In many cases, **rest** is stylistically preferable to **cdr** for readability.

(let ((element (first element-list))
      (details (rest element-list)))
  (if (member element goodlist :test #'eq)
      (do-something details)))

For a table of related items: See the section "Functions for Extracting from Lists".

---

&rest

Lambda List Keyword

If present, the following specifier is a single rest parameter specifier. There can only be one &rest argument.

It is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is stored in the function-calling stack, and loses its validity when the function returns. If a rest-argument is to be returned or made part of permanent list-structure, it must first be copied, as you must always assume that it is one of these special lists. See the function **sys:copy-if-necessary**.

The system does not detect the error of omitting to copy a rest-argument; you simply find that you have a value that seems to change behind your back. At other times the rest-args list is an argument that was given to **apply**; therefore it is not safe to **rplaca** this list, as you might modify permanent data structure. An attempt to **rplacd** a rest-args list is unsafe in this case, while in the first case it causes an error, since lists in the stack are impossible to **rplacd**.

---

zl:rest1 list

Function

Returns the rest of the elements of a list, starting with element 1 (counting the first element as the zeroth). Thus, **zl:rest1** is equivalent to **cdr**; the reason this function is provided is that it makes more sense when you are thinking of the argument as a list rather than just as a cons.

For a table of related items: See the section "Functions for Extracting from Lists".

---

zl:rest2 list

Function

Returns the rest of the elements of a list, starting with element 2 (counting the first element as the zeroth). Thus, **zl:rest2** is equivalent to **cddr**; the reason this function is provided is that it makes more sense when you are thinking of the argument as a list rather than just as a cons.
For a table of related items: See the section "Functions for Extracting from Lists".

**zl:rest3 list**

*Function*

Returns the rest of the elements of a list, starting with element 3 (counting the first element as the zeroth). Thus, *zl:rest2* is equivalent to *cdadr*. The reason this function is provided is that it makes more sense when you are thinking of the argument as a list rather than just as a cons.

For a table of related items: See the section "Functions for Extracting from Lists".

**zl:rest4 list**

*Function*

Returns the rest of the elements of a list, starting with element 4 (counting the first element as the zeroth). Thus, *zl:rest4* is equivalent to *cdadr*. The reason this function is provided is that it makes more sense when you are thinking of the argument as a list rather than just as a cons.

For a table of related items: See the section "Functions for Extracting from Lists".

**return &optional result**

*Special Form*

Returns control and a result value (or values) from an unnamed block. Such blocks are established by *(block nil ...)*. Among the macro constructs which establish such blocks are *do*, *dolist*, *dotimes*, unnamed *prog*, and *loop*.

To return more than one result value, use *values*. For example, *(return (values 'A 'B))* will return two values, and *(return (values))* will return no values.

It is also permissible to omit the result, as in *(return)*. This notation is functionally the same as *(return nil)*, but is usually used to emphasize the fact that the resulting value is not important. If the resulting value is significant in any way, it is recommended that you write *(return nil)* explicitly to emphasize the fact.

*(return result)* is functionally equivalent to *(return-from nil result)*. See the special form *return-from*.

Examples:

```lisp
;; find first even element
(dolist (j '(3 7 22 9 7)) (when (evenp j) (return j)))
=> 22
```
;; find position and value of first duplicated element
(let ((v '(2 7 16 61 7 4 4 9)) (n 0))
  (dotimes (j (length v))
    (let ((x (aref v j)))
      (when (= x n) (return (values (- j 1) x)))
      (setq n x))))
=> 5
4

;; one way to select a substring (there are much better ways)
(with-output-to-string (stream)
  (do ((string "To be or not to be? That is the question."))
       (index 0 (+ index 1))
    ((= index 5))
    (when (= index 5) (return))
    (write-char (char string index) stream)))
=> "To be"

Note that if you are using Genera, the function zl:break, the read-eval-print loop you enter recognizes the typed-in form (return result) specially. If this form is typed at such a breakpoint, result is evaluated and returned as the value of zl:break. If the result form itself returns multiple values, they are all returned as the value of zl:break. See the special form zl:break. Note that this special case relating to breakpoints does not exist in the CLOE Runtime system.

If not specially recognized by zl:break and not inside a block, return signals an error.

Zetalisp Note: In a past release, (return form1 form2 ...) meant what (return (values form1 form2 ...)) means now. In most cases, the compiler will warn you if you use the old syntax, and try to correct your error. In the case of (return), the compiler cannot be sure of your intent and so will normally assume that you mean (return nil), which is the modern interpretation. If you think you have old code which intends (return (values)) instead, you can set the variable compiler:*return-style-checker-on* to t in order to cause the compiler to warn you about this construct as well.

See the section "Blocks and Exits Functions and Variables".

sys:return-array array

 Function
Attempts to return array to free storage. It is is a subtle and dangerous feature that should be avoided by most users. If it is displaced, this returns the displaced array itself, not the data that the array points to. Because of the way storage allocation works, sys:return-array does nothing if the array is not at the end of its region, that is, if it was not the most recently allocated non-list object in its area. sys:return-array returns t if storage was really reclaimed, or nil if it was not.
It is the responsibility of any program that calls `sys:return-array` to ensure that there are no references to `array` anywhere in the Lisp world. This includes locative pointers to array elements, such as you might create with `zl:aloc`. The results of attempting to use such a reference to the returned array are unpredictable. Simply holding such a reference in a local variable, without attempting to access it or to print it out, is allowed, although it might thwart the garbage collector.

Other tools are available for manually allocating and freeing arrays. See the special form `sys:with-stack-array`.

```
return-from block-name &optional result
```

*Special Form*

Exits from a block or a construct such as `do` or `prog` that establishes an implicit block around its body.

The `value` subforms are optional. Any `value` subforms are evaluated, and the resulting values (either multiple, or none) are returned from the innermost block that has the same name and that lexically contains the `return-from` form. The returned values depend on how many `value` subforms are provided and on the syntax used as shown below:

<table>
<thead>
<tr>
<th>Value subforms</th>
<th>Syntax</th>
<th>Values returned from block</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td><code>(return-from name)</code></td>
<td><code>nil</code></td>
</tr>
<tr>
<td>None</td>
<td><code>(return-from name (values))</code></td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td><code>(return-from name value)</code></td>
<td>All values that result from evaluating the <code>value</code> subform</td>
</tr>
<tr>
<td>&gt;1</td>
<td><code>(return-from name (values value))</code></td>
<td>One value from each <code>value</code> subform</td>
</tr>
</tbody>
</table>

**Zetalisp Note:** The form `(return form1 form2 form3...)` is no longer valid, and generates a compiler message to that effect. Use the form `(return (values form1 form2 form3...))` to have multiple values returned.

Similarly, if you omit `value`, `return` now defaults to `nil`, rather than returning with zero values as formerly; the compiler generates a message to that effect also. Use `(return (values))` if you want zero values returned.

The variable `compiler:*return-style-checker-on*` controls compiler messages for these invalid formats of `return`. To disable the compiler messages specify a `nil` value for `compiler:*return-style-checker-on*`. 
block-name is not evaluated. It must be a symbol.

The scope of name is lexical. That is, the return-from form must be inside the block itself (or inside a block that that block lexically contains), not inside a function called from the block.

When a construct like do or an unnamed prog establishes an implicit block, its name is nill. You can use either (return-from nill value...) or the equivalent (return value...) to exit from such a construct.

The return-from form is unusual: It never returns a value itself, in the conventional sense. It is not useful to write (setq a (return-from name 3)), because when the return-from form is evaluated, the containing block is immediately exited, and the setq never happens.

Examples:

```
(block foo
  (print "enter foo")
  (when (< 1 2)
    (return-from foo (values 1 2 3 4)))
  (print "leave foo"): => "enter foo" 1 and 2 and 3 and 4

(block state-of
  (princ "H-2-0 ")
  (return-from state-of (values-list '(Ice Water Steam)))
  (princ "ice-cream"): => H-2-0 ICE and WATER and STEAM

(setq stuff '(north east south west right left up down))
 => (NORTH EAST SOUTH WEST RIGHT LEFT UP DOWN)

(defun index-of-thing (thing stuff)
  (do ((count 1 (+ count 1)))
      ((= count (length stuff)))
    (if (eq thing (car stuff))
      (return-from index-of-thing count))
    (setq stuff (cdr stuff))))
 => INDEX-OF-THING

(index-of-thing 'south stuff) => 3

(do ((j 0 (+ 1)))
    (nil) ; Do forever
    (format t "~%Input ~D: " j)
    (let ((item (read)))
      (if (null item) (return-from nil) ;Process items until nil seen.
        (format t "~&Output ~D: ~S" j (print item))))))
 => Input 0:
 ABCDEF
 Output 0: ABCDEF
 Input 1: NIL
```
For an explanation of named dos and progs in Zetalisp: See the special form zl:do-named.

Following is an example, returning a single value from an implicit block named nil:

Examples:

```
(do ((x x (cdr x))
    (n 0 (* n 2)))
   (null x) n)
(cond ((atom (car x))
       (setq n (+ n 1)))
   ((memq (caar x) '(sys boom bleah))
    (return-from nil n)))
```

Or

```
(block nil
 (print "rivers hills")
 (if (= 3 3.) (return-from nil "five")
 (print "water trees")) => "rivers hills" "five"
```

Following is another example, returning multiple values. The function below is like assoc, but it returns an additional value, the index in the table of the entry it found:

```
(defun assocn (x table)
   (do ((l table (cdr l))
       (n 0 (+ n 1)))
      (null l) nil)
   (if (eql (caar l) x)
      (return-from nil (values (car l) n))))
```

In the second example that follows, defun establishes an implicit block named foo around the defined function.

```
(block foo
 (block bar
   (let ((fred (my-compute *input-data*)))
     (if (symbolp fred) (return-from foo fred))
    (if (numberp fred) (return-from bar fred))
    (setq *in-process* (my-process-data fred)))
 (if (numberp *in-process*)
   (my-select-version-from-number *in-process*)
    (if (symbolp *in-process*) (*in-process* nil)))
```
(defun foo (a-number)
  (if (not (numberp a-number)) (return-from foo nil))
  (let ((num a-number)
             (result 0))
    (dotimes (i num result)
      (if (= i 20) (return result))
      (setq result (+ result (expt i 2))))))

For a table of related items: See the section "Blocks and Exits Functions and Variables".

return Keyword for loop

return expression

Immediately returns the value of expression as the value of the loop, without running the epilogue code. This is most useful with some sort of conditionalization, as discussed in the previous section. Unlike most of the other clauses, return is not considered to "generate body code", so it is allowed to occur between iteration clauses, as in:

(loop for entry in list
      when (not (numberp entry))
      return (error...)
      as from = (times entry 2)
      ... )

If you instead want the loop to have some return value when it finishes normally, you can place a call to the return function in the epilogue (with the finally clause).

See the section "loop Clauses".

zl:return-list form

Special Form
An obsolete function supported for compatibility with earlier releases. It is like return except that the block returns all of the elements of form as multiple values. This means that the following two forms are equivalent:

(return-list form)

(return (values-list form))

Examples:
The latter form is the preferred way to return list elements as multiple values from a block named nil. To direct the returned values to a named block, use:

\textbf{(return-from name (values-list form)).}

Example:

\begin{verbatim}
(block state-of
 (princ "H-2-O ")
 (return-from state-of (values-list '(Ice Water Steam)))
 (princ "ice-cream") => H-2-O
 ICE
 WATER
 STEAM
\end{verbatim}

For a table of related items: See the section "Blocks and Exits Functions and Variables".

\textbf{compiler:*return-style-checker-on*}

Variable

This style-checker variable is associated with the functions \texttt{return} and \texttt{return-from} and controls the display of compiler messages for invalid formats of these functions. The documentation for \texttt{return} and \texttt{return-from} describes the specific formats activating the style-checker.

\texttt{compiler:*return-style-checker-on*} is set to \texttt{t} by default; set it to \texttt{nil} to disable the compiler messages.

For a table of related items: See the section "Blocks and Exits Functions and Variables".
revappend x y  

Reverse the elements of list x and appends x to y, returning the resulting new list. (revappend x y) is functionally the same as (append (reverse x) y), except that it is potentially more efficient. The values of both x and y should be lists. The value of the x argument is copied, not destroyed. For example:

```
(setq a-list '(a b c)) => (A B C)
(setq b-list '(x y z)) => (X Y Z)
(revappend a-list b-list) => (C B A X Y Z)
a-list => (A B C)
(setq back '(c b a))
(revappend back '(d e f)) => (A B C D E F)
```

In the following example, revappend sorts queued entries in order of priority.

```
(defun sort-queue-1 (in-queue)
"Sorts arg first by priorities (car element), then by original order."
(let ((for-queue1 '())
     (for-queue2 '())
     (for-queue3 '()))
  (dolist (queue-element in-queue)
    (case (car queue-element)
      (1 (push queue-element for-queue1))
      (2 (push queue-element for-queue2))
      (3 (push queue-element for-queue3)))))
;; reverse the temporary lists
;; that were built by push
(revappend for-queue1
           (revappend for-queue2
                     (reverse for-queue3)))
(setq queue-all '((1 element-a) (2 element-b) (3 element-c) (2 element-d) (1 element-e)))
(sort-queue queue-all) =>
(((1 ELEMENT-A) (2 ELEMENT-B) (3 ELEMENT-C) (2 ELEMENT-D) (1 ELEMENT-E)))
```

For a table of related items: See the section "Functions for Constructing Lists and Conses".

reverse sequence  

Returns a new sequence of the same type as sequence, containing the same elements in reverse order. This operation is non-destructive.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:
(reverse '(heron flamingo loon)) => (LOON FLAMINGO HERON)

(reverse #(1 2 3)) => #(3 2 1)

For a table of related items: See the section "Functions for Modifying Lists".

For a table of related items: See the section "Sequence Modification".

**zl:reverse list**

Creates a new list whose elements are the elements of *list* taken in reverse order. *zl:reverse* does not modify its argument, unlike *zl:nreverse*, which is faster but does modify its argument. The list created by *zl:reverse* is not cdr-coded. Example:

(zl:reverse '(a b (c d) e)) => (e (c d) b a)

*zl:reverse* could have been defined by:

(defun zl:reverse (x)
  (do ((l x (cdr l)) ; scan down argument,
       (r nil        ; putting each element
        (cons (car l) r))) ; into list, until
     ((null l) r))) ; no more elements.

For a table of related items: See the section "Functions for Modifying Lists".

**rot x y**

Returns *x* rotated left *y* bits if *y* is positive or zero, or *x* rotated right \(|y|\) bits if *y* is negative. The rotation considers *x* as a 32-bit number. *x* and *y* must be fixnums. (There is no function for rotating bignums.)

Examples:

(rot 1 2) => #04
(rot 1 -2) => #10000000000
(rot -1 7) => #0-1
(rot #015 32.) => #015

For a table of related items: See the section "Machine-Dependent Arithmetic Functions".

**rotatef &rest references**

Exchanges two *references*. Each of the *references* can be any form acceptable as a generalized variable to *setf*. All the *references* form an end-around shift register that is rotated one place to the left, with the value of *reference1* being shifted around to *references*. *rotatef* always returns *nil*.

Here is an example as seen in a Lisp Listener:
(setq circus (list 'ringling-brothers 'barnum 'bailey))
  => (RINGLING-BROTHERS BARNUM BAILEY)
(rotatef (first circus) (second circus) (third circus))
  => NIL
  circus
  => (BARNUM BAILEY RINGLING-BROTHERS)

Here is another example as seen in a Lisp Listener:

(setq alpha (list 'able 'baker 'charlie 'dog 'easy 'fox))
  => (ABLE BAKER CHARLIE DOG EASY FOX)
(rotatef (first alpha) (third alpha) (fifth alpha))
  => NIL
  alpha
  => (CHARLIE BAKER EASY DOG ABLE FOX)

Finally:

(setq trio (list 'adam 'eve 'pinch-me-tight))
  => (ADAM EVE PINCH-ME-TIGHT)
(rotatef (first trio) (third trio))
  => NIL
  trio
  => (PINCH-ME-TIGHT EVE ADAM)

See the section "Generalized Variables".

\textbf{round number \&optional (divisor 1)} \\

\textit{Function}

When supplied with one-argument, converts its argument \textit{number} (which must not be complex) to be an integer. If the argument is already an integer, it is returned directly. If the argument is a ratio or floating-point number, \textit{round} converts its argument by rounding to the nearest integer; if \textit{number} is exactly halfway between two integers (that is, of the form integer +0.5), then it is rounded to the one that is even (divisible by two).

The arguments \textit{number} and \textit{divisor} must each be a non-complex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are \textit{Q} and \textit{R}, then \(+\) (* \textit{Q} \textit{divisor}) \textit{R} equals \textit{number}. If \textit{divisor} is 1, then \textit{Q} and \textit{R} add up to \textit{number}. If \textit{divisor} is 1 and \textit{number} is an integer, then the returned values are \textit{number} and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, then the second returned value is always a number of the same type as the argument.

Examples:

\(\text{(round 5)} \Rightarrow 5 \text{ and } 0\)
(round -5) => -5 and 0
(round 5.2) => 5 and 0.19999981
(round -5.2) => -5 and -0.19999981
(round 5.8) => 6 and -0.19999981
(round -5.8) => -6 and 0.19999981
(round 5 3) => 2 and -1
(round -5 3) => -2 and 1
(round 5 4) => 1 and 1
(round -5 4) => -1 and -1
(round 5.2 3) => 2 and -0.8000002
(round -5.2 3) => -2 and 0.8000002
(round 5.2 4) => 1 and 1.1999998
(round -5.2 4) => -1 and -1.1999998
(round 5.8 3) => 2 and -0.19999981
(round -5.8 3) => -2 and 0.19999981
(round 5.8 4) => 1 and 1.8000002
(round -5.8 4) => -1 and -1.8000002

For a table of related items: See the section "Functions that Divide and Convert Quotient to Integer".

rplaca x y

Function
Changes the car of x to y and returns (the modified) x. x must be a cons or a locative. Note that CLOE does not support locatives. y can be any Lisp object. Example:

(setq z '(e f)) => (E F)
(replaca 'f g) => (G F)

Here is another example:

(setq g '(a b c))
(rplaca (cdr g) 'd) => (d c)
Now g => (a d c)

In the following example, rplaca modifies an association list.

(defun exchange-rank(alist datum1 datum2)
  (let* ((element1 (rassoc datum1 alist))
         (element2 (rassoc datum2 alist))
         (tmprank (car element2)))
    (rplaca element2 (car element1))
    (rplaca element1 tmprank)
    (alist))
=> EXCHANGE-RANK

(setq ranked-list (pairlis '(2 1 3) '(mary jane freda)))
=> ((2 . MARY)(1 . JANE)(3 . FREDA))
Using the `setf` macro with `car` achieves the same effect on the `list` argument as `rplaca`, and is considered preferable except in cases using the returned value.

```
(setf (car list) object) => object
(rplaca list object) => list
```

For a table of related items: See the section "Functions for Modifying Lists".

**rplacd x y**

Changes the cdr of `x` to `y` and returns (the modified) `x`. `x` must be a cons or a locative. `y` can be any Lisp object. Example:

```
(setq x '(a b c))
(rplacd x 'd) => (a . d)
Now x => (a . d)
```

When `x` and `y` are cdr-coded and are at consecutive addresses, `rplacd` returns a cdr-coded list. Otherwise, `rplacd` forwards `x` to a new cons before modifying the cdr. For information on `rplacd`-forwarding: See the section "Cdr-Coding". The following usually returns a cdr-coded list:

```
(rplacd (list 'a) (list 'b))
```

In the following example, `rplacd` modifies an association list and returns two values, the two exchanged items. Because `setf` does not directly return the values we desire, we use `rplacd` instead of `setf` of `cdr`

```
(defun exchange-name( alist key1 key2 )
  (let* ((element1 (assoc key1 alist))
         (element2 (assoc key2 alist))
         (tmpname (cdr element2)))
    (values (rplacd element2 (cdr element1))
            (rplacd element1 tmpname))))
```

For a table of related items: See the section "Functions for Modifying Lists".

**zl:samepnamelp x y**

Returns `t` if the two symbols `x` and `y` have `string=` print-names, that is, if their printed representation is the same. If either or both of the arguments is a string
instead of a symbol, that string is used in place of the print-name. \texttt{zl:samepnamenp}
is useful for determining if two symbols would be the same except that for being
in different packages. Examples:
\begin{verbatim}(zl:samepnamenp 'xyz (maknam '(x y z))) => t
(zl:samepnamenp 'xyz (maknam '(w x y)) => nil
(zl:samepnamenp 'xyz "xyz") => t\end{verbatim}
This is the same function as \texttt{string=}. \texttt{zl:samepnamenp} is provided mainly for com-
patibility with older dialects of Lisp. In new programs, you just use \texttt{string=}.
See the section "Functions Relating to the Print Name of a Symbol".

\texttt{zl:sassoc} item in-list else \hspace{1cm} \textit{Function}
Looks up \texttt{item} in the association list \texttt{in-list}. Returns the first cons whose car is
\texttt{zl:equal} to \texttt{x}. \texttt{zl:sassoc} could have been defined by:
\begin{verbatim}(defun zl:sassoc (item alist function)
   (or (assoc item alist)
       (apply function nil)))\end{verbatim}
\texttt{zl:sassoc} is of limited use. It is primarily a leftover from earlier implementations
of Lisp.
For a table of related items: See the section "Functions that Operate on Associa-
tion Lists".

\texttt{zl:sassq} item in-list else \hspace{1cm} \textit{Function}
Looks up \texttt{item} in the association list \texttt{in-list}.
The argument \texttt{else} is a function.
\texttt{zl:sassq} returns the first cons whose car is \texttt{eq} to \texttt{x}, or, if none is, calls \texttt{function}
with no arguments. \texttt{zl:sassq} could have been defined by:
\begin{verbatim}(defun zl:sassq (item alist function)
   (or (assq item alist)
       (apply function nil)))\end{verbatim}
\texttt{zl:sassq} is of limited use. It is primarily a leftover from earlier implementations of
Lisp.

\texttt{satisfies} \hspace{1cm} \textit{Type Specifier}

\texttt{sbit} array &rest subscripts \hspace{1cm} \textit{Function}
Returns the element of \texttt{array} selected by the \texttt{subscripts}. The \texttt{subscripts} must be in-
tegers and their number must match the dimensionality of \texttt{array}. \texttt{sbit} is like \texttt{bit},
but for \texttt{sbit}, the array must be a simple array of bits.
(setq foo (make-array '(2 3)
  :adjustable nil
  :element-type 'bit
  :initial-contents '(((1 1 1)
      (1 0 1)))))

(sbit foo 1 1) => 0

Note that the bit-array in the previous example is simple. Therefore, we can use
sbit, which is more efficient than either aref or bit.

For a table of related items: See the section "Arrays of Bits".

scale-float float integer
  Function

Computes and returns (* float 2^integer).

Although the same result can be obtained by using exponentiation and multiplication,
the use of scale-float can be much more efficient and avoids the intermediate
overflow and underflow if the final result is representable.

Examples:
  (scale-float .5 2) => 2.0
  (scale-float .5 3) => 4.0
  (scale-float .5 4) => 8.0
  (scale-float .75 2) => 3.0

For a table of related items, see the section "Functions that Decompose and Con-
struct Floating-point Numbers".

schar string index
  Function

Returns the character at position index of string. The count is from zero. The
character is returned as a character object.

string must be a string.

index must be a non-negative integer less than the length of string.

Note that the array-specific function aref and the general sequence function elt al-
so work on strings.

To destructively replace a character within a string, use schar in conjunction with
the generic function setf.

  (schar "a string" 0) => #\a
  (string-char-p (schar "a string" 3)) => T

  (schar "a string" 1) => #\Space
(schar (make-array 4 :element-type 'character :initial-element #\y) 3) => #\y
(string-char-p (schar (make-array 4 :element-type 'character :initial-element #\. 2)) => T
(string-char-p (schar (make-array 4 :element-type 'character :initial-element #\.:fill-pointer 2) 1)) => T

(defvar a-simple-string
  (make-array 10
    :element-type 'string-char
    :initial-element #\a))
=> "aaaaaaaaaa"

(schar a-simple-string 0) => #\a
(setf (schar a-simple-string 1) #\b) => #\b

For a table of related items: See the section "String Access and Information".

search sequence1 sequence2 &key :from-end (:test eql) :test-not :key (:start1 0) (:start2 0) :end1 :end2

Function
Looks for a subsequence of sequence2 that element-wise matches sequence1. If no such subsequence exists, search returns nil. If such a subsequence exists, search returns the index into sequence2 of the leftmost element of the leftmost such matching subsequence. sequence1 and sequence2 can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.
If the value of the :from-end keyword is non-nil, the index of the leftmost element of the rightmost matching subsequence is returned. For example:

(search '(1 2) '(3 4 1 2 6 1 2 5)) => 2

(search '(1 2) '(3 4 1 2 6 1 2 5) :from-end t) => 5
:test specifies the test to be performed. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is true. Where testfun is the test function specified by :test, keyfn is the function specified by :key and x is an element of the sequence. The default test is eql.
:test-not is similar to :test, except that the sense of the test is inverted. An element of sequence satisfies the test if (funcall testfun item (keyfn x)) is false.
For example:

```lisp
(search '(2) '(1 2 2 3) :test-not #'>) => 1
```

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

The keyword arguments :start1, :end1, :start2, and :end2 are used to specify sub-sequences for each separate sequence.

:start1 and :end1 must be non-negative integer indices into the sequence. :start must be less than or equal to :end1, else an error is signalled. It defaults to zero (the start of the sequence).

:start1 indicates the start position for the operation within the sequence. :end1 indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence). If both :start1 and :end1 are omitted, the entire sequence is processed by default.

:start2 and :end2 operate the same as :start1 and :end1.

For example:

```lisp
(search #(a b) #(a b c d a b) :start2 3) => 4
```

```lisp
(search #(1 2 3) #(1 2 3 1 2 3 1 2 3) :start1 2 :start2 4) => 5
```

```lisp
(search #(1 2 3) #(1 2 3 1 2 3 1 2 3) :start1 2 :end1 3 :start2 4 :end2 9) => 5
```

```lisp
(search "of" "string of text") => 7
```

For a table of related items: See the section "Searching for Sequence Items".

### second list

**Function**

Takes a list as an argument, and returns the second element of the list. second is identical to cadr. It is also identical to:

`(nth 1 list)`

For example:

```lisp
(setq letters '(a b c d)) =>
(A B C D)
```

```lisp
(second letters) =>
B
```

This function is provided because it makes more sense when you are thinking of the argument as a list rather than just as a cons.

For a table of related items: See the section "Functions for Extracting from Lists".
**select** test-object &body clauses

A conditional that chooses one of its clauses to execute by comparing the value of
a form against various other forms. Its form is as follows:

```
(select key-form
  (test consequent consequent ...)
  (test consequent consequent ...)
  (test consequent consequent ...)
  ...
)
```

The first thing **select** does is to evaluate *key-form*; call the resulting value *key*. Then **select** considers each of the clauses in turn. If *key* matches the clause’s *test*,
the consequents of this clause are evaluated, and **select** returns the value of the
last consequent. If there are no matches, **select** returns **nil**.

A *test* can be any of the following:

- **A symbol** If the *key* is **eq** to the symbol, it matches.
- **A number** If the *key* is **eq** to the number, it matches. Only small num-
  bers (**integers**) work.
- **A list** If the *key* is **eq** to one of the elements of the list, then it
  matches. The elements of the list should be symbols or inte-
 gers.
- **t or otherwise** The symbols **t** and **otherwise** are special keywords that match
  anything. Either symbol can be used; **t** is mainly for compati-
  bility with Maclisp’s **caseq** construct. To be useful, this should
  be the last clause in the **select**.

**select** is the same as **zl:selectq**, except that the test elements are evaluated before
they are used.

This creates a syntactic ambiguity: if *(bar baz)* is seen the first element of a
clause, is it a list of two forms, or is it one form? **select** interprets it as a list of
two forms. If you want to have a clause whose test is a single form, and that form
is a list, you have to write it as a list of one form.

**Examples:**

```
(select (+ 1 2)
  ("four"  "four")
  ((5 6 7) "five six seven")
  (3 "three")
  (t "drop out") => "three"
```

**Where**

```
(select (frob x)
  (foo 1)
  ((bar baz) 2)
  (((current-frob)) 4)
  (otherwise 3))
```
is equivalent to:

```lisp
(let ((var (frob x)))
  (cond ((eq var foo) 1)
        ((or (eq var bar) (eq var baz)) 2)
        ((eq var (current-frob)) 4)
        (t 3)))
```

For a table of related items: See the section "Conditional Functions".

**selector** test-object test-function &body clauses

Special Form

A conditional that chooses one of its clauses to execute by comparing the value of a form against various constants, which are typically keyword symbols. Its form is as follows:

```lisp
(selector key-form test-function
  (test consequent consequent ...)
  (test consequent consequent ...)
  (test consequent consequent ...)
  ...)
```

The first thing selector does is to evaluate key-form; call the resulting value key. Then selector considers each of the clauses in turn. If test-function applied to key satisfies the clause’s test, the consequents of this clause are evaluated, and selector returns the value of the last consequent. If no clause is satisfied, selector returns nil.

test can be a symbol, a number, or a list whose elements are symbols or numbers. In place of a test selector also accepts a t or otherwise clause. t is mainly for compatibility with Maclisp’s caseq construct. To be useful, this should be the last clause in the selector.

test-function can be any user-specified function.

selector is the same as select, except that you get to specify the function used for the comparison instead of eq.

Examples:

```lisp
(let ((arg -14))
  (selector (abs arg) >
    (10 "greater than 10")
    (1 "greater than 1"))) => "greater than 10"
```

Where

```lisp
(selector (frob x) equal
  (('one . two)) (frob-one x))
  (('three . four)) (frob-three x))
  (otherwise (frob-any x)))
```

is equivalent to:
(let ((var (frob x)))
  (cond ((equal var '(one . two)) (frob-one x))
        ((equal var '(three . four)) (frob-three x))
        (t (frob-any x))))

For a table of related items: See the section "Conditional Functions".

**zl:selectq** test-object &body clauses  

*Special Form*

A conditional that chooses one of its clauses to execute by comparing the value of a form against various constants, which are typically keyword symbols. Its form is as follows:

(zl:selectq key-form
  (test consequent consequent ...)
  (test consequent consequent ...)
  (test consequent consequent ...)
  ...
)

The first thing zl:selectq does is to evaluate *key-form*; call the resulting value *key*. Then zl:selectq considers each of the clauses in turn. If *key* matches the clause’s test, the consequents of this clause are evaluated, and zl:selectq returns the value of the last consequent. If there are no matches, zl:selectq returns nil.

A test can be any of the following:

- A symbol If the *key* is eq to the symbol, it matches.
- A number If the *key* is eq to the number, it matches. Only small numbers (integers) work.
- A list If the *key* is eq to one of the elements of the list, then it matches. The elements of the list should be symbols or integers.
- t or otherwise The symbols t and otherwise are special keywords that match anything. Either symbol can be used; t is mainly for compatibility with Maclisp’s caseq construct. To be useful, this should be the last clause in the zl:selectq.

Note that the test elements are not evaluated; if you want them to be evaluated, use select rather than zl:selectq.

Examples:

(let ((voice 'tenor))
  (zl:selectq voice
    (bass "Barber of Seville")
    (Mezzo "Carmen"))) => NIL
(setq a 2) => 2
(zl:selectq a
 (1 "one")
 (2 "two")
 ((one two) "1 2")
 (otherwise "not one or two") => "two"
(let ((a 'big-bang))
 (zl:selectq a
 (light "day")
 (dark "night")
 (t "night and day"))) => "night and day"

Where
(let ((x 'Bird))
 (zl:selectq x
 (foo (do-this))
 (bar (do-that))
 ((baz quux mum) (do-the-other-thing))
 (otherwise (zl:ferror nil "Hey there, never heard of ~S~ x))))
=> Error: Hey there, never heard of BIRD

is equivalent to:
(let ((x 'Bird))
 (cond ((eq x 'foo) (do-this))
       ((eq x 'bar) (do-that))
       ((zl:memq x '(baz quux mum)) (do-the-other-thing))
       (t (zl:ferror nil "Hey there, never heard of ~S~ x))))
=> Error: Hey there, never heard of BIRD

For a table of related items: See the section "Conditional Functions".

**selectq-every**  

A conditional that chooses one of its clauses to execute by comparing the value of a form against various constants, which are typically keyword symbols. Its form is as follows:

```
(selectq-every key-form
 (test consequent consequent ...)
 (test consequent consequent ...)
 (test consequent consequent ...)
 ...)
```

The first thing **selectq-every** does is to evaluate **key-form**; call the resulting value **key**. Then **selectq-every** considers each of the clauses in turn. If **key** matches the clause's **test**, the consequents of this clause are evaluated, and **selectq-every** returns the value of the last consequent. If there are no matches, **selectq-every** returns **nil**.

A **test** can be any of the following:
A symbol If the key is eq to the symbol, it matches.

A number If the key is eq to the number, it matches. Only small numbers (integers) work.

A list If the key is eq to one of the elements of the list, then it matches. The elements of the list should be symbols or integers.

t or otherwise The symbols t and otherwise are special keywords that match anything. Either symbol can be used; t is mainly for compatibility with Maclisp's caseq construct. To be useful, this should be the last clause in the zl:selectq.

selectq-every is like zl:selectq, but like cond-every, selectq-every executes every selected clause, instead of just the first one. If an otherwise clause is present, it is selected if and only if no preceding clause is selected. The value returned is the value of the last form in the last selected clause. Multiple values are not returned.

Note that the test elements are not evaluated.

Examples:

(let ((book 'Lisp))
  (selectq-every book
    ((mystery fantasy science-fiction) (setq type 'fun))
    ((Lisp Pascal Fortran APL) (setq type 'Languages))
    ((Lisp History Math) (setq school 'homework))
    (otherwise (setq type 'unknown)))) => HOMEWORK

(type => LANGUAGES

(selectq-every animal
  ((cat dog) (setq legs 4))
  ((bird man) (setq legs 2))
  ((cat bird) (put-in-oven animal))
  ((cat dog man) (beware-of animal)))

For a table of related items: See the section "Conditional Functions".

self Variable

When a generic function is called on an object, the variable self is automatically bound to that object. This enables the methods to lexically manipulate the object itself (as opposed to its instance variables).

Note that since the compiler has a special way of dealing with variables named self, users should not name arguments or variables self.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

send object message-name &rest arguments Function
Sends the message named \texttt{message-name} to the \texttt{object}. \texttt{arguments} are the arguments passed. \texttt{send} does exactly the same thing as \texttt{funcall}. For stylistic reasons, it is preferable to use \texttt{send} instead of \texttt{funcall} when sending messages because \texttt{send} clarifies the programmer’s intent.

\begin{verbatim}
(send some-window :set-edges 10 10 40 40)
\end{verbatim}

\texttt{send} is supported for compatibility with previous versions of the flavor system. When writing new programs, it is good practice to use \texttt{funcall} instead of message-passing. See the section "Generic Functions".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

\begin{verbatim}
:send-if-handles message &rest arguments
\end{verbatim}

The object that receives this message performs the operation indicated by \texttt{message} with the given \texttt{arguments}, if it has a method for the operation. If no method for the operation is available, \texttt{nil} is returned.

\texttt{message} is a message name or a generic function object, such as the result of evaluating the form \texttt{(flavor:generic generic-function-name)}. \texttt{arguments} are the arguments for the operation.

For example:

\begin{verbatim}
;;; using :send-if-handles with a message
(send *cell-instance* :send-if-handles :describe)

;;; using :send-if-handles with a generic function
(send *cell-instance* :send-if-handles (flavor:generic aliveness))
\end{verbatim}

\texttt{flavor:vanilla} provides a method for \texttt{:send-if-handles}.

Instead of sending this message, you can use the \texttt{:send-if-handles} function. For information on restrictions in using \texttt{:send-if-handles} with generic functions, see the function \texttt{send-if-handles}.

Note that \texttt{:send-if-handles} works by sending the \texttt{:send-if-handles} message. You can customize the behavior of \texttt{:send-if-handles} by defining a method for the \texttt{:send-if-handles} message.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

\begin{verbatim}
send-if-handles object message &rest arguments
\end{verbatim}

The \texttt{object} performs the operation indicated by \texttt{message} with the given \texttt{arguments}, if it has a method for the operation. If no method for the operation is available, \texttt{nil} is returned.

\texttt{object} is a Lisp object, usually a flavor instance. \texttt{message} is a message name or a generic function object, such as the result of evaluating the form \texttt{(flavor:generic generic-function-name)}. \texttt{arguments} are the arguments for the operation.
For example:

```lisp
;;; using send-if-handles with a message
(send-if-handles *cell-instance* :describe)

;;; using send-if-handles with a generic function
(send-if-handles *cell-instance* (flavor:generic aliveness))
```

Note that `send-if-handles` works by sending the `:send-if-handles` message. You can customize the behavior of `send-if-handles` by defining a method for the `:send-if-handles` message.

Note that `send-if-handles`, `:send-if-handles`, and `lexpr-send-if-handles` were originally designed to work in the message-passing paradigm, and their use does not fit cleanly into the generic function paradigm. Any generic function that uses the `:function`, `:dispatch`, or `:compatible-message` option for `defgeneric`, or that uses the `flavor:solitary-method` declaration in `defmethod`, will not work as expected with these operations.

Instead of using these operations with generic functions, we suggest avoiding the need for the caller to test whether the generic function is handled before calling it, by ensuring that the generic function works for all arguments without signalling the `sys:unclaimed-message` error. For example, you could provide default handling by using the `:function` option to `defgeneric`, or by defining a method on a very general flavor.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**sequence** &optional (type '*)

`sequence` is the type specifier symbol for the predefined Lisp structure of that name.

The type `sequence` is a supertype of the types `vector` and `list`. These two types are an exhaustive partition of the type `sequence`.

In addition to a symbol form, Symbolics Common Lisp provides a list form for `sequence`. Used in list form, `sequence` defines the set of sequences whose elements are of type `type`. `type` must be one of the standard data types. The list form might not work in other implementations of Common Lisp. For standard Symbolics Common Lisp type specifiers, see the section "Type Specifiers".

Examples:

```lisp
(typep '(a b c d e) 'sequence) => T
(typep '(mom 25 dad 28) '(sequence list)) => T
(subtypep 'list 'sequence) => T and T
(subtypep 'vector 'sequence) => T and T
(sys:type-arglist 'sequence) => (&OPTIONAL (TYPE '*)) and T
```

See the section "Data Types and Type Specifiers". See the section "Sequences".
**set symbol value**

The primitive for assignment of a value to a dynamic (special) variable. The symbol's value is changed to value; value can be any Lisp object. set only changes the value of the current dynamic binding. If symbol has no current binding in effect, its most global value is changed. set returns value. Example:

```
(set (cond ((eq a b) 'c)
          (t 'd))
     'foo)
```
either sets c to foo or sets d to foo.

set does not work on local (lexically bound) variables.

```
(proclaim '(special *foo*)
*foo*
(TERMINAL-IO LISP)
(let ((*foo* '(foo lisp)))
  (set '*foo* (cons 'bar *foo*)))
  (print *foo*)
(nil))
(BAR FOO LISP)
NIL
*foo*
(TERMINAL-IO LISP)
(set *foo* (cons 'bar *foo*))
(BAR TERMINAL-IO LISP)
```
See the section "Functions Relating to the Value of a Symbol".

**set-char-bit char name value**

Changes the bit named name in char and returns the new character. value is nil to clear the bit or non-nil to set it.

```
(set-char-bit #\A :meta T) => #\m-A
(set-char-bit #\h-c-A :control NIL) => #\h-A
(setq char #\0)
(char-bit (set-char-bit char :control t) :control) => T
(char-bit char) => nil
```
For a table of related items, see the section "Making a Character".

**set-character-translation from-char to-char &optional readable**

Changes readable so that from-char is translated to to-char when read in, when readable is the current readtable. This is normally used only for translating lowercase letters to uppercase. Character translations are turned off by slash, string quotes, and vertical bars. readable defaults to the current readtable.
:set-cursorpos x y &optional (units :pixel)  
Message
This operation is supported by the same streams that support :read-cursorpos. It sets the position of the cursor. x and y are the new coordinates of the cursor and units is the same as the units argument to :read-cursorpos.

set-difference list1 list2 &key (test #eql) test-not (key #identity)  
Function
Returns a list of elements of list1 that do not appear in list2. Does not change the arguments. Note that there is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The keywords are:

:test  
Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not  
Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key  
If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

For all possible ordered pairs consisting of one element from list1 and one element from list2, the predicate is used to determine whether they match. An element of list1 appears in the result if and only if it does not match any element of list2. For example:

(setq a-list '(eagle hawk loon pelican)) =>
(EAGLE HAWK LOON PELICAN)

(setq b-list '(owl hawk stork)) => (OWL HAWK STORK)

(set-difference a-list b-list) => (EAGLE LOON PELICAN)

(set-difference b-list a-list) => (OWL STORK)

You can use set-difference to do things such as removing from a list of strings all of those strings containing one of a given list of characters. In this example, we remove all flavor names that contain the characters "c" or "w".

(set-difference '("strawberry" "chocolate" "banana" "lemon"
  "pistachio" "rhubarb") '#("c" "w")
  :test #'(lambda (s c) (find c s))) =>
("banana" "lemon" "rhubarb")

In the following example, set-difference returns the list of lists of all tenured professors who are not new.
(setq professors-with-tenure
  '((("Jones" CS101 CS242) ("smith" CS202 CS231)
    ("parks" CS221) ("hunter" CS216 CS232)))
(setq new-professors
  '((("Able" CS101 CS244) ("Cain" CS101 CS331)
    ("Parks" CS221) ("adams" CS215 CS222)))
(set-difference professors-with-tenure new-professors
  :test #'string-equal :key #'car)
=>
  (("Jones" CS201 CS242) ("smith" CS202 CS231)
 ("hunter" CS216 CS232))

For a table of related items: See the section “Functions for Comparing Lists”.

**set-dispatch-macro-character disp-char sub-char function &optional (a-readtable *readtable*)**

Causes function to be called when the disp-char followed by sub-char is read. function is called with three arguments, a stream, sub-char, and the non-negative integer whose decimal representation appears between disp-char and sub-char, or nil if no decimal integer appeared there. set-dispatch-macro-character returns t.

An error is signalled if sub-char is one of the ten decimal digits, since they are reserved for specifying an infix integer argument. Moreover, if sub-char is a lowercase character, its uppercase equivalent is used instead. This is how the rule is enforced that the case of a dispatch sub-character doesn't matter.

An error is also signalled if the specified disp-char is not a dispatch character in the specified readtable. It is necessary to use make-dispatch-macro-character to set up the dispatch character before specifying its sub-characters.

As an example, the definition of the sharp-sign single-quote dispatch macro character is:

```lisp
(defun sharp-single-quote-reader (stream sub-char arg)
  (declare (ignore char arg))
  (list-in-area 'sys:read-area 'function
    (read stream t nil t)))
(set-dispatch-macro-character #\# #\' #'sharp-single-quote-reader)
```

sharp-single-quote-reader reads an object following the single-quote and returns a list of the symbol function and that object. The char and arg arguments are ignored for this function. Note that the recursive-p argument to read is t, which means that this call to read is imbedded, not top-level.
(let ((*readtable* (copy-readtable nil)))
  (macfun (get-dispatch-macro-character #\# #\#))
  (set-dispatch-macro-character #\# #\Q macfun)
  (values (read-from-string "#Q+")))
=> #\+/j141

(set-exclusive-or list1 list2 &key (test #eql) test-not (key #identity))  Function

Returns a list of elements that appear in exactly one of list1 and list2. Does not change the arguments. Note that there is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The keywords are:

:test Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

For all possible ordered pairs consisting of one element from list1 and one element from list2, the predicate is used to determine whether they match. The result contains precisely those elements of list1 and list2 which appear in no matching pair. For example:

(setq a-list '((eagle hawk loon pelican)) =>
(EAGLE HAWK LOON PELICAN)

(setq b-list '((owl hawk stork)) => (OWL HAWK STORK)

(set-exclusive-or a-list b-list) => (EAGLE LOON PELICAN OWL STORK)

In the following example, > is the test. Each element of list-a is considered an element of list-b, in case it is greater than some element of list-b, and vice versa for the elements of list-b in relation to those of list-a. Thus, set-exclusive-or with :test > returns a list of the elements of one set, all of which are less than any element of the other set.

(setq list-a '(23 12 17 10))
(setq list-b '(42 16 31))

(set-exclusive-or list-a list-b :test #>) => (12 10)

For a table of related items: See the section "Functions for Comparing Lists".
z!:set-globally var value  

Works like set but sets the global value regardless of any bindings currently in effect.

z!:set-globally operates on the global value of a special variable; it bypasses any bindings of the variable in the current stack group. It resides in the global package.

z!:set-globally does not work on local variables.

See the section "Functions Relating to the Value of a Symbol".

z!:set-in-closure closure symbol value  

Sets the binding of symbol in the environment of closure to value; that is, it does what would happen if you restored the value cells known about by closure and then set symbol to value. This allows you to change the contents of the value cells known about by a dynamic closure. If symbol is not closed over by closure, this is just like set. See the section "Dynamic Closure-Manipulating Functions".

z!:set-in-instance instance symbol value  

Alters the value of an instance variable inside a particular instance, regardless of whether the instance variable was declared a :writable-instance-variable or a :settable-instance-variable. instance is the instance to be altered, symbol is the instance variable whose value should be set, and value is the new value. If there is no such instance variable, an error is signalled.

In Symbolics Common Lisp, this operation is performed by:

```
(setq (scl:symbol-value-in-instance instance symbol) value)
```

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

:set-input-interrupt-function function &rest args  

Assigns a function to be applied to any args whenever input becomes available on the connection, or the connection goes into an unusable state. The function is called in a non-simple process, and therefore can use :process-wait.

set-macro-character char function &optional non-terminating-p (a-readtable *readtable*)  

Causes char to be a macro character that causes function to be called when it is seen by the reader. If non-terminating-p is not nil (it defaults to nil), it will be a non-terminating macro character, which means that it may be embedded within extended tokens. set-macro-character returns t.
function is called with two arguments, stream and char. stream is the input stream, and char is the macro character itself. In the simplest case, function returns a Lisp object. This object is taken to be that whose printed representation was the macro character and any following characters read by the function. As an example, the definition of the single-quote macro character is:

```lisp
(defun single-quote-reader (stream char)
  (declare (ignore char))
  (list-in-area 'sys:read-area 'quote (read stream t nil t)))

(set-macro-character #'single-quote-reader)
```

single-quote-reader reads an object following the single-quote and returns a list of the symbol quote and that object. The char argument is ignored for this function. Note that the recursive-p argument to read is t, which means that this call to read is embedded, not top-level.

function should not have any side effects other than on stream. Because of backtracking and restarting of the read operation, front ends to the reader, such as editors and rubout handlers, can cause function to be called repeatedly during the reading of a single expression in which the macro character only appears once.

In the following example, square brackets are given a reader syntax which uses them to denote vectors.

```lisp
(defvar *square-bracket-depth* 0)

(defun square-bracket-vector-reader (stream char)
  (if (and (= *square-bracket-depth* 0) (char= char #\[]))
    (progn
      (set-syntax-from-char #\] #\[])
      (set-macro-character #\] #'square-bracket-vector-reader))
    (incf *square-bracket-depth*)
    (do ((result '())
     ((char= (peek-char t stream nil #\] t) #\])
     (if (= *square-bracket-depth* 0)
       (if result result (values))
       (progn
         (read-char stream)
         (defc *square-bracket-depth*)
         (coerce (nreverse result) 'vector))))
    (push (read stream t nil t) result)))
```
(let ((*readtable* (copy-readtable))
  (str "123 foobar [12 34 [56 78] 9] foobar")
  (result '()))
(set-macro-character #\[ #'square-bracket-vector-reader)
(with-input-from-string (stream str)
  (dotimes (i 4) (push (read stream) result)))
(set-syntax-from-char #\[
 \_]
(set-syntax-from-char #\]
 \_)
(nreverse result))
=> (123 FOOBAR #(12 34 #(56 78) 9) FOOBAR)

:set-pointer new-pointer
Message
Sets the reading position within the file to new-pointer (bytes in fixnum mode). For text files on PDP-10 file servers, this does not do anything reasonable unless new-pointer is 0, because of character-set translation.
See the section "Direct Access Output File Streams".
See the section "Direct Access Bidirectional File Streams".

dbg:set-proceed-types condition new-proceed-types
Generic Function
Sets the list of valid proceed types for this condition to new-proceed-types.
The compatible message for dbg:set-proceed-types is:

:set-proceed-types
For a table of related items, see the section "Basic Condition Methods and Init Options".

zl:set-syntax-#-macro-char char function &optional readtable
Function
Causes function to be called when #char is read. readtable defaults to the current readable. The function’s arguments and return values are the same as for normal macro characters. When function is called, the special variable si:xr-sharp-argument contains nil or a number that is the number or special bits between the # and char.

set-syntax-from-char to-char from-char &optional (to-readable *readtable*) from-readable
Function
This makes the syntax of to-char in to-readable be the same as the syntax of from-char in from-readable. The to-readable defaults to the current readable (the value of the global variable *readtable*), and from-readable defaults to nil, meaning to use the syntaxes from the standard Lisp readable.
The attributes whitespace, constituent, macro and escape are copied. If a macro character is copied, the macro definition is also copied. The attributes alphabetic and alphadigit, as well as marker characteristics such as plus sign, dot and float exponent marker, are not copied, since they are "hard-wired" into the extended-token parser. For example, if the definition of s is copied to *, * will become a constituent that is alphabetic but cannot be used as an exponent indicator for short-format floating-point number syntax.

You can copy a macro definition from a character such as " to another character and expect it to work properly, since the standard definition for " looks for another character that is the same as the character that invoked it. You probably don't want to copy the definition of ( to {, since it lets you write lists in the form {a b c}, not {a b c}, because the definition always looks for a closing parenthesis, not a closing brace.

```lisp
(let* ((foo "%zzz%zzz"))
  (newrt (copy-readtable))
  (*readtable* newrt)
  (result '()))
(push (read-from-string foo) result)
(set-syntax-from-char #\% #\")
(push (read-from-string foo) result)
(set-syntax-from-char #\% #\())
(push (read-from-string foo) result)
(nreverse result))
```

=> (%ZZZ%ZZZ "zzz" (ZZZ (ZZZ)))

**zl: set-syntax-from-char**

<table>
<thead>
<tr>
<th>to-char</th>
<th>from-char</th>
<th>&amp;optional to-readtable from-readtable</th>
</tr>
</thead>
</table>

Function

Makes the syntax of to-char in to-readtable be the same as the syntax of from-char in from-readtable. to-readtable defaults to the current readtable, and from-readtable defaults to the initial standard readtable.

**zl: set-syntax-from-description**

<table>
<thead>
<tr>
<th>char</th>
<th>description</th>
<th>&amp;optional readtable</th>
</tr>
</thead>
</table>

Function

Sets the syntax of char in readable to be that described by the symbol description. The following descriptions are defined in the standard readable:

- **si: alphabetic** An ordinary character such as "a".
- **zl: break** A token separator such as "(". (Of course, left parenthesis has other properties besides being a break.)
- **si: whitespace** A token separator that can be ignored, such as "@".
- **si: single** A self-delimiting single-character symbol. The initial readable does not contain any of these.
si:slash  The character quoter. In the initial readtable this is "/".

si:verticalbar  The symbol print-name quoter. In the initial readtable this is "|".

si:doublequote  The string quoter. In the initial readtable this is """.

macro  A macro character. Do not use this; use zl:set-syntax-macro-char.

si:circlecross  The octal escape for special characters. In the initial readtable this is "⊗". (si:circlecross exists only in the standard Zetalisp readtable, not the Symbolics Common Lisp readtable.)

si:bitscale  A character that causes the integer to its left to be doubled the number of times indicated by the integer to its right. In the initial readtable this is "_". See the section "What the Reader Recognizes".

si:digityscale  A character that causes the integer to its left to be multiplied by zl:ibase the number of times indicated by the integer to its right. In the initial readtable this is "^". See the section "What the Reader Recognizes".

si:non-terminating-macro

A macro character that is not a token separator. This is a macro character if seen alone but is just a symbol constituent inside a symbol. You can use it as a character of a symbol other than the first without slashing it. (# would be one of these if it were not built into the reader.)

readtable defaults to the current readtable.

zl:set-syntax-macro-char  char  function  &optional  readtable  non-terminating-p

Function

Changes readtable so that char is a macro character. When char is read, function is called. readtable defaults to the current readtable.

function is called with two arguments: list-so-far and the input stream. When a list is being read, list-so-far is that list (nil if this is the first element). At the "top level" of zl:read, list-so-far is the symbol :toplevel. After a dotted-pair dot, list-so-far is the symbol :after-dot. function can read any number of characters from the input stream and process them however it likes.

function should return three values, called thing, type, and splice-p. thing is the object read. If splice-p is nil, thing is the result. If splice-p is non-nil, when reading a list thing replaces the list being read — often it is list-so-far with something else nconc'ed onto the end. At top level and after a dot if splice-p is non-nil the thing is ignored and the macro character does not contribute anything to the result of zl:read. type is a historical artifact and is not really used; nil is a safe value. Most macro character functions return just one value and let the other two default to nil.
function should not have any side effects other than on the stream and list-so-far. Because of the way the input editor works, function can be called several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called :macro syntax).

If non-terminating-p is nil (the default), zl:set-syntax-macro-char makes a normal macro character. If it is t, zl:set-syntax-macro-char makes a nonterminating macro character. A nonterminating macro character is a character that acts as a reader macro if seen between tokens, but if seen inside a token it acts as an ordinary letter; it does not terminate the token.

zl:setarg i x

Function

Used only during the application of a lexpr. (zl:setarg i x) sets the lexpr’s i’th argument to x. i must be greater than zero and not greater than the number of arguments passed to the lexpr. After (zl:setarg i x) has been done, (zl:arg i) returns x.

dl:setarg exists only for compatibility with Maclisp lexprs. To write functions that can accept variable numbers of arguments, use the &optional and &rest keywords. See the section “Evaluating a Function Form”.

setf reference value &rest more-pairs

Macro

Takes a form that accesses something, and “inverts” it to produce a corresponding form to update the thing. A setf expands into an update form, which stores the result of evaluating the form value into the place referenced by the reference. If you supply more than one reference value pair, the pairs are processed sequentially.

The form of reference can be any of the following:

- The name of a variable (either local or global).
- A function call to any of the following functions:

<table>
<thead>
<tr>
<th>aref</th>
<th>car</th>
<th>svref</th>
<th>nth</th>
<th>cdr</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>elt</td>
<td>caar</td>
<td>getf</td>
<td>rest</td>
<td>cadr</td>
<td>gethash</td>
</tr>
<tr>
<td>first</td>
<td>cdar</td>
<td>symbol-value</td>
<td>second</td>
<td>cddr</td>
<td>symbol-function</td>
</tr>
<tr>
<td>third</td>
<td>caaar</td>
<td>fill-pointer</td>
<td>fourth</td>
<td>caadr</td>
<td>symbol-plist</td>
</tr>
<tr>
<td>fifth</td>
<td>caadar</td>
<td>macro-function</td>
<td>sixth</td>
<td>cdaadr</td>
<td></td>
</tr>
<tr>
<td>seventh</td>
<td>cdaaar</td>
<td></td>
<td>eighth</td>
<td>cddadr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cdaadr</td>
<td></td>
<td></td>
<td>cddadr</td>
<td></td>
</tr>
</tbody>
</table>
A function call whose first element is the name of a selector function created by `defstruct`.

A function call to one of the following functions paired with a value of the specified type so that it can be used to replace the specified "place":

<table>
<thead>
<tr>
<th>Function name</th>
<th>Required type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>string-char</td>
</tr>
<tr>
<td>schar</td>
<td>string-char</td>
</tr>
<tr>
<td>bit</td>
<td>bit</td>
</tr>
<tr>
<td>sbit</td>
<td>bit</td>
</tr>
<tr>
<td>subseq</td>
<td>sequence</td>
</tr>
</tbody>
</table>

In the case of `subseq`, the replacement value must be a sequence whose elements can be contained by the sequence argument to `subseq`. If the length of the replacement value does not equal the length of the subsequence to be replaced, then the shorter length determines the number of elements to be stored. See the function `replace`.

A function call to any of the following functions with an argument to that function in turn being a "place" form. The result of applying the specified update function is then stored back into this new place.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Argument that is a place</th>
<th>Update function used</th>
</tr>
</thead>
<tbody>
<tr>
<td>char-bit</td>
<td>first</td>
<td>set-char-bit</td>
</tr>
<tr>
<td>ldb</td>
<td>second</td>
<td>dpb</td>
</tr>
<tr>
<td>mask-field</td>
<td>second</td>
<td>deposit-field</td>
</tr>
</tbody>
</table>

A the type declaration form, in which case the declaration is transferred to the value form and the resulting `setf` form is analyzed. For example,

```
(setf (the integer (cadr x)) (+ y 3))
```

is processed as if it were

```
(setf (cadr x) (the integer (+ y 3)))
```

See the section "Generalized Variables".

For a table of related items: See the section "Basic Array Functions".

```
future-common-lisp:setf reference value &rest more-pairs
```

Macro

Expands the same as does `setf`. Calling `future-common-lisp:setf` has the same effect as calling `setf`.

---

ninth cddar caddar cdddar
tenth cdddr cadddr cdddddar
Because the argument order in defining setf methods and generic functions is different in CLOS and Flavors, the two symbols `setf` and `future-common-lisp:setf` are used in function specs for setf generic functions, to indicate which argument order is being used. The Flavors lambda-lists have the `new-value` parameter last, preceded by other arguments. The CLOS lambda-lists have the `new-value` parameter first, followed by other arguments.

```lisp
;;; Flavors
(defun (setf symbol) (instance args... new-value)
  (body))

(defun (setf symbol flavor) (args... new-value)
  (body))

;;; CLOS
(defun (future-common-lisp:setf symbol) (new-value instance args...)
  (body))

(defun (future-common-lisp:setf symbol) (new-value (instance class) args...)
  (body))
```

The symbols `setf` and `future-common-lisp:setf` are used in function specs for setf generic functions, to indicate which argument order is being used.

The `:writable-instance-variables` option to `defflavor` creates a method for a generic function whose function spec is of the form: `(setf symbol)`. The `:accessor` option to `clos:defclass` creates a method for a generic function whose function specs are of the form: `(future-common-lisp:setf symbol)`. For reasons of flexibility, it is possible to use either `future-common-lisp:setf` or `setf` with both the Flavors and CLOS forms of `defmethod` and `defgeneric`. By convention, however, Flavors programs use the Flavors argument order and create function specs with `setf`; CLOS programs use the CLOS argument order and create function specs with `future-common-lisp:setf`.

### zl:setf access-form value

Macro

Takes a form that accesses something, and "inverts" it to produce a corresponding form to update the thing. A `zl:setf` expands into an update form, which stores the result of evaluating the form `value` into the place referenced by the `access-form`. Examples:

```lisp
(zl:setf (array-leader foo 3) 'bar)
==> (store-array-leader 'bar foo 3)
(zl:setf a 3) ==> (setq a 3)
(zl:setf (plist 'a) '(foo bar)) ==> (setplist 'a '(foo bar))
(zl:setf (aref q 2) 56) ==> (aset 56 q 2)
(zl:setf (cadr w) x) ==> (rplaca (cdr w) x)
```
If `access-form` invokes a macro or a substitutable function, `zl:setf` expands the `access-form` and starts over again. This lets you use `zl:setf` together with `zl:defstruct` accessors.

For the sake of efficiency, the code produced by `zl:setf` does not preserve order of evaluation of the argument forms. This is only a problem if the argument forms have interacting side effects. For example, if you evaluate:

```lisp
  (setq x 3)
  (setf (aref a x) (setq x 4))
```

the form might set element 3 or element 4 of the array. We do not guarantee which one it will do; do not just try it and see and then depend on it, because it is subject to change without notice.

Furthermore, the value produced by `zl:setf` depends on the structure type and is not guaranteed; `zl:setf` should be used for side effect only. If you want well-defined semantics, you can use `setf` in your Symbolics Common Lisp programs.

See the section "Generalized Variables".

A generalization of variable assignment, this macro allows the update of a wide variety of storage locations, such as structure components, vector elements, or elements of a list. With `place` as a selector function, `psetf` uses the update form appropriate to the selector form to change the value at the accessed location to `newvalue`. The `place/newvalue` pairs are processed in order from left to right.

```lisp
  (setf a '(1 2 3))  is equivalent to  (setq a '(1 2 3))
  a → (1 2 3)

  (setf (cddr a) '(buckle my shoe))
    is equivalent to  (progn (rplacd (cdr a) '(buckle my shoe)) (cddr a))
  a → (1 2 buckle my shoe)
```

A large number of `place` forms are predefined, (see CLtL pages 94-97), and additions can be made via `defsetf` or `define-setf-method`.

See Also: CLtL 94, `psetf`, `defsetf`, `define-setf-method`

---

**zl:setplist** symbol list 

*Function*

Sets the list that represents the property list of `symbol` to `list`. Use `zl:setplist` with extreme caution, since property lists sometimes contain internal system properties, which are used by many useful system functions. Also, it is inadvisable to have the property lists of two different symbols be `eq`, since the shared list structure causes unexpected effects on one symbol if `zl:putprop` or `remprop` is done to the other.

See the section "Functions Relating to the Property List of a Symbol".
setq &rest vars-and-vals

Special Form

Used to set the value of one or more variables. The first variable is evaluated, and the first value is set to the result. Then the second variable is evaluated, the second value is set to the result, and so on for all the variable/value pairs. setq returns the last value, that is, the result of the evaluation of its last subform. Example:

(setq x (+ 3 2 1) y (cons x nil))

x is set to 6, y is set to (6), and the setq form returns (6). Note that the first variable was set before the second value form was evaluated, allowing that form to use the new value of x.

This function is acceptable for both special and lexical variables.

(setq a '(1 2 3) b '((4 5) 6) c (cons 0 a))

=> (0 1 2 3)

a => (1 2 3)
b => ((4 5) 6)
c => (0 1 2 3)

zl:setq-globally &rest vars-and-vals

Function

Use the Symbolics Common Lisp function symbol-value-globally instead of this. You use setf with symbol-value-globally to set global values in your init file.


Special Form

Sets the standard value of name to the value of form. If you want to change your default zl:base to 8 (octal), do this:

(zl:setq-standard-value zl:base 8)
(zl:setq-standard-value zl:ibase 8)

zl:setq-standard-value runs the validation function associated with the symbol and signals an error if the validation function fails. You can use only zl:setq-standard-value on symbols defined with sys:defvar-standard. zl:setq-standard-value and zl:setq-globally work with login-forms and are recommended for use in init files where you want your customizations to be undone when you log out.

For programs, zl:setq-standard-value has been superseded by setf of sys:standard-value.

zl:setsyntax character arg2 arg3

Function

Exists only for Maclisp compatibility. The other readtable functions are preferred in new programs. The syntax of character is altered in the current readtable, according to arg2 and arg3. character can be an integer, a symbol, or a string, that is, anything acceptable to the character function. arg2 is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for arg2:
:macro
The character becomes a macro character. arg3 is the name of a function to be invoked when this character is read. The function takes no arguments, can \texttt{zl:tyi} or \texttt{zl:read} from \texttt{zl:standard-input} (that is, can call \texttt{zl:tyi} or \texttt{zl:read} without specifying a stream), and returns an object that is taken as the result of the read.

:splicing
Like :macro, but the object returned by the macro function is a list that is \texttt{nconc}ed into the list being read. If the character is read not inside a list (at top level or after a dotted-pair dot), then it can return \texttt{nil}, which means it is ignored, or \texttt{(obj)}, which means that \texttt{obj} is read.

:single
The character becomes a self-delimiting single-character symbol. If \texttt{arg3} is an integer, the character is translated to that character.

nil
The syntax of the character is not changed, but if \texttt{arg3} is an integer, the character is translated to that character.

a symbol
The syntax of the character is changed to be the same as that of the character \texttt{arg2} in the standard initial readtable. \texttt{arg2} is converted to a character by taking the first character of its print name. Also if \texttt{arg3} is an integer, the character is translated to that character.

\texttt{zl:setsyntax-sharp-macro} character type function \&optional readable \textbf{Function}

Exists only for Maclisp compatibility. \texttt{zl:set-syntax-#-macro-char} is preferred. If \texttt{function} is \texttt{nil}, \texttt{#character} is turned off, otherwise it becomes a macro that calls \texttt{function}. \texttt{type} can be \texttt{:macro}, \texttt{:peek-macro}, \texttt{:splicing}, or \texttt{:peek-splicing} The splicing part controls whether \texttt{function} returns a single object or a list of objects. Specifying peek causes \texttt{character} to remain in the input stream when \texttt{function} is called; this is useful if \texttt{character} is something like a left parenthesis. \texttt{function} gets one argument, which is \texttt{nil} or the number between the \# and the \texttt{character}.

\texttt{seventh list} \textbf{Function}

Takes a list as an argument, and returns the seventh element of the list. \texttt{seventh} is identical to:

\begin{verbatim}
(nth 6 list)
\end{verbatim}

For example:

\begin{verbatim}
(setq letters `(a b c d e f g h i)) =>
(A B C D E F G H I)

(seventh letters) => G
\end{verbatim}

This function is provided because it makes more sense when you are thinking of the argument as a list rather than just as a cons.
For a table of related items: See the section "Functions for Extracting from Lists".

**shadow** symbols &optional package

*Function*
symbols should be a list of symbols or a single symbol. If symbols is nil, it is treated like an empty list. The name of each symbol is extracted, and package is searched for a symbol of that name. If no such symbol is present in this package (directly, not by inheritance), a new symbol is created with this name and inserted in package as an internal symbol. The symbol is also placed on the shadowing-symbols list of package.

package can be a package object or the name of a package (a symbol or a string). If unspecified, package defaults to the value of *package*. Returns t.

shadow should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

The following function checks if a list of symbols has already been made shadowing symbols of the indicated package, and if not, calls shadow.

```lisp
(defun my-shadow(symbols &optional (package *package*))
  (let ((shadowing-symbols (package-shadowing-symbols package)))
    (dolist (symbol symbols)
      (unless (member symbol shadowing-symbols)
        (shadow symbol package))))
```

**shadowing-import** symbols &optional package

*Function*
Like import, but does not signal an error even if the importation of a symbol would shadow some symbol already available in the package. If a distinct symbol with the same name is already present in the package, it is removed (using unintern). The imported symbol is placed on the shadowing-symbols list of package.

The symbols argument should be a list of symbols or a single symbol. If symbols is nil, it is treated like an empty list. package can be a package object or the name of a package (a symbol or a string). If unspecified, package defaults to the value of *package*. Returns t.

shadowing-import should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

```lisp
=> *package*
TURBINE-PACKAGE
=> (export valve-pressure)
T
=> (shadowing-import generator:valve-pressure)
```

**clos:shared-initialize** instance slot-names &rest initargs

*Generic Function*
Initializes the instance according to the initargs, then initializes any unbound slots in slot-names according to their initforms, and returns the initialized instance. This generic function is intended to be specialized by programmers, but not to be called directly.

instance The instance to initialize.

slot-names A list of slot names, or nil, or t. This specifies which slots should be initialized according to their initforms, if no initialization arguments are provided that initialize the slot. nil specifies no slots; t specifies all slots; and a list of slot names specifies the just the slots named.

initargs Alternating initialization argument names and values.

The default primary method for clos:shared-initialize does the following:

1. Fills slots with values according to the initargs. That is, for any initialization argument name that is associated with a slot, the value of the slot is initialized according to the argument given to clos:make-instance.

2. Fills any unbound slots indicated by the second argument to clos:shared-initialize with values according to the initform of the slot. The initform is specified by the :initform slot option to clos:defclass.

Users can define after-methods for clos:shared-initialize, to customize the initialization behavior that occurs in several cases. Note that a user-defined primary method for clos:shared-initialize would override the default method, and thus could prevent the usual slot-filling behavior. The clos:shared-initialize generic function is called in these cases:

- When an instance is first created; that is, when clos:make-instance is called.

- When an instance is reinitialized; that is, when clos:reinitialize-instance is called.

- When the class of an instance is changed; that is, when clos:update-instance-for-different-class is called.

- When a class is redefined; that is, when clos:update-instance-for-redefined-class is called.

shiftf &rest references-and-values

Macro

Each references-and-values can be any form acceptable as a generalized variable to setf. All the forms are treated as a shift register; the last references-and-values is shifted in from the right, all values shift over to the left one place, and the value shifted out of the first references-and-values position is returned.
For example, as seen in a Lisp Listener:

```lisp
(setq forces (list army navy air-force marines))
=> (ARMY NAVY AIR-FORCE MARINES)

(shiftf (car forces) (cadr forces) 'new-york-cops)
=> ARMY

forces
=> (NAVY NEW-YORK-COPS AIR-FORCE MARINES)

(shiftf (cadr forces) (cddr forces) 'monterey-lifeguards)
=> NEW-YORK-COPS

forces
=> (NAVY (AIR-FORCE MARINES) . MONTEREY-LIFEGUARDS)
```

A large number of place forms are predefined, and additions can be made via `defsetf` or `define-setf-method`. See the macro `setf`.

The following example illustrates the use of `shiftf` in scrolling a line-segment of bits, such as for a portion of a bit-mapped display.

```lisp
(setq s #*10011101)
#*10011101
(setq carry-bit
  (shiftf (bit s 0) (bit s 1) (bit s 2) (bit s 3)
    (bit s 4) (bit s 5) (bit s 6) (bit s 7)
      0))
1
s
#*00111010
```

### short-float

**Type Specifier**

`short-float` is the type specifier symbol for the predefined Lisp single-precision floating-point number type.

The type `short-float` is a subtype of the type `float`. In Symbolics Common Lisp `short-float` is identical with `single-float`.

The type `short-float` is disjoint with the types `long-float` and `double-float`.

Examples:

```lisp
(typep 0.0 'short-float) => T

(subtypep 'short-float 'float) => T and T ; subtype and certain

(commonp 1.0) => T
```
(equal-typep 'short-float 'single-float) => T

See the section "Data Types and Type Specifiers". See the section "Numbers".

**short-float-epsilon**  
*Constant*

The value is the smallest positive floating-point number $e$ of a format such that it satisfies the expression:

$$\text{(not (= (float 1 e) (+ (float 1 e) e)))}$$

In Symbolics Common Lisp `short-float-epsilon` has the same value as `single-float-epsilon`, namely: 5.960465e-8.

**short-float-negative-epsilon**  
*Constant*

The value is the smallest positive floating-point number $e$ of a format such that it satisfies the expression:

$$\text{(not (= (float 1 e) (- (float 1 e) e)))}$$

In Symbolics Common Lisp the value of `short-float-negative-epsilon` is the same as that of `single-float-negative-epsilon`, namely: 2.9802326e-8.

**short-site-name**  
*Function*

Returns a string that is the name of your site. This is the contents of the `site` field in your site's namespace object.

The CLOE Runtime environment does not provide a uniform way to obtain a "site" designation. If the value of the variable `cloe::*short-site-name*` is `nil`, you are prompted to enter the correct values for your site. Initially, `cloe::*short-site-name*` is set to "CLOE-USER-SITE."

**si:show-login-history**  &optional (whole-history si:login-history)  
*Function*

Prints one line for each time the login command has been used since the world was last cold booted. See the section "Show Login History Command".

**signal**  *flavor*  &rest *init-options*  
*Function*

The primitive function for signalling a condition. The argument *flavor* is a condition flavor symbol. The *init-options* are the init options when the `condition-object` is created; they are passed in the :init message to the instance. (See the function `make-instance`.) *signal* creates a new condition object of the specified flavor, and signals it. If no handler handles the condition and the object is not an error object, *signal* returns `nil`. If no handler handles the condition and the object is an error object, the Debugger assumes control.
In a more advanced form of `signal`, `flavor` can be a condition object that has been created with `make-condition` but not yet signalled. In this case, `init-options` is ignored.

Note that in CLOE, if `typep` condition `cloe::*break-on-signals*` is true, then the debugger will be entered prior to beginning the signalling process. The `continue` restart may be used to continue with the signalling process. This is true also for all other functions and macros which signal conditions, such as `warn`, `error`, `cerror`, `assert`, and `check-type`.

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

**signal-proceed-case**  
*Special Form*

Signals a proceedable condition. It has a clause to handle each proceed type of the condition. It has a slightly more complicated syntax than most special forms: you provide some variables, some argument forms, and some clauses:

```
(signal-proceed-case ((var1 var2 ... arg1 arg2 ...)  
                     (proceed-type-1 body1...)  
                     (proceed-type-2 body2...)  
                     ...)  
```

The first thing this form does is to call `signal`, evaluating each `arg` form to pass as an argument to `signal`. In addition to the arguments you supply, `signal-proceed-case` also specifies the `dbg:proceed-types` init option, which it builds based on the `proceed-type-i` clauses.

When `signal` returns, `signal-proceed-case` treats the first returned value as the symbol for a proceed type. It then picks a `proceed-type-i` clause to run, based on that value. It works in the style of `case`: each clause starts with a proceed type (a keyword symbol), or a list of proceed types, and the rest of the clause is a list of forms to be evaluated. `signal-proceed-case` returns the values produced by the last form.

`var1`, `var2`, and so on, are bound to successive values returned from `signal` for use in the body of the `proceed-type-i` clause selected.

One `proceed-type-i` can be `nil`. If `signal` returns `nil`, meaning that the condition was not handled, `signal-proceed-case` runs the `nil` clause if one exists, or simply returns `nil` itself if no `nil` clause exists. Unlike `case`, no otherwise clause is available for `signal-proceed-case`.

The value passed as the `dbg:proceed-types` option to `signal` lists the various proceed types in the same order as the clauses, so that the Debugger displays them in that order to the user and the `RESUME` command runs the first one.

**signed-byte**  
*Type Specifier*

`signed-byte` is the type specifier denoting the set of integers that can be represented in two's-complement form in a byte of `n` bits. It is the same as the type specifier `integer`. 
zl:signp test x

Tests the sign of a number. It is present only for compatibility with older versions of Lisp, and is not recommended for use in new programs. zl:signp returns t if x is a number that satisfies the test, nil if it is not a number or does not meet the test. test is not evaluated, but x is. test can be one of the following:

l x < 0
le x ≤ 0
e x = 0
n x ≠ 0
ge x ≥ 0
g x > 0

Examples:

(zl:signp ge 12) => t
(zl:signp le 12) => nil
(zl:signp n 0) => nil
(zl:signp g 'foo) => nil

For a table of related items, see the section "Numeric Property-checking Predicates".

signum number

Determines the sign of its argument.

For a rational argument, signum returns -1, 0, or 1, depending on whether the argument is negative, zero, or positive.

If the argument is a floating-point number, the result is a floating-point number of the same format whose value is minus one, zero, or one.

For a non-zero complex argument z, (signum z) returns a complex number of the same phase as z but with unit magnitude. If z is a complex zero, signum returns zero.

Examples:

(signum -2.5)  =>  -1.0
(signum 3.9)  =>  1.0
(signum 0)  =>  0
(signum 59)  =>  1
(signum #C(3 4)) => #C(0.6 0.8)

For a table of related items: See the section "Arithmetic Functions".

simple-array &optional ( element-type "*" ) ( dimensions "*" )

simple-array is the type specifier symbol for the Lisp data structure of that name.
The type **simple-array** is a **subtype** of the type **array**.

The types **simple-vector**, **simple-string**, and **simple-bit-vector** are **disjoint subtypes** of the type **simple-array**: **simple-vector** means (simple-array t (*)); **simple-string** means (simple-array string-char) or (simple-array character); **simple-bit-vector** means (simple-array bit (*)).

This type specifier can be used in either symbol or list form. Used in list form, **simple-array** allows the declaration and creation of specialized simple arrays whose members are all members of the type **element-type** and whose dimensions match **dimensions**. This is equivalent to

```
(array element-type dimensions)
```

except that it additionally specifies that objects of the type are **simple** arrays. (A simple array is an array that has no fill pointer, whose contents are not shared with another array, and whose size is not adjusted dynamically after creation.)

**element-type** must be a valid type specifier, or unspecified. For standard Symbolics Common Lisp type specifiers: See the section "Type Specifiers".

**dimensions** can be a non-negative integer, which is the number of dimensions, or it can be a list of non-negative integers representing the length of each dimension (any of which can be unspecified). **dimensions** can also be unspecified.

Examples:

```
(setq example-array (make-array '(3) :fill-pointer 2))
=> #<ART-Q-3 1321277>
(setq example-simple-array (make-array '(3))) => #<ART-Q-3 1330466>
(typep example-simple-array 'simple-array) => T
(zl:typep example-simple-array) => :ARRAY
(subtypep 'simple-array 'array) => T and T
(sys:type-arglist 'simple-array)
=> (&OPTIONAL (ELEMENT-TYPE '*) (DIMENSIONS '*)) and T
```

See the section "Data Types and Type Specifiers". See the section "Arrays".

**simple-bit-vector** &optional ( **size** '*)

---

**Type Specifier**

**simple-bit-vector** is the type specifier symbol for the Lisp data structure of that name.

**simple-vector**, **simple-string**, and **simple-bit-vector** are **disjoint subtypes** of the type **simple-array**: **simple-vector** means (simple-array t (*)); **simple-string** means (simple-array string-char) or (simple-array character); **simple-bit-vector** means (simple-array bit (*)).

This type specifier can be used in either symbol or list form. Used in list form, **simple-bit-vector** defines the set of bit-vectors of the indicated **size**. This means the same as (simple-array bit (**size**)).

Examples:
(setq array-bit-vector-not-simple
  (make-array '(3) :element-type 'bit :fill-pointer 2))
=> #<ART-1B-3 43035106>
(setq array-bit-vector-simple
  (make-array '(3) :element-type 'bit))
=> #<ART-1B-3 43054543>
(typep array-bit-vector-simple 'simple-array) => T
(typep array-bit-vector-not-simple 'simple-array) => NIL
(typep #*1 '(simple-bit-vector 1)) => T
(subtypep 'simple-bit-vector 'simple-array) => T and T
(subtypep 'simple-bit-vector 'bit-vector) => T and T
(simple-bit-vector-p array-bit-vector-simple) => T
(sys:type-arglist 'simple-bit-vector)
=> (&OPTIONAL (SIZE '*)) and T

See the section "Data Types and Type Specifiers". See the section "Arrays".

**simple-bit-vector-p** object

Tests whether the given object is a simple bit vector. A simple bit vector is a one-dimensional array whose elements are required to be bits; the array is not displaced to another array and has no fill pointer. See the type specifier **simple-bit-vector**. Under CLOE, a simple bit vector has no fill pointer, and is not adjustable or displaced.

(setq foo (make-array '(5) :element-type 'bit))
(setq bar (make-array '(5) :element-type 'bit :adjustable t))

(simple-bit-vector-p foo) => t
(simple-bit-vector-p bar) => nil

(simple-bit-vector-p
  (make-array 3 :element-type 'bit))
=> T

(simple-bit-vector-p
  (make-array 5 :element-type 'bit :fill-pointer 2))
=> NIL

For a table of related items: See the section "Operations on Vectors".

**simple-string** &optional (size '*)

**simple-string** is the type specifier symbol for the predefined Lisp data type, simple string.

The type **simple-string** is a subtype of the type **string**.
Note: Although string is a subtype of vector, simple-string is not a subtype of simple-vector.

The types simple-vector, simple-string, and simple-bit-vector are disjoint subtypes of the type simple-array: simple-vector means (simple-array t (*)); simple-string means (simple-array string-char) or (simple-array character); simple-bit-vector means (simple-array bit (*)).

This type specifier can be used in either symbol or list form. Used in list form, simple-string defines the set of simple strings whose size is restricted to size. This means the same as (simple-array string-char (size)), or (simple-array character (size)).

Examples:

```lisp
(setq string-one (make-string 5 :initial-element #\.)) => "....."); a thin, simple string
(setq string-two (make-array 3 :element-type 'character
  :initial-element #\x)) => "xxx" ; a fat, simple string
(typep string-one 'simple-string) => T
(typep string-two 'simple-string) => T
(simple-string-p string-one) => T
(simple-string-p string-two) => T
(subtypep 'simple-string 'string) => T and T
(subtypep 'simple-string 'vector) => T and T
(subtypep 'simple-string 'simple-array) => T and T
(commonp string-two) => T
(sys:type-arglist 'simple-string) => (&OPTIONAL (SIZE *)) and T
```

See the section "Data Types and Type Specifiers". See the section "Strings".

**simple-string-p object**

Function

Determines if object is a simple string array (one with no fill pointer and no displacement), returning t if it is, and nil otherwise. Accepts any object as an argument. A simple string is a one-dimensional array; under Genera, its elements can be characters of type string-char or character. Under CLOE, its elements must be of type string-char. In both CLOE and Genera, the array must have no fill pointer or displacement. Additionally, in CLOE the string must not be adjustable.

simple-string is a subtype of type string. simple-string-p is always t for strings built with make-string.

Examples:
(simple-string-p "fred") => T
(simple-string-p (make-string 3 :initial-element #\z)) => T
(simple-string-p (make-string 4 :initial-element #\hyper-a)) => T
(simple-string-p (make-array 5 :element-type 'string-char :fill-pointer t)) => NIL
(simple-string-p (make-array 2 :element-type 'character :initial-element #\b)) => T
(setq foo (make-array '(5) :element-type 'character))
(setq bar (make-array '(5) :element-type 'character :adjustable t))
(simple-string-p foo) => t
(simple-string-p bar) => nil

For a table of related items: See the section "String Type-Checking Predicates".

**simple-vector** &optional (size '*)

Type Specifier

**simple-vector** is the type specifier symbol for the Lisp data structure of that name.

The type **simple-vector** is a **subtype** of the types:

- **vector**
- (vector t)

**Note:** Although **string** is a subtype of **vector**, **simple-string** is not a subtype of **simple-vector**.

The types **simple-vector**, **simple-string**, and **simple-bit-vector** are **disjoint subtypes** of the type **simple-array**: **simple-vector** means (simple-array t (*)); **simple-string** means (simple-array string-char) or (simple-array character); **simple-bit-vector** means (simple-array bit (*)).

This type specifier can be used in either symbol or list form. Used in list form, **simple-vector** defines the set of specialized one-dimensional arrays of size **size**. This is the same as (vector t size), except that it additionally specifies that its elements are simple general vectors.

**Examples:**

(typep #(13 3 0) 'simple-vector) => T
(subtypep 'simple-vector 'vector) => T and T
(sys:type-arglist 'simple-vector) => (&OPTIONAL (SIZE '*)) and T
(simple-vector-p #(a b c)) => T
(typep #\(1 \, 1 \, 2\) \'(simple-vector 3)) => T

See the section "Data Types and Type Specifiers". See the section "Arrays".

**simple-vector-p object**

Tests whether the given *object* is a simple general vector. A simple general vector is a one-dimensional array whose elements have no type constraints; the array is not displaced to another array and has no fill pointer. Additionally, in CLOE it cannot be adjustable. See the type specifier **simple-vector**.

```lisp
(simple-vector-p (make-array 3)) => T

(simple-vector-p (make-array 5 :element-type 'bit :fill-pointer 2)) => NIL

(setq foo (make-array '(5) :initial-element 12))
(setq bar (make-array '(5) :initial-element 12 :adjustable t))

(simple-vector-p foo) => t
(simple-vector-p bar) => nil
```

For a table of related items: See the section "Operations on Vectors".

**sin radians**

Returns the sine of *radians*. Examples:

```
(sin 0) => 0.0
(sin (/ pi 2)) => 0.9999999999999999d0
```

For a table of related items: See the section "Trigonometric and Related Functions".

**sind degrees**

Returns the sine of *degrees*. *degrees* can be any numeric type.

Examples:

```
(sind #\C(30 40)) => #\C(0.62687695 0.65492296)
(sind 30.0) => 0.5
(sind 30) => 0.5
(sind #\C(0.0 30.0)) => #\C(0.0 0.5478535)
```

For a table of related items: See the section "Trigonometric and Related Functions".
single-float

`single-float` is the type specifier symbol for the predefined Lisp single-precision floating-point number type.

The type `single-float` is a `subtype` of the type `float`. In Symbolics Common Lisp `single-float` is equivalent to `short-float`.

The type `single-float` is `disjoint` with the types `long-float` and `double-float`.

Examples:

```
(typep .007000 'single-float) => T
(subtypep 'single-float 'float) => T and T ;subtype and certain
(zl:typenp .123456 ) => :SINGLE-FLOAT
(typep -0.3 'common) => T
(sys:single-float-p 1.e3) => T
(equal-typep 'single-float 'short-float) => T
(sys:type-arglist 'single-float) => NIL and T
(type-of 63e8) => SINGLE-FLOAT
```

See the section "Data Types and Type Specifiers". See the section "Numbers".

single-float-epsilon

The value is the smallest positive floating-point number $e$ of a format such that it satisfies the expression:

```
(not (= (float 1 e) (+ (float 1 e) e))
```

The current value of `single-float-epsilon` is: 5.960465e-8.

single-float-negative-epsilon

The value is the smallest positive floating-point number $e$ of a format such that it satisfies the expression:

```
(not (= (float 1 e) (- (float 1 e) e))
```

The current value of `single-float-negative-epsilon` is: 2.9802326e-8.

sys:single-float-p object

`Function`

Returns `t` if `object` is a single-precision floating-point number, otherwise `nil`.

For a table of related items, see the section "Numeric Type-checking Predicates".

sinh radians

`Function`

Returns the hyperbolic sine of `radians`. Example:

```
(sinh 0) => 0.0
```
For a table of related items: See the section "Hyperbolic Functions".

### sixth list

*Takes a list as an argument, and returns the sixth element of the list.* *sixth* is identical to:

```
(nth 5 list)
```

*For example:*

```
(setq letters '(a b c d e f g)) => (A B C D E F G)
```

```
(sixth letters) => F
```

This function is provided because it makes more sense when you are thinking of the argument as a list rather than just as a cons.

For a table of related items: See the section "Functions for Extracting from Lists".

### clos:slot-boundp instance slot-name

*Returns true if the given slot has a value, otherwise returns false.*

*instance* The instance.

*slot-name* The name of the slot of interest. This can be a local or shared slot.

One use for *clos:slot-boundp* is in writing after-methods for *clos:initialize-instance* in order to initialize unbound slots.

If there is no slot of the given name accessible to the instance, *clos:slot-missing* is called. The default method for *clos:slot-missing* signals an error.

### clos:slot-exists-p object slot-name

*Returns true if the object has a slot named *slot-name*, otherwise returns false.*

*object* Any Lisp object.

*slot-name* The name of the slot of interest.

### clos:slot-makunbound instance slot-name

*Makes the given slot unbound. Returns the instance.*

*instance* An instance.

*slot-name* The name of the slot that should be made unbound. This can be a local or shared slot.
If there is no slot of the given name accessible to the instance, **clos:slot-missing** is called. The default method for **clos:slot-missing** signals an error.

**clos:slot-missing** class instance slot-name operation &optional new-value

Generic Function

Provides a mechanism for users to control what happens when a slot’s value is desired for access (when **clos:slot-value** is called, among other operations), and there is no slot of the given name accessible to the instance. The default method for **clos:slot-missing** signals an error.

The typical way to specialize **clos:slot-missing** is to define a primary method, which would override the default primary method.

This generic function is called automatically, and is not intended to be called by users.

class The class of the instance whose slot value is desired for access.

instance The instance whose slot value is desired for access.

slot-name The name of the slot desired for access.

operation The operation that caused **clos:slot-missing** to be invoked. This can be one of the following symbols:

  - **clos:slot-value**
  - **clos:slot-boundp**
  - **clos:slot-makunbound**
  - **future-common-lisp:setf**, indicating that
    
    (setf clos:slot-value)

new-value This argument is the new value which is to be written into the slot, when (setf **clos:slot-value**) is called. This argument is provided only if the operation argument is **future-common-lisp:setf**.

If a method for **clos:slot-missing** returns values, these values are returned as the values of the function that caused **clos:slot-missing** to be called.

**clos:slot-unbound** class instance slot-name

Generic Function

Provides a mechanism for users to control what happens when a slot’s value is desired for access and the slot is unbound. This generic function is called automatically in that case, and is not intended to be called by users.

The default primary method signals an error.

The typical way to specialize **clos:slot-unbound** is to define a primary method, which would override the default primary method.

class The class of the instance.
instance The instance whose slot is unbound.
slot-name The name of the unbound slot.

If a method for clos:slot-unbound returns a value, this value is returned as the value of the function that caused clos:slot-unbound to be called.

clos:slot-value instance slot-name
Function
Returns the value of a given slot. You can use setf with clos:slot-value to change the value of a slot. You can use locf with clos:slot-value to get a locative pointer to the cell inside an instance that contains the value of a slot.

instance The instance whose slot is desired.
slot-name The name of the slot whose value is desired. This can be a local or a shared slot.

If there is no slot of the given name accessible to the instance, then clos:slot-missing is called. The default method for clos:slot-missing signals an error.
If the slot is unbound, then clos:slot-unbound is called. The default method for clos:slot-unbound signals an error.
If you use location-contents on a shared slot which is unbound, the system has no way of knowing which instance you are interested in. Thus, clos:slot-unbound is called with an instance, but that instance is not necessarily the one of interest to you.

Note that you cannot use clos:slot-value on a class defined by defstruct.

software-type
Function
Returns a string that is the name of the operating system Lisp is running in.

    (software-type) => "Lisp Machine"

For the CLOE Developer

    (software-type) =>"Genera"

and for the CLOE Application Generator

    (software-type) =>"UNIX" or "MS-DOS"

software-version
Function
Returns a string that represents the versions of all the systems running in your world. This includes any local systems you have loaded. This is similar to the information displayed by the Show Herald command.

For the CLOE Developer

    (software-version) =>"8.0"
and for the CLOE Application Generator

(software-version) => "V.3" or "3.1"

math:solve lu ps b &optional x

Function

Takes the LU decomposition and associated permutation array produced by math:decompose, and solves the set of simultaneous equations defined by the original matrix a and the right-hand sides in the vector b. If x is supplied, the solutions are stored into it and it is returned; otherwise, an array is created to hold the solutions and that is returned. b must be a one-dimensional array.

some predicate sequence &rest more-sequences

Function

Returns a non-nil value as soon as any invocation of predicate returns a non-nil value. predicate must take as many arguments as there are sequences provided. predicate is first applied to the elements of the sequences with an index of 0, then with an index of 1, and so on, until a termination criterion is reached or the end of the shortest of the sequences is reached. If the end of a sequence is reached, some returns nil. Thus considered as a predicate, it is true if some invocation of predicate is true.

If predicate has side effects, it can count on being called first on all those elements with an index of 0, then all those with an index of 1, and so on. sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

For example:

(some #'oddp '(1 2 5)) => T

(some #'equal '(0 1 2 3) '(3 2 1 0)) => NIL

However, since some returns whatever the predicate returns, it does not have to be t.

For example:

(some #'(lambda (x) (if (oddp x) x)) '(2 4 3)) => 3

By using an anonymous function, the following example demonstrates how some implements a test to determine whether any element of a sequence exceeds a limiting value.

(setq limit-value 212 sequence (vector 16 64 512 128 32))

(some #'(lambda (x) (> x limit-value)) sequence) => t

For a table of related items: See the section "Predicates that Operate on Lists".

For a table of related items: See the section "Functions for Extracting from Lists".

For a table of related items: See the section "Predicates that Operate on Sequences".
**zl:some** *list pred &optional (step #’cdr)*

*Function*

Returns a tail of *list*, such that the car of the tail is the first element that satisfies *pred*. Returns *nil* if *pred* returns *nil* for every element. Example:

```
(setq list ’(a b 1 2)) => (A B 1 2)
(zl:some list #'numberp) => (1 2)
```

For a table of related items: See the section "Predicates that Operate on Sequences".

**sort** *sequence predicate &key key*  

*Function*

Destructively modifies *sequence* by sorting it according to an order determined by *predicate*. *predicate* should take two arguments and return a non-*nil* value if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), *predicate* should return *nil*.

The **sort** function determines the relationship between two elements by giving keys extracted from the elements to *predicate*. The :key argument, when applied to an element, should return the key for that element. It defaults to the identity function, thereby making the element itself the key.

*sequence* can be either a list or a vector (one-dimensional array). Note that *nil* is considered to be a sequence, of length zero.

The :key function should not have any side effects. A useful example of a :key function would be a component selector function for a *defstruct* structure, used in sorting a sequence of structures.

If the :key and *predicate* functions always return, the sorting operation will always terminate, producing a sequence containing the same elements as the original sequence (that is, the result is a permutation of *sequence*). This is guaranteed even if *predicate* does not really consistently represent a total order (in which case the elements will be scrambled in some unpredictable way, but no element will be lost). If the :key function consistently returns meaningful keys, and the predicate does reflect some total ordering criterion on those keys, the elements of the result sequence will be properly sorted according to that ordering.

For example:

```
(sort #(1 3 2 4 3 5) #’>) => #(5 4 3 3 2 1)

(sort ’(((up 2) (down 1) (west 4) (south 3)) #’< :key #’cadr)
  => (((DOWN 1) (UP 2) (SOUTH 3) (WEST 4)))
```

The sorting operation performed by **sort** is not guaranteed stable. Elements considered equal by *predicate* may or may not stay in their original order. *predicate* is assumed to consider two elements *x* and *y* to be equal if (funcall *predicate* *x* *y*) and (funcall *predicate* *y* *x*) are both false. The function **stable-sort** guarantees stability, but can be slower than **sort** in some situations.
The sorting operation is destructive, so in the cases where the argument should not be destroyed, you must sort a copy of the argument. When the argument is an vector, the sort is accomplished by permuting the elements in place. When the argument is a list, the sort is accomplished by destructive reordering in the same manner as \texttt{nreverse}.

If the execution of either the \texttt{:key} or \texttt{predicate} functions causes an error, the state of the list or vector being sorted is undefined. However, if the error is corrected, the sort will proceed correctly.

Note that since sorting requires many comparisons, and thus many calls to \texttt{predicate}, sorting will be much faster if \texttt{predicate} is a compiled function rather than interpreted.

For example:

\begin{verbatim}
(setq bird-list '(heron stork loon owl flamingo turkey)) =>
(HERON STORK LOON OWL FLAMINGO TURKEY)

(srt bird-list #'string-lessp) =>
(FLAMINGO HERON LOON OWL STORK TURKEY)

(setq a (vector 1 2 3 4 5))

(setq a (sort a #'>)) => #(5 4 3 2 1)
\end{verbatim}

For a table of related items: See the section "Functions for Sorting Lists".

For a table of related items: See the section "Sorting and Merging Sequences".

\begin{verbatim}
(zl:sort x sort-lessp-predicate) Function

Sorts the contents of the one-dimensional array or list \texttt{x}, under the ordering imposed by \texttt{sort-lessp-predicate}, and returns the array or list modified into sorted order. Note that since sorting requires many comparisons, and thus many calls to the predicate, sorting is much faster if the predicate is a compiled function rather than interpreted.

The first argument to \texttt{zl:sort}, \texttt{x}, is a one-dimensional array or a list. The second, \texttt{sort-lessp-predicate}, is a predicate, which must be applicable to all the objects in the array or list. The predicate should take two arguments, and return \texttt{non-nil} if and only if the first argument is strictly less than the second (in some appropriate sense). The predicate should return \texttt{nil} if its arguments are equal. For example, to sort in the opposite direction from <, use >, not \texttt{\geq}. This is because the quicksort algorithm used to sort arrays and cdr-coded lists becomes very much slower if predicate has to return \texttt{non-nil} for equal elements. Example:

\begin{verbatim}
(defun mostcar (x)
  (cond ((symbolp x) x)
        ((mostcar (car x)))))
\end{verbatim}
\end{verbatim}
(zl:sort fooarray
  (function (lambda (x y)
       (alphalessp (mostcar x) (mostcar y))))))

If fooarray contained these items before the sort:
  (Tokens (The lion sleeps tonight))
  (Carpenters (Close to you))
  ((Rolling Stones) (Brown sugar))
  ((Beach Boys) (I get around))
  (Beatles (I want to hold your hand))

after the sort fooarray would contain:
  ((Beach Boys) (I get around))
  (Beatles (I want to hold your hand))
  (Carpenters (Close to you))
  ((Rolling Stones) (Brown sugar))
  (Tokens (The lion sleeps tonight))

When zl:sort is given a list, it can change the order of the conses of the list (using rplacd), and so it cannot be used merely for side effect; only the returned value of zl:sort is the sorted list. This changes the original list; if you need both the original list and the sorted list, you must copy the original and sort the copy. See the function copy-list.

Sorting an array just moves the elements of the array into different places, and so sorting an array only for side effect is all right.

If the x argument to zl:sort is an array with a fill pointer, note that, like most functions, zl:sort considers the active length of the array to be the length, and so, only the active part of the array is sorted. See the function zl:array-active-length.

For a table of related items: See the section "Functions for Sorting Lists".

For a table of related items: See the section "Sorting and Merging Sequences".

sort-grouped-array a gs sort-lessp-predicate

Sorts the records (units of gs elements each) of a with respect to one another. The sort-lessp-predicate is applied to the first element of each record, so the first elements act as the keys, on which the records are sorted.

sort-grouped-array is a Symbolics extension to Common Lisp.

sort-grouped-array-group-key a gs sort-lessp-predicate

Sorts the records (units of gs elements each) of a with respect to one another. sort-lessp-predicate is applied to four arguments: an array, an index into that array, a second array, and an index into the second array. sort-lessp-predicate should consider each index as the subscript of the first element of a record in the corresponding array, and compare the two records. This is more general than sort-
grouped-array, since the function can get at all of the elements of the relevant records, instead of only the first element.

sort-grouped-array-group-key is a Symbolics extension to Common Lisp.

**zl:sortcar** $x$ sort-lessp-predicate-on-car

*Function*

Same as **zl:sort**, except that the `sort-lessp-predicate-on-car` is applied to the *cars* of the elements of $x$, instead of directly to the elements of $x$. Example:

$$(\text{zl:sortcar } '((3 . dog) (1 . cat) (2 . bird)) #'<)$$

$=> ((1 . cat) (2 . bird) (3 . dog))$$

Remember that **zl:sortcar**, when given a list, can change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side effect; only the returned value of **zl:sortcar** is the sorted list.

For a table of related items: See the section "Functions for Sorting Lists".

**special** var1 var2 ...

*Declaration*

Specifies that *vars* are to be considered special.

See the section "Declaration Specifiers".

**dbg:special-command** condition &rest per-command-args

*Generic Function*

Sent when the user invokes the special command. It uses :case method-combination and dispatches on the name of the special command. No arguments are passed. The syntax is:

$$(\text{defmethod } (\text{dbg:special-command my-flavor :my-command-keyword}) ()$$

"documentation"

body...$$

Any communication with the user should take place over the *query-io* stream. The method can return nil to return control to the Debugger or it can return the same thing that any of the **sys:proceed** methods would have returned in order to proceed in that manner.

The compatible message for **dbg:special-command** is:

:*special-command*

For a table of related items: See the section "Debugger Special Command Functions".

**dbg:special-command-p** condition special-command

*Function*

Returns t if *command-type* is a valid Debugger special command for this condition object; otherwise, returns nil.

The compatible message for **dbg:special-command-p** is:
:special-command-p
For a table of related items, see the section "Basic Condition Methods and Init Options".

dbg:special-commands condition  
Generic Function
Returns a list of all Debugger special commands for this condition. See the section "Debugger Special Commands".
The compatible message for dbg:special-commands is:

:special-commands
For a table of related items, see the section "Basic Condition Methods and Init Options".

dbg:*special-command-special-keys*
Variable
The value should be an alist associating names of special commands with characters. When an error supplies any of these special commands, the Debugger assigns that special command to the specified key. For example, this is the mechanism by which the :package-dwim special command is offered on the c-sh-p keystroke.
For a table of related items, see the section "Debugger Special Key Variables".

special-form-p function  
Function
If function globally names a special form, returns a non-nil value, otherwise returns nil.

It is possible for both special-form-p and macro-function to be true for a given symbol. This is possible because implementors of Common Lisp dialects are permitted to implement any macro as a special form for speed.

This function is useful in writing code walking functions.

(special-form-p special-form-p)
NIL
(special-form-p 'quote)
#<function:542324>

sqrt number  
Function
Computes and returns the principal square root of number. If number is not complex but is negative, the result will be a complex number.

Examples:
For a table of related items, see the section "Arithmetic Functions".

**zl:sqrt**  \( n \) 

Returns the square root of \( n \). \( n \) must be a non-negative number.

Example:

\[
\begin{align*}
(zl:sqrt 4) & \Rightarrow 2.0 \\
(sqrt 81) & \Rightarrow 9.0 \\
(sqrt -4) & \Rightarrow #c(0.0 2.0) \\
(sqrt #c(-5.0 12.0)) & \Rightarrow #c(2.0 3.0)
\end{align*}
\]

For a table of related items: See the section "Arithmetic Functions" and see CLtL 205.

**zl:sstatus**  \( status\text{-}function \)  \( item \)  \( Special \)  \( Form \)

The \( zl\text{ status} \) and \( zl\text{ sstatus} \) special forms exist for compatibility with Maclisp. Programs that are designed to run in both Maclisp and Zetalisp can use \( zl\text{ status} \) to determine in which one they are running. Also, \( (zl\text{ sstatus} \text{ feature} \ldots) \) can be used as it is in Maclisp.

\( (zl\text{:sstatus} \text{ feature} \text{ symbol}) \) adds \( \text{symbol} \) to the list of features.

\( (zl\text{:sstatus} \text{ nofeature} \text{ symbol}) \) removes \( \text{symbol} \) from the list of features.

**stable-sort**  \( sequence \)  \( predicate \)  \( \&\text{key} \)  \( key \)  \( Function \)

Destructively modifies \( sequence \) by sorting it according to an order determined by \( predicate \). \( stable\text{- sort} \) is the stable version of \( sort \). \( stable\text{- sort} \) guarantees that elements considered equal by \( predicate \) will remain in their original order. \( predicate \) is assumed to consider two elements \( x \) and \( y \) to be equal if \( (\text{funcall predicate } x \ y) \) and \( (\text{funcall predicate } y \ x) \) are both false. \( stable\text{- sort} \) can be slower than \( sort \) in some situations.

See the function \( sort \).

In the following example, although considered equal by \( \text{char\text{-} lessp} \), \#\text{"a} and \#\text{"A} remain in their original order.

\[
\begin{align*}
(\text{stable\text{- sort} (vector #\text{"b} #\text{"A} #\text{"a} #\text{"c}) #\text{\'char\text{-} lessp})} \\
\Rightarrow ( #\text{"A} #\text{"a} #\text{"b} #\text{"c})
\end{align*}
\]

For a table of related items: See the section "Functions for Sorting Lists".
For a table of related items: See the section "Sorting and Merging Sequences".

\textbf{zl:stable-sort} \textit{x lessp-predicate} \ \ \ \textit{Function}

Like \textbf{zl:sort}, but if two elements of \textit{x} are equal, that is, \textit{lessp-predicate} returns \texttt{nil} when applied to them in either order, those two elements remain in their original order.

For a table of related items: See the section "Functions for Sorting Lists".
For a table of related items: See the section "Sorting and Merging Sequences".

\textbf{zl:stable-sortcar} \textit{x sort-lessp-predicate-on-car} \ \ \ \textit{Function}

Like \textbf{zl:sortcar}, but if two elements of \textit{x} are equal, that is, \textit{sort-lessp-predicate-on-car} returns \texttt{nil} when applied to their cars in either order, then those two elements remain in their original order.

For a table of related items: See the section "Functions for Sorting Lists".

\textbf{standard-char} \ \ \ \textit{Type Specifier}

This is the type specifier symbol for the predefined Lisp standard character data type \textbf{standard-char}.

The type \textbf{standard-char} is a \textit{subtype} of the type \textbf{string-char}.

Examples:

\begin{verbatim}
  (setq a-string (make-array 4 :element-type 'standard-char :initial-element '#\∞)) => "∞∞∞∞"
  (typep #\> 'standard-char) => T
  (subtypep 'standard-char 'string-char) => T and T
  (string-char-p (char a-string 1)) => T
  (standard-char-p '#\!) => T
  (sys:type-arglist 'standard-char) => NIL and T
\end{verbatim}

See the section "Data Types and Type Specifiers". See the section "Characters".

\textbf{standard-char-p} \textit{char} \ \ \ \textit{Function}

Returns \texttt{t} if \textit{char} is one of the Common Lisp standard characters. \textit{char} must be a character object.

The Common Lisp standard character set includes:

\begin{verbatim}
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
  a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
\end{verbatim}

See the section "Type Specifiers and Type Hierarchy for Characters".
(standard-char-p #\C) => t

(standard-char-p #\Control-C) => nil

For a table of related items: See the section "Character Predicates".

clos:standard-class

The default class of classes defined by clos:defclass.

The term "user-defined class" means a class whose class is clos:standard-class. You can define methods that specialize on these classes; you can use clos:make-instance to create instances of these classes; and you can redefine these classes.

clos:standard-generic-function

The default class of generic function objects. By default, the class of a generic function object created by clos:defgeneric is clos:standard-generic-function.

*standard-input*

In the normal Lisp top-level loop, input is read from whatever stream is the value of *standard-input*. Many input functions, including read and read-char, take a stream argument that defaults to *standard-input*.

(read) = (read *standard-input*)

zl:standard-input

In your new programs, we recommend that you use the variable *standard-input*, which is the Common Lisp equivalent of zl:standard-input.

In the normal Lisp top-level loop, input is read from zl:standard-input (that is, whatever stream is the value of zl:standard-input). Many input functions, including zl:tyi and zl:read, take a stream argument that defaults to zl:standard-input.

clos:standard-method

The default class of method objects. By default, the class of a method object created by clos:defmethod is clos:standard-method.

*standard-output*

In the normal Lisp top-level loop, output is sent to whatever stream is the value of *standard-output*. Many input functions, including write and write-char, take a stream argument that defaults to *standard-output*.

(print 'foo) = (print 'foo *standard-output*)
The variable *standard-output* may be set to a file, for example, rather than an interactive stream, thus redirecting subsequent output to the file:

```lisp
(setq outstream
  (open "myfile" :direction :output))  ; opens myfile.lisp
(setq old-standard-out *standard-output*)  ; save old value
(setq *standard-output* outstream)         ; redirects output
(print 'foo)                               ; prints 'foo in myfile.lisp
(setq *standard-output* old-standard-out)  ; restore *standard-output*
```

It is much better, however, to use let to temporarily bind the stream:

```lisp
(with-open-file (outstream "myfile" :direction :output)
  (let ((*standard-output* outstream))  ; redirects output
    (print 'foo))                       ; end of let form restores
    ; *standard-output*
    ; more forms
)                                      ; end of with-open-file closes file
```

By setting *standard-output* to a synonym-stream of *terminal-io*, *standard-output* can resume writing to the user console.

### zl:standard-output

In your new programs, we recommend that you use the variable *standard-output*, which is the Common Lisp equivalent of zl:standard-output. See the variable *standard-output*.

### si:standard-readtable

The value is that readtable to use when typing forms interactively to the Lisp interpreter. When a distribution world is cold booted, the value of si:standard-readtable is a copy of si:initial-readtable. If you wish to customize the syntax of forms typed to the Lisp interpreter, you should make your customizations to si:standard-readtable. *readtable* is bound to si:standard-readtable whenever a break loop or debug loop is entered. *readtable* is set to si:standard-readtable using the standard variable mechanism whenever the machine is warm booted.

If warm booting the machine were impossible, si:standard-readtable would not be necessary. The top-level value of *readtable* could be used instead. However, if the machine is warm booted while *readtable* is bound, the top-level value of *readtable* is lost.

Examples:

- This example illustrates the use of binding *readtable* in order to implement a special syntax. Forms are to be read from a file while preserving the case of symbols.

  ```lisp
  (defvar *case-sensitive-readtable* (copy-readtable))
  ```
(loop for code from (char-code #/a) to (char-code #/z)
     as char = (code-char code)
     do (setf (si:rdtbl-trans *case-sensitive-readtable* code) char))

(defun read-case-sensitive-file (file)
  (with-open-file (stream file :direction :input)
    (let ((*readtable* *case-sensitive-readtable*))
      (loop do (process-form (read stream))))))

In case an error occurs while inside process-form or inside a reader macro invoked by read, *readable* is bound to si:standard-readtable, which is most useful for debugging.

- This example illustrates the use of si:standard-readtable and si:initial-readtable to customize the environment for typing expressions interactively. "@" is defined as an abbreviation for location-contents, in the same manner that "'" is an abbreviation for quote.

  (defun at-sign-macro (ignore stream)
    (values (list 'location-contents (read stream)) 'list))

  (defvar *my-readtable* (copy-readtable))
  (set-syntax-macro-char #/@ 'at-sign-macro *my-readtable*)

  (defun enable-my-readtable ()
    (setq si:standard-readtable *my-readtable*)
    (setq *readtable* *my-readtable*))

  (defun disable-my-readtable ()
    (setq si:standard-readtable si:initial-readtable)
    (setq *readtable* si:initial-readtable))

While it is useful for the user to set the values of *readtable* and si:standard-readtable, the value of si:initial-readtable should never be changed. In addition, the readable that is the value of si:initial-readtable should never be modified, modifications should be made only to the readable that is the value of si:standard-readtable. See the function copy-readtable.

See the section "The Readtable".

**sys:standard-value symbol &key (listener nil) (global-p nil)**

Function

Returns the standard value associated with symbol. If global-p is t, it returns the standard value independent of any standard value bindings made with sys:standard-value-let or sys:standard-value-progv. If listener is non-nil, it must be a flavor instance that supports the standard value binding environment protocol. The value returned will be the binding specific to that environment.
You change the standard value of `symbol` with `(setf (sys:standard-value symbol &key (listener nil) (global-p nil) (setq-p nil)))`. Note that if there is a standard value binding for `symbol`, only the bound value is changed. The usual constraints apply to the values of `listener`.

If `setq-p` is `t`, the value cell of `symbol` is set to the same value as the standard value.

If `global-p` is `t`, both the standard value setting and the value cell setting, if any, are set in the global environment rather than in any existing binding environment.

<table>
<thead>
<tr>
<th>Ordinary Symbol</th>
<th>Standard Value Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(setq foo t)</code></td>
<td><code>(setf (sys:standard-value foo :setq-p t) t)</code></td>
</tr>
<tr>
<td><code>(zl:set-globally 'foo t)</code></td>
<td><code>(setf (sys:standard-value foo :global-p t :setq-p t) t)</code></td>
</tr>
</tbody>
</table>

See the section "Standard Variables".

**sys:standard-value-let** `vars-and-vals &body body` **Macro**

Like `let` except that it also pushes the values in `vals` onto the `si:*interactive-bindings*` alist, causing them to become the new standard bindings. All the symbols in `vars` are then bound to `vals` (with a `let`) and `body` is executed in this context.

Example:

```lisp
(defun octal-top-level ()
  (sys:standard-value-let
   ((base 8)
    (ibase 8)
    (package (pkg-find-package 'new-command-loop)))
   (let ((standard-io 'terminal-io))
     (loop
      as form = (read)
      do (print (eval form))))))
```

See the section "Standard Variables".

**sys:standard-value-let*** `vars-and-vals &body body` **Macro**

Like `let*` except that it also pushes the values in `vals` onto the `si:*interactive-bindings*` alist, causing them to become the new standard bindings. All the symbols in `vars` are then bound to `vals` (with a `let*`) and `body` is executed in this context. See the section "Standard Variables".

**sys:standard-value-p** `symbol` **Function**

Returns `t` if `symbol` has a standard value. See the section "Standard Variables".
sys:standard-value-progv vars vals &body body

Macro

Causes all of the symbols in vars to have their corresponding value in vals pushed onto the si:*interactive-bindings* alist (that is, those values become the new standard bindings). All the symbols in vars are then bound to vals (with a progv) and body is executed in this context. This is useful for writing Lisp-style command loops. See the section "Standard Variables".

:start-open-auxiliary-stream active-p &key local-id foreign-id stream-options application-id

Message

Sent to a stream to establish another stream, via another connection, over the same network medium, to the same host. It is used for either end of the connection.

If active-p is t, it means this side will connect and the remote side should listen; if active-p is nil, the remote side will connect and this side will listen.

If this side is active, foreign-id is the foreign contact identifier to connect to.

If this side is not active, local-id is the local identifier to listen on. The content of foreign-id and local-id depends on the network implementation. If this side is not active, and no local-id is supplied, application-id must be supplied. application-id is a string that the network uses as part of the the contact identifier it will create and return.

:start-open-auxiliary-stream returns two arguments, stream and contact-identifier. stream is a new stream. It is not yet usable. You can do one of two things with it:

- Terminate the establishment of the new connection by sending the message :close :abort or :close-with-reason :abort to the stream.

- Wait for the connection to be fully established, by sending :complete-connection to the stream.

contact-identifier is a string representing the contact name actually being listened to, in the case that this side is not active. This is the string to convey to the other side, so that the other side can supply it as the foreign-id argument of start-open-auxiliary-stream, to connect back to this side.

zl:status status-function &optional item

Special Form

The zl:status and zl:sstatus special forms exist for compatibility with Maclisp. Programs that are designed to run in both Maclisp and Zetalisp can use zl:status to determine in which one they are running. Also, (zl:sstatus feature ...) can be used as it is in Maclisp.

(zl:status features) returns a list of symbols indicating features of the Lisp environment. The default list for 3600-family machines is:
The value of this list will be kept up to date as features are added or removed from the Genera system. Most important is the symbol :lispm; this indicates that the program is executing on a Symbolics 3600-family machine. The order of this list should not be depended on, and might not be the same as shown above.

The following symbols in the features list can be used to distinguish different Lisp implementations, using the #+ and #- reader syntax.

Three symbols indicate which Lisp Machine hardware is running:

:lispm Any kind of Lisp Machine, as opposed to Maclisp
:cadr An M.I.T. CADR
:3600 A 3600-family machine

One symbol indicates which kind of Lisp Machine software is running:

:symbolics Symbolics software

See the section "Sharp-sign Reader Macros".

(zl:status feature symbol) returns t if symbol is on the (zl:status features) list, otherwise nil.

(zl:status nofeature symbol) returns t if symbol is not on the (zl:status features) list, otherwise nil.

(zl:status userid) returns the name of the logged-in user.

(zl:status tabsize) returns the number of spaces per tab stop (always 8). Note that this can actually be changed on a per-window basis: however, the zl:status function always returns the default value of 8.

(zl:status opsys) returns the name of the operating system, always the symbol :lispm.

(zl:status site) returns the name of the local machine, for example, "WOMBAT". Note that this is not the same as the value of zl:site-name.

(zl:status zl:status) returns a list of all zl:status operations.

(zl:status zl:ssstatus) returns a list of all zl:ssstatus operations.

Some of these zl:status functions are subsumed by the Common Lisp variable *features* and the functions software-type, short-site-name, and long-site-name.

step form Special Form

Evaluates form with single stepping. It returns the value of form.

For example, if you have a function named foo, and typical arguments to it might be t and 3, you could say
(step (foo t 3))

See the section "Stepping Through an Evaluation".

Note that at deep levels of recursion, the indentation of the step output is reset to column 0, so the output is more readable to the user, instead of running into the right margin of the screen. The variable si:*step-indentation-restart-fraction* controls when the indentation is set back to 0. Its value is a non-zero fraction of the screen size after which the stepper should go back to column 0 for its indentation, or nil to prevent the stepper from ever resetting to column 0.

**zl:step form**

Evaluates form with single stepping. It returns the value of form.

For example, if you have a function named foo, and typical arguments to it might be t and 3, you could say

    (step (foo t 3))

See the section "Stepping Through an Evaluation".

Note that at deep levels of recursion, the indentation of the step output is reset to column 0, so the output is more readable to the user, instead of running into the right margin of the screen. The variable si:*step-indentation-restart-fraction* controls when the indentation is set back to 0. Its value is a non-zero fraction of the screen size after which the stepper should go back to column 0 for its indentation, or nil to prevent the stepper from ever resetting to column 0.

**step-form**

Holds the current form when you are using step.

**step-value**

Holds the first returned value when you are using step

**step-values**

Holds the list of returned values when you are using step.

**zl:store-array-leader value array index**

Stores value in the indexed element of array's leader. array should be an array with a leader, and index should be an integer. value can be any object. zl:store-array-leader returns value.

However, the preferred method is to use setf and array-leader, as shown in the following example:
(make-array '(2 3) :leader-list '(t nil))
(setf (array-leader array 1) 'x)

stream

stream-copy-until-eof from-stream to-stream &optional leader-size

Function

Inputs characters from from-stream and outputs them to to-stream until it reaches
the end-of-file on the from-stream. For example, if x is bound to a stream for a file
opened for input, (stream-copy-until-eof x zl:terminal-io) prints the file on the
console.

If from-stream supports the :line-in operation and to-stream supports the :line-out
operation, stream-copy-until-eof uses those operations instead of :tyi and :tyo, for
greater efficiency. leader-size is passed as the argument to the :line-in operation.

sys:stream-default-handler stream op arg1 rest

Function

Tries to handle the op operation on stream, given arguments of arg1 and the ele-
ments of rest. The action taken for each of the defined operations is explained with
the documentation on that operation. The handler sends the :any-tyi message for :
line-in messages to streams that do not handle :line-in themselves.

For examples of the use of this function, see the section "Examples of Making
Your Own Stream".

stream-element-type stream

Function

Returns a type specifier which indicates what objects can be read from or written
to stream. Streams created by open will have an element type restricted to a sub-
set of character or integer, but in principle a stream may transfer any Lisp ob-
ject.

(setq file-stream
  (open "foo" :direction :output :element-type 'character))

(stream-element-type file-stream) => CHARACTER

streamp x

Function

Returns t if x is a stream, otherwise returns nil.

(streamp *standard-output*) => T
(streamp 'ａ*standard-output*) => NIL
(streamp t) => NIL
(streamp nil) => NIL
(streamp 3) => NIL
string &optional (size '*)

string is the type specifier symbol for the predefined Lisp string data type. This type specifier can be used in either symbol or list form. Used in list form, string allows the declaration and creation of specialized types of strings whose size is restricted to size.

The type string is a subtype of the type vector; string means (vector string-char) or (vector character).

The types string, (vector t), and bit-vector are disjoint.

The type string is a supertype of the type simple-string.

typep returns t for both thin strings (vector string-char), and fat strings (vector character). For example:

(equal-typep 'string '(vector string-char)) => T

(typep (make-array 1 :element-type 'character :initial-element #\control-a) 'string) => T

subtypep on the other hand, currently recognizes only (vector string-char) as a string.

(subtypep 'string '(vector string-char)) => T and T

Examples:

(typep "1;oi498f" 'string) => T
(typep "123" '(string 3)) => T
(typep "123" '(string 5)) => NIL
(zl:typen "U.S. Telephone Area Codes") => :STRING
(subtypep 'string 'vector) => T and T
(stringp "artificial intelligence") => T
(stringp (make-array 3 :element-type 'string-char :initial-element #\s :fill-pointer 2)) => T

(sys:type-arglist 'string) => (&OPTIONAL (SIZE '*)) and T

See the section "Data Types and Type Specifiers". See the section "Strings".

string x  

Function

Coerces x into a string. Most of the string functions apply this to their string arguments.

If x is a string, it is returned.

If x is a symbol, its print name is returned.

If x is a character, a string containing that character is returned.
If \( x \) is a pathname, under Genera the "string for printing" is returned. See the section "Pathname Messages: Naming of Files". Under CLOE, the name-string of \( x \) is returned.

If \( x \) is any instance that handles the :string-for-printing message, a "string for printing" is returned. This is incompatible with Common Lisp, which requires that string signal an error if its argument is neither a string, a symbol, nor a string-char. See the section "Pathname Messages: Naming of Files".

string does not convert a list or other sequence of characters to be a string. Use the function coerce for that purpose. (Unlike string, coerce does not work for symbols, though.)

If you want to get the string representation of a number or any other Lisp object, string is not what you should use. You can use format, passing a first argument of nil. You might also want to use with-output-to-string, prin1-to-string, or princ-to-string.

Examples:

\[
\begin{align*}
\text{(string "a string")} & \Rightarrow "\text{a string}"
\\
\text{(string 'symbol)} & \Rightarrow "\text{SYMBOL}"
\\
\text{(string \#\c)} & \Rightarrow "\text{c}"
\end{align*}
\]

The following are equivalent:

\[
\begin{align*}
\text{(string (si:patch-system-pathname "LMFS" :system-directory))} & \Rightarrow "\text{SYS:LMFS;PATCH;LMFS.SYSTEM-DIR.NEWEST}"
\\
\text{(send}
\\
\text{(si:patch-system-pathname "LMFS" :system-directory ) :string-for-printing)} & \Rightarrow "\text{SYS:LMFS;PATCH;LMFS.SYSTEM-DIR.NEWEST}"
\end{align*}
\]

For a table of related items: See the section "String Construction".

string\# string1 string2 &key (:start1 0) (:end1 (:start2 0) :end2) Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.

string\# returns nil unless string1 is not equal to string2. If the condition is satisfied, string\# returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

string1 and string2 must be strings, or objects that can be coerced to strings. See the function string.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.
:start1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :start1 must be ≤ :end1.

:end1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.

The case-insensitive version of string# is the function string-not-equal.

Examples:

(string# "apple" "apple") => NIL
(string# "apple" 'apple) => 0
(string# "apple" "apply") => 4
(string# "apple" "apropos") => 2
(string# "banana" "anachronism" :start1 1 :end1 4) => 3
(string# "banana" "anachronism" :start1 1 :end1 4 :end2 3) => NIL

The following function is a synonym of string#:

string/=  

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

zl:string# string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2 Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

idx1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

idx2 Specifies the position within string2 from which to begin the comparison. Default is 0.

lim1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

lim2 Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.

Examples:
The following functions are synonyms of `zl:string≠`:

```lisp
string≠
user::string
```

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

```lisp
string≤ string1 string2 &key (:start1 0) :end1 (:start2 0) :end2
```

**Function**

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.

`string≤` returns `nil` unless `string1` is less than or equal to `string2`. If the condition is satisfied, `string≤` returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

`string1` and `string2` must be strings, or objects that can be coerced to strings. See the function `string`.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

`:start1` Specifies the position within `string1` from which to begin the comparison (counting from 0). Default is 0, the first character in the string. `:start1` must be ≤ `:end1`.

`:end1` Specifies the position within `string1` of the first character beyond the end of the comparison. Default is `nil`, that is, the operation continues to the end of the string.

`:start2` and `:end2` Work in analogous fashion for `string2`.

The case-insensitive version of `string≤` is the predicate `string-not-greaterp`. 
The following function is a synonym of `string≤`:

\[ \textbf{string} \leq \]

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

\[ \textbf{zl:string} \leq \text{string1 string2} \quad \& \text{optional (idx1 0) (idx2 0) lim1 lim2} \quad \text{Function} \]

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

\begin{itemize}
  \item \textit{idx1} Specifies the position within \text{string1} from which to begin the comparison (counting from 0). Default is 0, the first character in the string.
  \item \textit{idx2} Specifies the position within \text{string2} from which to begin the comparison. Default is 0.
  \item \textit{lim1} Specifies the position within \text{string1} of the first character beyond the end of the comparison. Default is \textit{nil}, that is, the operation continues to the end of the string.
  \item \textit{lim2} Specifies the position within \text{string2} of the first character beyond the end of the comparison. Default is \textit{nil}.
\end{itemize}

Examples:

\[
\begin{align*}
(\text{zl:string} \leq \text{"apple" "apple"}) & \Rightarrow \text{T} \\
(\text{zl:string} \leq \text{"apple" 'apple'}) & \Rightarrow \text{NIL} \\
(\text{zl:string} \leq \text{"sneeze" "snow"}) & \Rightarrow \text{T} \\
(\text{zl:string} \leq \text{"elephant" "aardvark"}) & \Rightarrow \text{NIL} \\
(\text{zl:string} \leq \text{"ZY" "ab"}) & \Rightarrow \text{T} \\
(\text{zl:string} \leq \text{"painting" "interest" 2 0 5}) & \Rightarrow \text{T}
\end{align*}
\]

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

\[ \textbf{string≥} \text{string1 string2 \&key (:start1 0) :end1 (:start2 0) :end2} \quad \text{Function} \]
A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.

`string≥` returns `nil` unless `string1` is greater than or equal to `string2`. If the condition is satisfied, `string≥` returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

`string1` and `string2` must be strings, or objects that can be coerced to strings. See the function `string`.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

- `:start1` Specifies the position within `string1` from which to begin the comparison (counting from 0). Default is 0, the first character in the string. `:start1` must be ≤ `:end1`.
- `:end1` Specifies the position within `string1` of the first character beyond the end of the comparison. Default is `nil`, that is, the operation continues to the end of the string.
- `:start2` and `:end2` Work in analogous fashion for `string2`.

The case-insensitive version of `string≥` is the predicate `string-not-lessp`.

Examples:

```
(string≥ "apple" "apple") => 5
(string≥ "dog" "DOG") => 0
(string≥ "flat" "flush") => NIL
(string≥ "ab" "ZY") => 0
(string≥ "detonate" "unnatural" :start1 4 :start2 2 :end2 5) => 7
(string≥ "dog" "elephant" :start2 3) => NIL
```

The following function is a synonym of `string≥`:

`string>=`

For a table of related items: See the section “Case-Sensitive String Comparison Predicates”.

```
zl:string≥ string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2
```

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.
idx1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

idx2 Specifies the position within string2 from which to begin the comparison. Default is 0.

lim1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

lim2 Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.

Examples:

(zl:string≥ "apple" "apple") => T
(zl:string≥ "dog" "DOG") => T
(zl:string≥ "flat" "flush") => NIL
(zl:string≥ "ab" "ZY") => T
(zl:string≥ "detonate" "unnatural" 4 2 nil 5) => T
(zl:string≥ "dog" "elephant" 0 3) => NIL

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

string/= string1 string2 &key (:start1 0) :end1 (:start2 0) :end2

Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.

string/= returns nil unless string1 is not equal to string2. If the condition is satisfied, user::string//////////= returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

string1 and string2 must be strings, or objects that can be coerced to strings. See the function string.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

:start1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :start1 must be ≤ :end1.

:end1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.
The case-insensitive version of `user::string//=` is the function `string-not-equal`.

Examples:

```
(string/= "apple" "apple") => NIL
(string/= "apple" 'apple) => 0
(string/= "apple" "apply") => 4
(string/= "apple" "apropos") => 2
(string/= "banana" "anachronism" :start1 1 :end1 4) => 3
(string/= "banana" "anachronism" :start1 1 :end1 4 :end2 3) => NIL
```

The following function is a synonym of `user::string//=`:

```
string#
```

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

**Compatibility Note:** In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL.

```
string< string1 string2 &key (:start1 0) :end1 (:start2 0) :end2
```

**Function**

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.

`string<` returns `nil` unless `string1` is less than `string2`. If the condition is satisfied, `string<` returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

`string1` is less than `string2` if the first characters that differ satisfy `char<`, or if `string1` is a proper subset of `string2` (of shorter length and matches in all characters of `string1`).

`string1` and `string2` must be strings, or objects that can be coerced to strings. See the function `string`.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

`:start1` Specifies the position within `string1` from which to begin the comparison (counting from 0). Default is 0, the first character in the string. `:start1` must be ≤ `:end1`. 
Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.

The case-insensitive version of string< is the function string-lessp.

Examples:

(string< "ostrich" "giraffe") => NIL
(string< "demo" "demonstrate") => 4
(string< "abcd" "bazy") => 0
(string< "fred" "Fred") => NIL
(string< "Chicken" "chicken") => 0
(string< "apple" "nap" :start2 1) => NIL
(string< "test" "overestimate" :start1 1 :start2 4) => 5

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL. Note that you cannot use this extension with CLOE.

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

zl:string< string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2

Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

idx1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

idx2 Specifies the position within string2 from which to begin the comparison. Default is 0.

lim1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

lim2 Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.

Examples:
For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

\[\text{string} \leq \text{string1 string2} \ \& \ {\text{key} (\text{start1 0}) (\text{end1 nil}) (\text{start2 0}) (\text{end2 nil})} \]

Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.

\text{string} \leq \text{returns nil unless string1 is less than string2. If the condition is satisfied, string} \leq \text{returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.}

\text{string1 and string2} \text{ must be strings, or objects that can be coerced to strings. See the function string.}

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

\text{:start1} \text{ specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :start1 must be } \leq \text{ :end1.}

\text{:end1} \text{ specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.}

\text{:start2} \text{ and } \text{:end2} \text{ work in analogous fashion for string2.}

The case-insensitive version of \text{string} \leq \text{ is the predicate string-not-greaterp.}

\begin{align*}
\text{(string} \leq \text{ "apple" "apple")} & \Rightarrow 5 \\
\text{(string} \leq \text{ "apple" 'apple')} & \Rightarrow \text{NIL} \\
\text{(string} \leq \text{ "sneeze" "snow")} & \Rightarrow 2 \\
\text{(string} \leq \text{ "elephant" "aardvark")} & \Rightarrow \text{NIL} \\
\text{(string} \leq \text{ "ZY" "ab")} & \Rightarrow 0 \\
\text{(string} \leq \text{ "painting" "interest" :start1 2 :end1 5) } & \Rightarrow 5 \\
\end{align*}

The following function is a synonym of \text{string} \leq : \text{string} \leq

\text{Compatibility Note: In the Genera implementation this function is extended to ac-}
cept character arguments, in addition to the argument types string and symbol, which are specified by CLtL.

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

\[ \text{string} = \text{string1 string2} \ & \text{key} (\text{:start1 0}) :\text{end1} (\text{:start2 0}) :\text{end2} \]

**Function**

Compares two strings or substrings of them, exactly. \( \text{string} = \) returns \( t \) if corresponding characters in the two strings are identical in all character fields, including modifier bits, character set, character style, and alphabetic case; otherwise returns nil.

If the (sub)strings compared are of unequal length, \( \text{string} = \) is false.

\( \text{string1} \) and \( \text{string2} \) must be strings, or objects that can be coerced to strings. See the function \( \text{string} \).

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

- \( :\text{start1} \): Specifies the position within \( \text{string1} \) from which to begin the comparison (counting from 0). Default is 0, the first character in the string. \( :\text{start1} \) must be \( \leq :\text{end1} \).

- \( :\text{end1} \): Specifies the position within \( \text{string1} \) of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

- \( :\text{start2} \) and \( :\text{end2} \): Work in analogous fashion for \( \text{string2} \).

The case-insensitive version of \( \text{string} = \) is the function \( \text{string-equal} \).

Example:

\[
\begin{align*}
(\text{string} = \text{'symbol} \text{"SYMBOL")} & \Rightarrow T \\
(\text{string} = \text{"apple} \text{"orange")} & \Rightarrow \text{NIL} \\
(\text{string} = \text{"apple} \text{"please} :\text{start1} 2 :\text{end2} 3) & \Rightarrow T \\
(\text{string} = \text{"apple} \text{"APPLE")} & \Rightarrow \text{NIL} \\
(\text{string} = \text{"apple} \text{"apply")} & \Rightarrow \text{NIL} \\
(\text{string} = \text{"apple} \text{"applesauce")} & \Rightarrow \text{NIL}
\end{align*}
\]

**Compatibility Note:** In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL. Note that this extension is not available under CLOE.

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".
Function

sys:%string= string1 index1 string2 index2 count

Performs a low-level string comparison, possibly more efficiently than the other comparisons. Its only current efficiency advantages are its simplified arguments and minimized type-checking.

The function compares two strings or substrings of them, exactly. sys:%string= returns t if corresponding characters in the two strings are identical in all character fields, including modifier bits, character set, character style, and alphabetic case; otherwise it returns nil.

If the (sub)strings compared are of unequal length, sys:%string= is false.

string1 and string2 must be strings.

index1 and index2 specify the starting position for the search within string1 and string2 respectively.

count specifies the number of characters to be compared in both strings.

Examples:

```
(sys:%string= "apple" 0 "apple" 0 nil) => T
(sys:%string= "apple" 0 "APPLE" 0 nil) => NIL
(sys:%string= "ccc" 0 "cccc" 0 nil) => NIL
(sys:%string= "ccc" 0 "cccc" 0 3) => T
(sys:%string= "anything" 3 "third" 0 3) => T
(sys:%string= "anything" 3 "third" 1 3) => NIL
(sys:%string= "moooo" 3 (make-array 5
    :element-type 'character
    :initial-element #\o) 3 nil) => T
```

The case-insensitive version of sys:%string= is the function

sys:%string-equal

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

Function

zl:string= string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

idx1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

idx2 Specifies the position within string2 from which to begin the comparison. Default is 0.
lim1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

lim2 Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.

Examples:

(zl:string= 'symbol "SYMBOL") => T
(zl:string= "apple" "orange") => NIL
(zl:string= "apple" "please" 2 0 nil 3) => T
(zl:string= "apple" "APPLE") => NIL
(zl:string= "apple" "apply") => NIL

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

The Common Lisp equivalent to zl:string= is the function:

\[ \text{string=} \]

\[ \text{string> string1 string2 &key (:start1 0) :end1 (:start2 0) :end2} \]

Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.

string> returns nil unless string1 is greater than string2. If the condition is satisfied, string> returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

string1 and string2 must be strings, or objects that can be coerced to strings. See the function string.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

:start1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :start1 must be ≤ :end1.

:end1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.

The case-insensitive version of string> is the predicate string-greaterp.
Examples:

(string> "apple" "apple") => NIL
(string> "true" "TRUE") => 0
(string> "arm" "aim") => 1
(string> "puppet" "puzzle") => NIL
(string> "book" "ball" :start1 1 :start2 2 :end2 3) => 1

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLIL.

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

**zl:string>** string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2

Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx1</td>
<td>Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.</td>
</tr>
<tr>
<td>idx2</td>
<td>Specifies the position within string2 from which to begin the comparison. Default is 0.</td>
</tr>
<tr>
<td>lim1</td>
<td>Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.</td>
</tr>
<tr>
<td>lim2</td>
<td>Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.</td>
</tr>
</tbody>
</table>

Examples:

(zl:string> "apple" "apple") => NIL
(zl:string> "true" "TRUE") => T
(zl:string> "arm" "aim") => T
(zl:string> "puppet" "puzzle") => NIL
(zl:string> "book" "ball" 1 2 nil 3) => T

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

**string>=** string1 string2 &key (:start1 0) :end1 (:start2 0) :end2

Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including modifier bits, character set, character style, and alphabetic case.
string>= returns nil unless string1 is greater than or equal to string2. If the condition is satisfied, string>= returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

string1 and string2 must be strings, or objects that can be coerced to strings. See the function string.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

: start1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. : start1 must be ≤ : end1.

: end1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

: start2 and : end2 Work in analogous fashion for string2.

The case-insensitive version of string>= is the predicate string-not-lessp.

Examples:

(string>= "apple" "apple") => 5
(string>= "dog" "DOG") => 0
(string>= "flat" "flush") => NIL
(string>= "ab" "ZY") => 0
(string>= "detonate" "unnatural" : start1 4 : start2 2 : end2 5) => 7
(string>= "dog" "elephant" : start2 3) => NIL

The following function is a synonym of string>=:

string≥

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL.

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

string-a-or-an string &optional (both-words t) (case : downcase) Function

Computes whether the article "a" or "an" is used when introducing a noun. If both-words is true, the result is the concatenation of the article, a space, and the noun; otherwise, the article is returned. The case argument controls the case of the article. For example:
(string-a-or-an 'rock) => "a ROCK"

(string-a-or-an 'rock t :upcase) => "A ROCK"

(string-a-or-an "egg") => "an egg"

string-append &rest strings Function

Copies and concatenates any number of strings into a single string.

strings are strings or objects that can be coerced to strings. See the function string.

With a single argument, string-append simply copies it.

string-append returns an array of the same type as the argument with the greatest number of bits per element. For example, if the arguments are arrays with elements of type string-char and of type character, an array with elements of type character is returned.

The destructive version of string-append is the function string-nconc.

Example:

(string-append "Hell" "o") => "Hello"
(string-append #\! "foo" #\!) => "!foo!"
(string-append #\! 'foo #\!) => "!FOO!"
(string-append #\1 "2") => "12"
(string-append) => 

(setq string (make-array 5 :element-type 'string-char :initial-contents "hello" :fill-pointer t)) => "hello"
(string-append string " there") => "hello there"
(string-append string #\!) => "hello!"

(setq thin-string (make-string 3)) => "•••"
(setq fat-string (make-array 3 :element-type 'character :initial-element #\A)) => "AAA"
(setq new (string-append thin-string fat-string)) => "•••AAA"
(string-fat-p new) => T

For a table of related items: See the section "String Construction".

string-capitalize string &key (start 0) (end nil) Function

Returns a copy of string; for every word in the copy, the initial character, if case-modifiable, is uppercased. All other case-modifiable characters in the word are lowercased. For the purposes of string-capitalize, a word is defined as a consecutive subsequence of alphanumeric characters or digits, delimited at each end either by a non-alphanumeric character, or by an end of string.
The keywords let you select portions of the string argument for uppercasing. These keyword arguments must be non-negative integer indices into the string array. The result is always the same length as string, however.

:start Specifies the position within string from which to begin uppercasing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end Specifies the position within string of the first character beyond the end of the uppercasing operation. Default is nil, that is, the operation continues to the end of the string.

The destructive version of string-capitalize is the function nstring-capitalize.

Examples:

(string-capitalize "lexington") => "Lexington"
(string-capitalize 'symbol) => "Symbol"
(string-capitalize "one two three" :start 5) => "one tWo Three"
(string-capitalize "a MIxeD-Up sTrinG" :start 2) => "a Mixed-Up String"
(string-capitalize "a MIxeD-Up sTrinG" :start 2 :end 10) => "a Mixed-Up sTrinG"
(string-capitalize "tom&jerry aren't in room 15d")
=> "Tom&Jerry Aren'T In Room 15d"

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtl. Note that you cannot use this extension in CLOE.

For a table of related items: See the section "String Conversion".

string-capitalize-words string &key (start 0) :end Function

Returns a copy of string, such that hyphens are changed to spaces and initial characters of each word are capitalized if they are case-modifiable.

string is a string or a object that can be coerced to a string. See the function string.

The keywords let you select portions of the string argument for uppercasing. These keyword arguments must be non-negative integer indices into the string array. The result is always the same length as string, however.

:start Specifies the position within string from which to begin uppercasing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end Specifies the position within string of the first character beyond the end of the uppercasing operation. Default is nil, that is, the operation continues to the end of the string.

The destructive version of string-capitalize-words is the function nstring-capitalize-words.
Examples:

(string-capitalize-words "string-capitalize-words")
=> "String Capitalize Words"

(string-capitalize-words "three-hyphenated-words" :start 6 :end 8)
=> "three-Hyphenated-words"

For a table of related items: See the section "String Conversion".

zl:string-capitalize-words string &optional (copy-p t) keep-hyphen Function
Changes hyphens to spaces and capitalizes each word in the argument string. The effect on the original argument depends on the value of copy-p: if copy-p is not nil, a copy of string is returned; this is the default; if copy-p is nil, string itself is modified and returned.

If string is not a string, an error is signalled. See the function string.
You can retain hyphens in string by setting keep-hyphen to a non-nil value.

Examples:

(zl:string-capitalize-words "Lisp-listener")
=> "Lisp Listener"

(zl:string-capitalize-words "LISP-LISTENER")
=> "Lisp Listener"

(zl:string-capitalize-words "lisp--listener")
=> "Lisp Listener"

(zl:string-capitalize-words "symbol-processor-3" t t)
=> "Symbol-Processor-3"

(zl:string-capitalize-words "use--some-hyphens" nil)
=> "Use Some Hyphens"

(zl:string-capitalize-words "use--some-hyphens" nil t)
=> "Use Some Hyphens"

The Symbolics Common Lisp equivalent to zl:string-capitalize-words are the functions:

nstring-capitalize-words
string-capitalize-words

For a table of related items: See the section "String Conversion".

string-char Type Specifier
**string-char** is the type specifier symbol for the predefined Lisp string character data type.

The type **string-char** is a *subtype* of the type **character**. Characters that are in the Symbolics standard character set with bits field of zero and style of NIL.NIL.NIL are of type **string-char**.

The type **string-char** is a *supertype* of the type **standard-char**.

Examples:

```lisp
(setq a-string (make-array 3 :element-type 'string-char
    :initial-element #\,)) => ",,"

(typep (char a-string 2) 'string-char) => T

(setq b-string (make-string 9 :initial-element #\.)) => "........."

(typep (char b-string 4) 'string-char) => T

(subtypep 'string-char 'character) => T and T

(subtypep 'standard-char 'string-char) => T and T

(sys:type-arglist 'string-char) => NIL and T

(string-char-p #\g) => T
```

For more information about type specifiers for characters: See the section "Type Specifiers and Type Hierarchy for Characters". See the section "Data Types and Type Specifiers". For a discussion of characters: See the section "Characters". For a discussion of strings: See the section "Strings".

**string-char-p char**

Determines if char can be stored into a thin string (that is, if it is a standard character), returning t if it can, and nil otherwise. Accepts a character argument only. Any character that is a standard character satisfies this test.

Examples:

```lisp
(string-char-p "r") ;signals an error; char must be a character
(string-char-p #\∞) => T
(string-char-p #\meta-m) => NIL

(setq string-var (make-string 10 :initial-element #\m))

(string-char-p (char string-var 4)) => T
```

For a table of related items: See the section "String Type-Checking Predicates". See the section "Character Predicates".
**string-compare** string1 string2 &key (:start1 0) (:start2 0) :end1 :end2  

Function

Compares two strings, or substrings of them. The comparison is case-insensitive, ignoring character style and alphabetic case.

**string-compare** returns:

- a positive number if string1 > string2
- zero if string1 = string2
- a negative number if string1 < string2

If the strings are not equal, the absolute value of the number returned is one more than the index (in string1) at which the difference occurred.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

**:start1**  
Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :start1 must be ≤ :end1. If the value of :start1 is non-zero, the magnitude of the answer is relative to the beginning of string1, not to the beginning of the substring being compared.

**:end1**  
Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

**:start2** and **:end2**  
Work in analogous fashion for string2.

Examples:

```
(string-compare "one" "one") => 0
(string-compare "puppet" "puppet" :start1 3 :start2 3) => 0
(string-compare "puppet" "PUPPET") => 0
(string-compare 'symbol 'foo) => 1
(string-compare "alabaster" " alas!"") => -4
(string-compare "george" "forgery" :start1 2 :start2 1 :end2 5)
=> 0
```

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

The case-sensitive version of **string-compare** is the function:

**string-exact-compare**

### sys:%string-compare** string1 index1 string2 index2 count  

Function

Performs a low-level, case-insensitive string comparison, possibly more efficiently than the other comparisons. Its only current efficiency advantage is its simplified arguments and minimized type-checking.
index1 and index2 specify the starting position for the search within string1 and string2 respectively.

If the value of index1 is non-zero, the sign of the result is meaningful, but the magnitude of the result is not.

count specifies the number of characters to be compared in both strings. If count is nil (unspecified), the entire length of the (sub)strings is compared.

**sys:%string-compare** returns:

- 0 if string1 is equal to string2
- a positive number if string1 > string2
- a negative number if string1 < string2

If the strings are not equal, the absolute value of the number returned is one more than the index in string1 at which the difference occurred.

Examples:

```
(sys:%string-compare "tom" 0 "toM" 0 nil) => 0
(sys:%string-compare "feeding" 3 "dinner" 0 3) => 0
(sys:%string-compare "b" 0 "a" 0 nil) => 1
(sys:%string-compare "a" 0 "b" 0 nil) => -1
(sys:%string-compare "word" 0 "words" 0 nil) => -5
(sys:%string-compare "words" 0 "word" 0 nil) => 5
(sys:%string-compare "...." 0 (make-array 4
:element-type 'character
:initial-element #\.) 0 nil) => 0
```

The case-sensitive version of **sys:%string-compare** is **sys:%string-exact-compare**.

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

**zl:string-compare** string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2    Function

Compares the characters of string1 starting at idx1 and ending just below lim1 with the characters of string2 starting at idx2 and ending just below lim2. The comparison is in alphabetical order. string1 and string2 are strings or objects that can be coerced to strings.

If the value of idx1 is non-zero, the sign of the result is meaningful, but the magnitude of the result is not.

See the function **string**. lim1 and lim2 default to the lengths of the strings. **string-compare** returns:

- a positive number if string1 > string2
- zero if string1 = string2
- a negative number if string1 < string2
If the strings are not equal, the absolute value of the number returned is one more than the index (in string1) at which the difference occurred.

Examples:

(zl:string-compare "one" "one") => 0
(zl:string-compare "puppet" "puppet" 3 3) => 0
(zl:string-compare "puppet" "PUPPET") => 0
(zl:string-compare 'symbol 'foo) => 1
(zl:string-compare "alabaster" " alas!") => -4
(zl:string-compare "abcd" "abce" 1 1) => -3

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

The Symbolics Common Lisp equivalent to zl:string-compare is the function:

\[ \text{string-compare} \]

**string-downcase** \( \text{string} \ \&\text{key} \ (\text{start} \ 0) \ (\text{end} \ \text{nil}) \) \text{ Function}

Returns a copy of string, with uppercase alphabetic characters replaced by the corresponding lowercase characters. (char-downcase is applied to each character of string.)

string is a string or an object that can be coerced to a string.

See the function string.

The keywords let you select portions of the string argument for uppercasing. These keyword arguments must be non-negative integer indices into the string array. The result is always the same length as string, however.

:start Specifies the position within string from which to begin uppercasing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end Specifies the position within string of the first character beyond the end of the uppercasing operation. Default is nil, that is, the operation continues to the end of the string.

Examples:

(string-downcase "A TITLE") => "a title"
(string-downcase "A BUNCH OF WORDS" :start 10) => "A BUNCH OF words"
(string-downcase "A BUNCH OF WORDS" :start 0 :end 1) => "a BUNCH OF WORDS"
(setq string "THREE UPPERCASE WORDS") => "THREE UPPERCASE WORDS"
(string-downcase string :start 0 :end 5 ) => "three UPPERCASE WORDS"
(string-downcase string :start 16 :end nil) => "THREE UPPERCASE words"
string => "THREE UPPERCASE WORDS"

The destructive version of string-downcase is the function nstring-downcase.
**Compatibility Note:** In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by *CLtL*. Note that you cannot use this extension in CLOE.

For a table of related items: See the section "String Conversion".

### `zl:string-downcase string &optional (from 0) to (copy-p t)`

Replaces uppercase alphabetic characters in argument `string` with the corresponding lowercase characters. The effect on the original argument depends on the value of `copy-p`: if `copy-p` is not **nil**, a copy of `string` is returned; if `copy-p` is **nil**, `string` itself is modified and returned.

If `string` is not a string, an error is signalled. See the function `string`.

`from` is the index in `string` at which to begin lowercasing characters. If `to` is supplied, it is used in place of `(array-active-length string)` as the index one greater than the last character to be lowercased.

Examples:

- `(zl:string-downcase "A TITLE") => "a title"
- `(zl:string-downcase "A BUNCH OF WORDS" 10) => "A BUNCH OF words"
- `(zl:string-downcase "A BUNCH OF WORDS" 0 1) => "a BUNCH OF words"
- `(setq string "THREE UPPERCASE WORDS") => "THREE UPPERCASE WORDS"
- `(zl:string-downcase string 0 5 nil) => "three UPPERCASE WORDS"
- `(zl:string-downcase string 16 nil nil) => "three UPPERCASE words"
- `string => "three UPPERCASE words"`

The Common Lisp equivalents to `zl:string-downcase` are the functions:

- `nstring-downcase`
- `string-downcase`

For a table of related items: See the section "String Conversion".

### `string-equal string1 string2 &key (:start1 0) :end1 (:start2 0) :end2`

Compares two strings, or substrings of them. The comparison ignores the character fields for character style and alphabetic case. Two characters are considered to be the same if `char-equal` is true of them.

`string-equal` returns **t** if the strings are the same, and **nil** otherwise. If the (sub)strings compared are of unequal length, `string-equal` is false.

`string1` and `string2` are strings or objects that can be coerced to strings. See the function `string`.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.
Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.\texttt{:start1} must be $\leq$ \texttt{:end1}.

\texttt{:end1} Specifies the position within string1 of the first character beyond the end of the comparison. Default is \texttt{nil}, that is, the operation continues to the end of the string.

\texttt{:start2} and \texttt{:end2} Work in analogous fashion for string2.

The case-sensitive version of \texttt{string-equal} is the predicate \texttt{string=}.

Examples:

\begin{verbatim}
(string-equal 'symbol "SYMBOL") => T
(string-equal "apple" "orange") => NIL
(string-equal "apple" "please" :start1 2 :end2 3) => T
(string-equal "apple" "APPLE") => T
(string-equal "apple" "apply") => NIL
\end{verbatim}

\textbf{Compatibility Note:} In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by \texttt{CLtL}. Note that you can not use this extension with CLOE.

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

ds\texttt{ys}\%\texttt{string-equal} \texttt{string1} \texttt{index1} \texttt{string2} \texttt{index2} \texttt{count} Function

Performs a low-level, case-insensitive string comparison, possibly more efficiently than the other comparisons. Its only current efficiency advantage is its simplified arguments and minimized type-checking. \texttt{sys}%\texttt{string-equal} returns \texttt{T} if the \texttt{count} characters of \texttt{string1} starting at \texttt{idx1} are \texttt{char-equal} to the \texttt{count} characters of \texttt{string2} starting at \texttt{idx2}, or \texttt{NIL} if the characters are not equal or if \texttt{count} runs off the length of either array.

Instead of an integer, \texttt{count} can also be \texttt{NIL}. In this case, \texttt{sys}%\texttt{string-equal} compares the substring from \texttt{idx1} to (\texttt{string-length string1}) against the substring from \texttt{idx2} to (\texttt{string-length string2}). If the lengths of these substrings differ, then they are not equal and \texttt{NIL} is returned.

Note that \texttt{string1} and \texttt{string2} must really be strings; the usual coercion of symbols and characters to strings is not performed. This function is documented because certain programs that require high efficiency and are willing to pay the price of less generality might want to use \texttt{sys}%\texttt{string-equal} in place of \texttt{string-equal}.

Examples:

To compare the two strings "hat" and "hat":

\begin{verbatim}
(sys\%string-equal "hat" 0 "hat" 0 nil) => T
\end{verbatim}

To see if the string "Dante" starts with the characters "dan":

\begin{verbatim}

The case-sensitive version of `sys:%string-equal` is the function:

`sys:%string-equal` = 

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

**zl:string-equal** string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2

Function

Compares two strings, returning `t` if they are equal and `nil` if they are not. The comparison ignores character fields for character style and alphabetic case.

`zl:equal` calls `zl:string-equal` if applied to two strings. `string1` and `string2` are strings or objects that can be coerced to strings. See the function `string`.

The optional arguments let you specify substrings of the two string arguments for comparison.

- `idx1` Specifies the position within `string1` from which to begin the comparison (counting from 0). Default is 0, the first character in the string.
- `idx2` Specifies the position within `string2` from which to begin the comparison. Default is 0.
- `lim1` Specifies the position within `string1` of the first character beyond the end of the comparison. Default is `nil`, that is, the operation continues to the end of the string.
- `lim2` Specifies the position within `string2` of the first character beyond the end of the comparison. Default is `nil`.

Examples:

```
(zl:string-equal "Foo" "foo") => T
(zl:string-equal "foo" "bar") => NIL
(zl:string-equal "element" "select" 0 1 3 4) => T
(zl:string-equal 'symbol "SYMBOL") => T
(zl:string-equal "apple" "orange") => NIL
(zl:string-equal "apple" "please" 2 0 nil 3) => T
(zl:string-equal "apple" "APPLE") => T
(zl:string-equal "apple" "apply") => NIL
```

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

The Common Lisp equivalent to `zl:string-equal` is the function:

`string-equal`
Function

A comparison predicate that compares two strings or substrings of them, exactly including the character fields for character style and alphabetic case.

string-exact-compare returns:

- a positive number if string1 > string2
- zero if string1 = string2
- a negative number if string1 < string2

If the strings are not equal, the absolute value of the number returned is one more than the index (in string1) at which the difference occurred.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

:start1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :start1 must be ≤ :end1. If the value of :start1 is non-zero, the magnitude of the answer is relative to the beginning of string1, not to the beginning of the substring being compared.

:end1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.

Examples:

(string-exact-compare "aaa" "aaa") => 0

(string-exact-compare "yo" "YO") => 1

(string-exact-compare "this is it" "This Is it") => 1

(setq fat-string (make-string 3 :initial-element #\k
 :element-type 'character)) => "kkk"

(string-exact-compare fat-string "kkk") => 0

(string-exact-compare fat-string "asdjf") => 1

(string-exact-compare #\d "mmmm..") => -1

The case-insensitive version of string-exact-compare is the predicate:

string-compare
For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

**sys:%string-exact-compare** string1 index1 string2 index2 count  

Function

Performs a low-level string comparison, possibly more efficiently than the other comparisons. Its only current efficiency advantage is its simplified arguments and minimized type-checking.

**sys:%string-exact-compare** returns:

- a positive number if string1 > string2
- zero if string1 = string2
- a negative number if string1 < string2

string1 and string2 must be strings.

index1 and index2 specify the starting position for the search within string1 and string2 respectively.

If the value of index1 is non-zero, the sign of the result is meaningful, but the magnitude of the result is not.

count specifies the number of characters to be compared in both strings.

Examples:

```
(sys:%string-exact-compare "apple" 0 "apple" 0 nil) => 0
(sys:%string-exact-compare "apple" 0 "APPLE" 0 nil) => 1
(sys:%string-exact-compare "orange" 0 "organ" 0 nil) => -3
(sys:%string-exact-compare "orange" 1 "organ" 0 3) => 1
(sys:%string-exact-compare "hello" 1 "yelp!" 1 2) => 0
(sys:%string-exact-compare "hello" 1 "yelp!" 1 3) => -3
(sys:%string-exact-compare "aaaa" 0 (make-array 4  
:element-type 'character  
:initial-element #\a) 0 nil) => 0
```

The case-insensitive version of **sys:%string-exact-compare** is the function **sys:%string-compare**.

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

**zl:string-exact-compare** string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2  

Function

A comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including character style and alphabetic case.

**zl:string-exact-compare** returns:

- a positive number if string1 > string2
• zero if \textit{string1} = \textit{string2}
• a negative number if \textit{string1} < \textit{string2}

\textit{string1} and \textit{string2} must be strings, or objects that can be coerced to strings. See the function \textit{string}.

The optional arguments let you specify substrings of the two string arguments for comparison.

\begin{itemize}
  \item \texttt{idx1} Specifies the position within \textit{string1} from which to begin the comparison (counting from 0). Default is 0, the first character in the string. If the value of \texttt{idx1} is non-zero, the sign of the result is meaningful, but the magnitude of the result is not.
  \item \texttt{idx2} Specifies the position within \textit{string2} from which to begin the comparison. Default is 0.
  \item \texttt{lim1} Specifies the position within \textit{string1} of the first character beyond the end of the comparison. Default is \texttt{nil}, that is, the operation continues to the end of the string.
  \item \texttt{lim2} Specifies the position within \textit{string2} of the first character beyond the end of the comparison. Default is \texttt{nil}.
\end{itemize}

Examples:
\begin{verbatim}
(zl:string-exact-compare "apple" "apple") => 0
(zl:string-exact-compare "APPLE" "apple") => -1
(zl:string-exact-compare "orange" "organ") => -3
(zl:string-exact-compare "airplane" "aardvark") => 2
(zl:string-exact-compare "baseball" "seven" 2) => -3
(zl:string-exact-compare "flight" "salient" 1 2 nil 5) => 3
\end{verbatim}

For a table of related items: See the section "Case-Sensitive String Comparison Predicates".

\textbf{string-fat-p \textit{string}}

\textbf{Function}

Determines if \textit{string} is an array of fat characters, returning \texttt{T} if it is, and \texttt{NIL} otherwise. Accepts a string argument only. Array-elements of type \texttt{character} are wider characters with bits holding information about modifier bits, character set, and character style.

It is an error if the argument is not a string.

Examples:
\begin{verbatim}
(string-fat-p "string") => NIL
\end{verbatim}
\begin{verbatim}
(string-fat-p "string") => T
\end{verbatim}
(string-fat-p (string-append "fred" \meta-q)) => T

(string-fat-p (make-string 3 :initial-element \hyper-super-a)) => T

(string-fat-p (make-string 3 :element-type 'character)) => T

(string-fat-p (make-array 4 :element-type 'character
 :initial-element \a)) => T

(string-fat-p 4) => NIL

For a table of related items: See the section "String Type-Checking Predicates".

**string-flipcase**

`string &key (start 0) (end nil)`

*Function*

Returns a copy of `string`, with uppercase alphabetic characters replaced by the corresponding lowercase characters, and with lowercase alphabetic characters replaced by the corresponding uppercase characters.

`string` is a string or an object that can be coerced to a string. See the function `string`.

The keywords let you select portions of the string argument for case changing. These keyword arguments must be non-negative integer indices into the string array. The result is always the same length as `string`, however.

**:start** Specifies the position within `string` from which to begin case changing (counting from 0). Default is 0, the first character in the string. **:start** must be ≤ **:end**.

**:end** Specifies the position within `string` of the first character beyond the end of the case changing operation. Default is `nil`, that is, the operation continues to the end of the string.

**Examples:**

```
(string-flipcase "a sTrANGe UsE OF CaPitalS")
=> "A StRangE uSe of cApITALs"

(string-flipcase 'symbol) => "symbol"
(string-flipcase 'symbol :start 2 :end 4) => "SYMBoL"
(string-flipcase "End" :start 2) => "EnD"
(string-flipcase "STRing") => "strING"
```

The destructive version of **string-flipcase** is the function:

**nstring-flipcase**

For a table of related items: See the section "String Conversion".
**zl:string-flipcase**  
*string* &optional (from 0) to (copy-p t)  

The `zl:string-flipcase` function reverses the alphabetic case in its argument: it changes uppercase alphabetic characters to lowercase and lowercase characters to uppercase. The effect on the original argument depends on the value of `copy-p`: if `copy-p` is not `nil`, a copy of `string` is returned; this is the default; if `copy-p` is `nil`, `string` itself is modified and returned.

If `string` is not a string, an error is signalled. See the function `string`.

*from* is the index in `string` at which to begin exchanging the case of characters. If *to* is supplied, it is used in place of `(array-active-length string)` as the index one greater than the last character whose case is to be exchanged.

**Examples:**

```lisp
(zl:string-flipcase "small LARGE") => "SMALL large"
(zl:string-flipcase "small LARGE" 6) => "small large"
(zl:string-flipcase "small LARGE" 1 3) => "sMAll LARGE"
(setq string "STRING") => "STRING"
(zl:string-flipcase string 0 nil nil) => "STRING"
(zl:string-flipcase string 0 nil nil) => "STRING"
```

The Symbolics Common Lisp equivalents to `zl:string-flipcase` are the functions:

- `string-flipcase`
- `nstring-flipcase`

For a table of related items: See the section "String Conversion".

**string-greaterp**  
*string1* *string2* &key (:start1 0) :end1 (:start2 0) :end2  

A comparison predicate that compares two strings, or substrings of them. The comparison ignores character fields for character style and alphabetic case.

`string-greaterp` returns `nil` unless `string1` is greater than `string2`. If the condition is satisfied, `string-greaterp` returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

`string1` and `string2` must be strings, or objects that can be coerced to strings. See the function `string`.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

- **:start1**  
  Specifies the position within `string1` from which to begin the comparison (counting from 0). Default is 0, the first character in the string. **:start1** must be ≤ **:end1**.

- **:end1**  
  Specifies the position within `string1` at which to end the comparison (counting from 0). If not supplied, it defaults to the length of `string1`.

- **:start2**  
  Specifies the position within `string2` from which to begin the comparison (counting from 0).

- **:end2**  
  Specifies the position within `string2` at which to end the comparison (counting from 0). If not supplied, it defaults to the length of `string2`.

Examples:

```lisp
(string-greaterp "aaa" "bb") => 0
(string-greaterp "aaa" "cb") => 0
(string-greaterp "aaa" "cba") => 0
(string-greaterp "aaa" "aaa") => nil
(string-greaterp "aaa" "aaa") => nil
```

The Symbolics Common Lisp equivalents to `string-greaterp` are the functions:

- `string-greaterp`
- `nstring-greaterp`
:end1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.

The case-sensitive version of string-greaterp is the predicate string>. Examples:

```
(string-greaterp "apple" "apple") => NIL
(string-greaterp "true" "TRUE") => NIL
(string-greaterp "arm" "aim") => 1
(string-greaterp "puppet" "puzzle") => NIL
(string-greaterp "book" "ball" :start1 1 :start2 2 :end2 3) => 1
```

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLIL.

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

zl:string-greaterp string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2 Function

Compares two strings or substrings of them. The comparison ignores the character fields for character style and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

idx1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

idx2 Specifies the position within string2 from which to begin the comparison. Default is 0.

lim1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

lim2 Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.

Examples:

```
(zl:string-greaterp "apple" "apple") => NIL
(zl:string-greaterp "true" "TRUE") => NIL
(zl:string-greaterp "arm" "aim") => T
(zl:string-greaterp "puppet" "puzzle") => NIL
(zl:string-greaterp "book" "ball" 1 2 0 3) => T
```
For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

`.string-in` `eof-option` `vector` &optional `(start 0)` `end`  

Message

Reads characters from an input stream into `vector`, using the sub-vector delimited by `start` and `end`.

`start` defaults to 0 and `end` defaults to the length of the vector. The difference between `end` and `start` constitutes a character count for this operation.

`eof-option` specifies stopping actions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nil</code></td>
<td>Reading characters into the vector stops either when it has transferred the specified character count or when it reaches end-of-file, whichever happens first. For vectors with a fill pointer, it sets the fill pointer to point to the location following the last one filled by the read.</td>
</tr>
<tr>
<td><code>not nil</code></td>
<td>If the end-of-file is encountered while trying to transfer a specific number of characters, it signals <code>sys:end-of-file</code>, with the value of <code>eof</code> as the report string.</td>
</tr>
</tbody>
</table>

`.string-in` accepts a string for some input streams, and an array for others.

`.string-in` returns two values. The first value is one greater than the last location of `vector` into which it stored a character. The second value is `t` if it reached end-of-file and `nil` if it did not. Using `.string-in` at the end of a file returns `0` and `t` and sets the fill pointer of `vector` to `start` (if `vector` has a fill pointer).

For example, suppose the file `my-host:~george:~tiny.text` contains "Here is some tiny text.".

```lisp
(setq string (make-array 100 :element-type 'string-char))"

(with-open-file (stream "my-host:~george:~tiny.text")
  (send stream ':string-in nil string))
23

string => "Here is some tiny text."
```

If `vector` has an array-leader, the fill pointer is adjusted to `start` plus the number of characters stored into `vector`.

`vector` can be any type of vector that will hold the elements being read from the stream.

The `.string-in` message can be sent to windows. It interacts correctly with the input editor, including correct handling of activation characters.
The interface to this method for windows and the returned value is exactly the same as the equivalent methods for `si:input-stream` and `si:unbuffered-line-input-stream`.

**string-left-trim char-set string**  
*Function*

Strips the characters in *char-set* of the beginning of *string*. Returns a substring of *string*. Under CLOE, if no characters require trimming, *string* is returned rather than a copy.

*string* is a string or an object that can be coerced to a string. See the function *string*.

*char-set* is a set of characters that can be represented as a list of characters, an array of characters, or a string of characters.

Examples:

```
(string-left-trim '(#\p) "pop") => "op"
(string-left-trim #(#\sp) " spaces ") => " spaces ">
(string-left-trim "atn" "attack at dawn") => "ck at dawn"
(string-left-trim "abcxyz" "abcdefg...uvwxyz") => "defg...uvwxyz"
(string-left-trim (vector #\Newline #\Space) " a b c ") => "a b c ">
```

**Compatibility Note:** In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by *CllL*.

For a table of related items: See the section "String Manipulation".

**zl:string-left-trim char-set string**  
*Function*

Strips the characters in *char-set* off the beginning of *string*. Returns a substring of *string*.

*string* is a string or an object that can be coerced to a string. See the function *string*.

*char-set* is a set of characters that can be represented as a list of characters, an array of characters, or a string of characters.

Examples:

```
(zl:string-left-trim '(#\p) "pop") => "op"
(zl:string-left-trim "atn" "attack at dawn") => "ck at dawn"
```

The Common Lisp equivalent to `zl:string-left-trim` is the function:  

**string-left-trim**

For a table of related items: See the section "String Manipulation".
**string-length** string

Function

Returns the number of characters in **string**. **string** must be a string or an object that can be coerced into a string. See the function **string**.

**string-length** returns the **zl:array-active-length** if **string** is a string, or the **zl:array-active-length** of the print name if **string** is a symbol.

Examples:

```
(string-length "mississippi") => 11
(string-length 'alabama) => 7
(string-length
    (make-array 10 :element-type 'string-char :fill-pointer 7)) => 7
(string-length #\4) => 1
```

For a table of related items: See the section "String Access and Information".

**string-lessp** string1 string2 &key (:start1 0) (:end1 (:start2 0) :end2

Function

Compares two strings, or substrings of them. The comparison ignores character fields for character style and alphabetic case.

**string-lessp** returns **nil** unless **string1** is less than **string2**. If the condition is satisfied, **string-lessp** returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

**string1** is less than **string2** if the first characters that differ satisfy **char-lessp**, or if **string1** is a proper subset of **string2** (of shorter length and matches in all characters of **string1**).

**string1** and **string2** must be strings, or objects that can be coerced to strings. See the function **string**.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

**:start1**

Specifies the position within **string1** from which to begin the comparison (counting from 0). Default is 0, the first character in the string. **:start1** must be ≤ **:end1**.

**:end1**

Specifies the position within **string1** of the first character beyond the end of the comparison. Default is **nil**, that is, the operation continues to the end of the string.

**:start2** and **:end2**

Work in analogous fashion for **string2**.

The case-sensitive version of **string-lessp** is the predicate **string<**.

Examples:
(string-lessp "ostrich" "giraffe") => NIL
(string-lessp "demo" "demonstrate") => 4
(string-lessp "abcd" "bazy") => 0
(string-lessp "fred" "Fred") => NIL
(string-lessp "Chicken" "chicken") => NIL
(string-lessp "apple" "nap" :start2 1) => NIL
(string-lessp "test" "overestimate" :start1 1 :start2 4) => 5

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL.

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

zl:string-lessp string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2 Function

Compares two strings using alphabetical order (as defined by char-lessp). The result is t if string1 is the lesser, or nil if they are equal or string2 is the lesser.

The optional arguments let you specify substrings of the two string arguments for comparison.

idx1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

idx2 Specifies the position within string2 from which to begin the comparison. Default is 0.

lim1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

lim2 Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.

Examples:

(zl:string-lessp "ostrich" "giraffe") => NIL
(zl:string-lessp "demo" "demonstrate") => T
(zl:string-lessp "abcd" "bazy") => T
(zl:string-lessp "fred" "Fred") => NIL
(zl:string-lessp "Chicken" "chicken") => NIL
(zl:string-lessp "apple" "nap" :start2 1) => NIL
(zl:string-lessp "test" "overestimate" :start1 1 :start2 4) => T

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

:string-line-in eof string &optional (start 0) end Message
A combination of :string-in and :line-in that reads many lines successively into the same buffer without creating strings. :string-line-in reads a line from a file into a string (or other array) supplied by the user. It returns the array index plus one, whether an eof was encountered and whether the entire line was read into the buffer.

This message fills up a string as does :string-in, but reads only one line, as does :line-in. As with :line-in, the carriage return character at the end of the line is not stored into your buffer. :line-in reads a line from a stream and creates a string with that line in it. :string-line-in is given a string; it fills in the string (or other array) that you give it from the stream.

 :string-line-in reads a line from a stream and fills the supplied array with that line. As with :string-in, if the string (or other array) has a fill pointer, it is set to the number of characters placed into the buffer plus the start offset.

 :string-line-in returns three values:

• The number of active characters in the string or array. The number is calculated as one plus the array index into the buffer of the last item added to the string by this call.

• Whether the end of the input stream was encountered while trying to read in the string. eof is identical to the eof-option argument in :string-in.

• nil if the entire line fit in the buffer supplied, otherwise t. If t is returned for this value, as much of the line as could fit was stored in the buffer and more of the line is waiting to be read.

If the second and third values are both nil, a carriage return was read. If either is t, no carriage return was read from the stream.

**string-nconc modified-string &rest strings**

The destructive version of string-append. Instead of making a new string containing the concatenation of its arguments, string-nconc modifies its first argument. modified-string must be a string with a fill-pointer so that additional characters can be tacked onto it.

The value of string-nconc is modified-string or a new, longer copy of it if the strings don’t fit; in the latter case the original copy is forwarded to the new copy.

If string is not a string, an error is signalled. See the function adjust-array.

Unlike nconc, string-nconc with more than two arguments modifies only its first argument, not every argument but the last.

Examples:
(setq string (make-array 5 :element-type 'string-char
  :initial-contents "hello" :fill-pointer 5)) => "hello"
(string-nconc string " there") => "hello there"
(string-nconc string #\!) => "hello there!"
string => "hello there!"

For a table of related items: See the section "String Construction".

zl:string-nconc to-string &rest strings

Function

Like string-append, except that instead of making a new string containing the
concatenation of its arguments, zl:string-nconc modifies its first argument.
to-string must be a string with a fill-pointer so that additional characters can be
tacked onto it. See the function zl:array-push-extend.
The value of zl:string-nconc is to-string or a new, longer copy of it; in the latter
case the original copy is forwarded to the new copy. See the function zl:adjust-
array-size.

Unlike nconc, zl:string-nconc with more than two arguments modifies only its
first argument, not every argument but the last.
The Symbolics Common Lisp equivalent to zl:string-nconc is the function:

  string-nconc

For a table of related items: See the section "String Construction".

string-nconc-portion to-string (from-string from to) ...

Function

Adds information onto a string without allocating intermediate substrings.
to-string must be a string with a fill-pointer so that additional characters can be
added onto it. The remaining arguments can be any number of "string portion
specs", which are string/from/to triples. from and to are required but can be nil
and nil. Even though the arguments are called strings, they can be anything that
can be coerced to a string with string (for example, symbols or characters).
The value of string-nconc-portion is to-string or a new, longer copy of it; in the
latter case the original copy is forwarded to the new copy (see
zl:adjust-array-size).

string-nconc-portion is like string-nconc except that it takes parts of strings
without consing substrings.

Example:

  (let ((a (make-array 10 :element-type 'string-char :fill-pointer 0)))
    (zl:string-nconc-portion a 'xxxfoobar 3 nil
      #\sp nil nil
      "tempstuff" 0 4)) => "FOOBAR temp"

string-nconc-portion uses zl:array-push-portion-extend internally, which uses
zl:adjust-array-size to take care of growing the to-string if necessary.
For a table of related items: See the section "String Construction".

**string-not-equal** string1 string2 &key (:start1 0) :end1 (:start2 0) :end2

**Function**

Compares two strings, or substrings of them. The comparison ignores character fields for character style and alphabetic case.

*string-not-equal* returns *nil* unless *string1* is not equal to *string2*. If the condition is satisfied, *string-not-equal* returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

*string1* and *string2* must be strings, or objects that can be coerced to strings. See the function *string*.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

:***start1***

Specifies the position within *string1* from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :***start1*** must be ≤ :***end1***.

:***end1***

Specifies the position within *string1* of the first character beyond the end of the comparison. Default is *nil*, that is, the operation continues to the end of the string.

:***start2*** and :***end2***

Work in analogous fashion for *string2*.

The case-sensitive version of *string-not-equal* is the predicate *string*≠.

**Examples:**

```
(string-not-equal "apple" "apple") => NIL
(string-not-equal "apple" 'apple) => NIL
(string-not-equal "apple" "apply") => 4
(string-not-equal "apple" "apropos") => 2
(string-not-equal "banana" "anachronism" :start1 1 :end1 4) => 3
(string-not-equal "banana" "anachronism" :start1 1 :end1 4 :end2 3) => NIL
```

**Compatibility Note:** In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by *CLtL*.

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

**zl:string-not-equal** string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2

**Function**
Compares two strings or substrings of them. The comparison ignores character fields for character style and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

$idx1$ Specifies the position within $string1$ from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

$idx2$ Specifies the position within $string2$ from which to begin the comparison. Default is 0.

$lim1$ Specifies the position within $string1$ of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

$lim2$ Specifies the position within $string2$ of the first character beyond the end of the comparison. Default is nil.

Examples:

$$
(zl:string-not-equal "apple" "apple") \Rightarrow \text{NIL}
(zl:string-not-equal "apple" 'apple) \Rightarrow \text{NIL}
(zl:string-not-equal "apple" "apply") \Rightarrow \text{T}
(zl:string-not-equal "apple" "apropos") \Rightarrow \text{T}
(zl:string-not-equal "banana" "anachronism" 1 0 4) \Rightarrow \text{T}
(zl:string-not-equal "banana" "anachronism" 1 0 4 3) \Rightarrow \text{NIL}
$$

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

string-not-greaterp string1 string2 &key (:start1 0) :end1 (:start2 0) :end2

Function

A comparison predicate that compares two strings, or substrings of them. The comparison ignores character fields for character style and alphabetic case.

string-not-greaterp returns nil unless $string1$ is less than or equal to $string2$. If the condition is satisfied, string-not-greaterp returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

$string1$ and $string2$ must be strings, or objects that can be coerced to strings. See the function string.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

$\text{:start1}$ Specifies the position within $string1$ from which to begin the comparison (counting from 0). Default is 0, the first character in the string. $\text{:start1}$ must be $\leq \text{:end1}$. 
Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.

The case-sensitive version of string-not-greaterp is the predicate string≤. Examples:

- (string-not-greaterp "apple" "apple") => 5
- (string-not-greaterp "apple" 'apple) => 5
- (string-not-greaterp "sneeze" "snow") => 2
- (string-not-greaterp "elephant" "aardvark") => NIL
- (string-not-greaterp "ZY" "ab") => NIL
- (string-not-greaterp "painting" "interest" :start1 2 :end1 5) => 5

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL.

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

zl:string-not-greaterp string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2

Function

Compares two strings or substrings of them. The comparison ignores character fields for character style and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

 idx1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string.

 idx2 Specifies the position within string2 from which to begin the comparison. Default is 0.

 lim1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

 lim2 Specifies the position within string2 of the first character beyond the end of the comparison. Default is nil.

Examples:
(zl:string-not-greaterp "apple" "apple") => T
(zl:string-not-greaterp "apple" 'apple) => T
(zl:string-not-greaterp "sneeze" "snow") => T
(zl:string-not-greaterp "elephant" "aardvark") => NIL
(zl:string-not-greaterp "ZY" "ab") => NIL
(zl:string-not-greaterp "painting" "interest" 2 0 5) => T

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

string-not-lessp string1 string2 &key (:start1 0) :end1 (:start2 0) :end2 Function

A comparison predicate that compares two strings, or substrings of them. The comparison ignores character fields for character style and alphabetic case.

string-not-lessp returns nil unless string1 is greater than or equal to string2. If the condition is satisfied, string-not-lessp returns the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

string1 and string2 must be strings, or objects that can be coerced to strings. See the function string.

The keywords let you specify substrings of the two string arguments for comparison. These keyword arguments must be non-negative integer indices into the string array.

:start1 Specifies the position within string1 from which to begin the comparison (counting from 0). Default is 0, the first character in the string. :start1 must be ≤ :end1.

:end1 Specifies the position within string1 of the first character beyond the end of the comparison. Default is nil, that is, the operation continues to the end of the string.

:start2 and :end2 Work in analogous fashion for string2.

The case-sensitive version of string-not-lessp is the predicate string≥.

Examples:

(string-not-lessp "apple" "apple") => 5
(string-not-lessp "dog" "DOG") => 3
(string-not-lessp "flat" "flush") => NIL
(string-not-lessp "ab" "ZY") => NIL
(string-not-lessp "detonate" "unnatural" :start1 4 :start2 2 :end2 5) => 7
(string-not-lessp "dog" "elephant" :start2 3) => NIL

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL.
For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

**zl:string-not-lessp string1 string2 &optional (idx1 0) (idx2 0) lim1 lim2**  
*Function*

A comparison predicate that compares two strings, or substrings of them. The comparison ignores character fields for character style and alphabetic case.

The optional arguments let you specify substrings of the two string arguments for comparison.

- **idx1**: Specifies the position within `string1` from which to begin the comparison (counting from 0). Default is 0, the first character in the string.
- **idx2**: Specifies the position within `string2` from which to begin the comparison. Default is 0.
- **lim1**: Specifies the position within `string1` of the first character beyond the end of the comparison. Default is `nil`, that is, the operation continues to the end of the string.
- **lim2**: Specifies the position within `string2` of the first character beyond the end of the comparison. Default is `nil`.

**Examples:**

- `(zl:string-not-lessp "apple" "apple") => T`
- `(zl:string-not-lessp "dog" "DOG") => T`
- `(zl:string-not-lessp "flat" "flush") => NIL`
- `(zl:string-not-lessp "ab" "ZY") => NIL`
- `(zl:string-not-lessp "detonate" "unnatural" 4 2 0 5) => NIL`
- `(zl:string-not-lessp "dog" "elephant" 0 3) => NIL`

For a table of related items: See the section "Case-Insensitive String Comparison Predicates".

**string-nreverse string &key (start 0) (end nil)**  
*Function*

Returns `string` with the order of characters reversed, modifying the original string, rather than creating a new one. This reverses a one-dimensional array of any type. If `string` is a character, it is simply returned.

`string` is a string, a one-dimensional array, or an object that can be coerced to a string. Since `string-nreverse` is destructive, coercion should be used with care since a string internal to the object might be modified. See the function `string`.

The keywords let you select portions of the string argument for reversing. These keyword arguments must be non-negative integer indices into the string array. The entire argument, `string`, is returned, however.

- **:start**: specifies the position within `string` from which to begin reversing (counting from 0). Default is 0, the first character in the string. **:start** must be ≤ **:end**.
:end specifies the position within string of the first character beyond the end of the reversing operation. Default is nil, that is, the operation continues to the end of the string.

The nondestructive version of string-nreverse is the function string-reverse.

Examples:

```lisp
(setq a "bloom") => "bloom"
(string-nreverse a) => "moolb"
a => "moolb"
(string-nreverse "mysbolics" :start 0 :end 3) => "symbolics"
```

For a table of related items: See the section "String Manipulation".

zl:string-nreverse string

Function

Returns string with the order of characters reversed, modifying the original string, rather than creating a new one. This reverses a one-dimensional array of any type. If string is a character, it is simply returned.

If string is not a string, an error is signalled.

See the function string.

Examples:

```lisp
(zl:string-nreverse 'symbol)
;; signals an error: "illegal to modify the pname of a symbol"
(zl:string-nreverse "word") => "drow"
(setq string "two words") => "two words"
(zl:string-nreverse string) => "sdrow owt"
```

The Symbolics Common Lisp equivalent to zl:string-nreverse is the function:

string-nreverse

For a table of related items: See the section "String Manipulation".

:string-out string &optional start end

Message

Outputs the characters of string successively to the stream. This operation is provided for two reasons: it saves the writing of a frequently used loop, and many streams can perform this operation much more efficiently than the equivalent sequence of :tyo operations. If the stream does not support :string-out itself, the default handler converts it to :tyos.

If start and end are not supplied, the entire string is output. Otherwise a substring is output; start is the index of the first character to be output (defaulting to 0), and end is one greater than the index of the last character to be output (defaulting to the length of the string). Callers need not pass these arguments, but all streams that handle :string-out must check for them and interpret them appropriately.
string-pluralize string

Returns a copy of its string argument containing the plural of the word in string. Any added characters go in the same case as the last character of string.

string is a string or an object that can be coerced to a string. See the function string.

Examples:

(string-pluralize "event") => "events"
(string-pluralize "Man") => "Men"
(string-pluralize "Can") => "Cans"
(string-pluralize "key") => "keys"
(string-pluralize "TRY") => "TRIES"
(string-pluralize 'part) => "PARTS"

For words with multiple plural forms depending on the meaning, string-pluralize cannot always do the right thing.

For a table of related items: See the section "String Conversion".

zl:string-pluralize string

Returns a copy of its string argument containing the plural of the word in string. Any added characters go in the same case as the last character of string.

string is a string or an object that can be coerced to a string. See the function string.

Examples:

(zl:string-pluralize "event") => "events"
(zl:string-pluralize "Man") => "Men"
(zl:string-pluralize "Can") => "Cans"
(zl:string-pluralize "key") => "keys"
(zl:string-pluralize "TRY") => "TRIES"
(zl:string-pluralize "child") => "children"

For words with multiple plural forms depending on the meaning, zl:string-pluralize cannot always do the right thing.

The Symbolics Common Lisp equivalent to zl:string-pluralize is the function:

string-pluralize

string-reverse string &key (start 0) (end nil)

Creates and returns a copy of string with the order of characters reversed. This reverses a one-dimensional array of any type. If string is not a string or another one-dimensional array, it is coerced into a string. See the function string.

The keywords let you select portions of the string argument for reversing. These keyword arguments must be non-negative integer indices into the string array. The entire argument, string, is returned, however.
:start specifies the position within string from which to begin reversing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end specifies the position within string of the first character beyond the end of the reversing operation. Default is nil, that is, the operation continues to the end of the string.

The generic function reverse also works on strings.

The destructive version of string-reverse is string-nreverse.

Examples:

(string-reverse #\a) => "a"
(string-reverse 'symbol) => "LOBMYS"
(string-reverse "a string") => "gnirts a"
(string-reverse "end" :start 1) => "edn"
(string-reverse "start" :end 3) => "atsrt"
(string-reverse "middle" :start 1 :end 5) => "mlldie"

For a table of related items: See the section "String Manipulation".

zl:string-reverse string
Function

Creates and returns a copy of string with the order of characters reversed. This reverses a one-dimensional array of any type. If string is not a string or another one-dimensional array, it signals an error. See the function string.

Examples:

(zl:string-reverse #/a) => "a"
(zl:string-reverse 'symbol) => "LOBMYS"
(zl:string-reverse "a string") => "gnirts a"
(zl:string-reverse "end" 1) ; signals an error

The Symbolics Common Lisp equivalent to zl:string-reverse is the function:

string-reverse
For a table of related items: See the section "String Manipulation".

zl:string-reverse-search key string &optional from (to 0) (key-start 0) key-end
Function

Searches for the string key in the string string, using string-equal to do the comparison. The search proceeds in reverse order, starting from the index one less than from, which defaults to the length of string, and returns the index of the first (leftmost) character of the first instance found, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. The from condition, restated, is that the instance of key found is the rightmost one whose rightmost character is before the from’th character of string. If the to argument is supplied, it limits the extent of the search. string is a string or an object that can be coerced to a string. See the function string. Example:
For a table of related items: See the section "Case-Insensitive String Searches".

**(zl:string-reverse-search-char)**  
*char string &optional from (to 0)*  

Function
Searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character that is **char-equal** to *char*, or **nil** if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search. *string* is a string or an object that can be coerced to a string. See the function **string**. Example:

```
(zl:string-reverse-search-char #/n "banana") => 4
```

For a table of related items: See the section "Case-Insensitive String Searches".

**(zl:string-reverse-search-exact)**  
*key string &optional from (to 0) (key-start 0) key-end*

Function
Searches one string for another, comparing characters exactly and depending on all fields including bits, style, and alphabetic case. Substrings of either argument can be specified.

For a table of related items: See the section "Case-Sensitive String Searches".

**(zl:string-reverse-search-exact-char)**  
*char string &optional from (to 0)*  

Function
Searches a string or a substring for the specified character, starting from the end of the string. In other words, it searches the string for the last occurrence of the specified character. It compares all fields of the character, including bits, style, and alphabetic case. Use the optional *from* argument to end the search at the specified position.

zl:string-reverse-search-exact-char returns:

- The position of the last occurrence of the character if the character is found.
- **nil** if the character is not contained within the string.

For example:

```
(zl:string-reverse-search-exact-char #/a "bbab") => 2
(zl:string-reverse-search-exact-char #/a "bbaba") => 4
(zl:string-reverse-search-exact-char #/a "bbb") => NIL
```
(zl:string-reverse-search-exact-char #/a "bAcBA") => NIL

For a table of related items: See the section "Case-Sensitive String Searches".

zl:string-reverse-search-not-char char string &optional from (to 0) Function

Searches through string in reverse order, starting from the index one less than from, which defaults to the length of string, and returns the index of the first character that is not char-equal to char, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the to argument is supplied, it limits the extent of the search. string is a string or an object that can be coerced to a string. See the function string. Example:

(zl:string-reverse-search-not-char #/a "banana") => 4

For a table of related items: See the section "Case-Insensitive String Searches".

zl:string-reverse-search-not-exact-char char string &optional from (to 0) Function

Searches a string or a substring for occurrences of any character other than the specified character, starting from the end of the string. It compares all fields of the character, including bits, style, and alphabetic case. Use the optional from argument to end the search at the specified position.

zl:string-reverse-search-not-exact-char returns:

- The position of the last occurrence of a character that does not match the specified character.
- nil if the string contains only the specified character.

For example:

(zl:string-reverse-search-not-exact-char #/a "aaa") => nil

(zl:string-reverse-search-not-exact-char #/a "bbab") => 3

(zl:string-reverse-search-not-exact-char #/a "bbaba") => 3

(zl:string-reverse-search-not-exact-char #/a "bbb") => 2

(zl:string-reverse-search-not-exact-char #/a "bAcBA") => 4

For a table of related items: See the section "Case-Sensitive String Searches".

zl:string-reverse-search-not-set char-set string &optional from (to 0) Function
Searches through string in reverse order, starting from the index one less than from, which defaults to the length of string, and returns the index of the first character that is not char-equal to any element of char-set, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the to argument is supplied, it limits the extent of the search. char-set is a set of characters, which can be represented as a list of characters or a string of characters. string is a string or an object that can be coerced to a string. See the function string.

\[
(zl:string-reverse-search-not-set '(#/a #/n) "banana") \rightarrow 0
\]

For a table of related items: See the section "Case-Insensitive String Searches".

\[
\text{zl:string-reverse-search-set} \ char-set \ string \ \&\optional \ from \ (to \ 0) \quad \text{Function}
\]

Searches through string in reverse order, starting from the index one less than from, which defaults to the length of string, and returns the index of the first character that is char-equal to some element of char-set, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the to argument is supplied, it limits the extent of the search. char-set is a set of characters, which can be represented as a list of characters or a string of characters. string is a string or an object that can be coerced to a string. See the function string.

\[
(zl:string-reverse-search-set "ab" "banana") \rightarrow 5
\]

For a table of related items: See the section "Case-Insensitive String Searches".

\[
\text{string-right-trim} \ char-set \ string \quad \text{Function}
\]

Strips the characters in char-set off the end of string. Returns a substring of string. Under CLOE, if no characters require trimming, string is returned, rather than a copy.

string is a string or an object that can be coerced to a string. See the function string.

char-set is a set of characters, that can be represented as a list of characters, an array of characters, or a string of characters.

Examples:

\[
\begin{align*}
& (\text{string-right-trim } '#(\#4) "456454") \rightarrow "45645" \\
& (\text{string-right-trim } '#(#\t #\h) "that tooth") \rightarrow "that too" \\
& (\text{string-right-trim } "o" "otto") \rightarrow "ott"
\end{align*}
\]

Related Functions:

string-trim
string-left-trim

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL. Note that you cannot use this extension in CLOE.
(string-right-trim "abcxyz" "abcdefg...uvwxyz")
=> "abcdefg...uvw"

(string-right-trim (vector #\Newline #\Space) " a b c ")
=> " a b c"

For a table of related items: See the section "String Manipulation".

**zl:string-right-trim char-set string**

Function

Strips the characters in **char-set** from the end of **string**. Returns a substring of **string**.

**string** is a string or an object that can be coerced to a string. See the function **string**.

**char-set** is a set of characters that can be represented as a list of characters or a string of characters.

Examples:

(zl:string-right-trim '(#/4) "456454") => "45645"
(zl:string-right-trim "o" "otto") => "ott"

The Common Lisp equivalent to **zl:string-right-trim** is the function:

**string-right-trim**

For a table of related items: See the section "String Manipulation".

**string-search key string &key :from-end (:start1 0) :end1 (:start2 0) :end2**

Function

Searches **string** looking for occurrences of **key**. The search uses **char-equal** which ignores character fields for character style and alphabetic case.

**string-search** returns **nil**, or the position of the first character of **key** occurring in the (sub)string. To reverse the search, returning the position of the last occurrence of the initial **key** character in the (sub)string searched, specify a non-**nil** value for **:from-end**.

**key** and **string** must be strings, or objects that can be coerced to a string. See the function **string**.

The keywords let you specify the parts of **string** to be searched, as well as the parts of **key** to search for. These keyword arguments must be non-negative integer indices into the string array.

**:from-end**

If a non-**nil** value is specified, returns the position of the first character of the last occurrence of **key** in the string or the specified substring.

**:start1**

Specifies the position within **key** from which to begin the search (counting from 0). Default is 0, the first character in the string. **:start1** must be ≤ **:end1**.
:end1 Specifies the position within key of the first character beyond the end of the search. Default is nil, that is the entire length of key is used.

:start2 and :end2 Work analogously for string.

Examples:

(string-search "es" "witches") => 5
(string-search "es" "tresses") => 2
(string-search "es" "tresses" :from-end t) => 5
(string-search "er" "tresses") => NIL
(string-search "er" "tresses" :from-end t) => NIL
(string-search "es" "tresses" :start2 3) => 5

(string-search #\a "banana") => 1

(string-search 'symbol "abolish" :start1 3) => 1
(string-search 'symbol "abolish" :start1 3 :end2 3) => NIL

The case-sensitive version of string-search is the function:

string-search-exact

For a table of related items: See the section "Case-Insensitive String Searches".

zl:string-search key string &optional (from 0) to (key-start 0) key-end Function

Searches for the string key in the string string, using string-equal to do the comparison. The search begins at from, which defaults to the beginning of string. The value returned is the index of the first character of the first instance of key, or nil if none is found. If the to argument is supplied, it is used in place of (string-length string) to limit the extent of the search. string is a string or an object that can be coerced to a string. See the function string. Example:

(zl:string-search "an" "banana") => 1
(zl:string-search "an" "banana" 2) => 3
(zl:string-search "es" "witches") => 5
(zl:string-search "es" "tresses") => 2
(zl:string-search "er" "tresses") => NIL

The Symbolics Common Lisp equivalent to zl:string-search is the function:

string-search

For a table of related items: See the section "Case-Insensitive String Searches".

string-search-char char string &key :from-end (:start 0) :end Function

Searches string looking for the character char. The search uses char-equal, which ignores the character fields for character style and alphabetic case.
string-search-char returns nil if it does not find char; if successful, it returns the position of the first occurrence of char. To reverse the search, returning the position of the last occurrence of char in the (sub)string searched, set :from-end to t.

char must be a character object.

string must be a string, or an object that can be coerced to a string. See the function string.

The keywords let you specify the parts of string to be searched. These keyword arguments must be non-negative integer indices into the string array.

:from-end If set to a non-nil value, returns the position of the last occurrence of char in the string or the specified sub-string.

:start Specifies the position within string from which to begin the search (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end Specifies the position within string of the first character beyond the end of the search. Default is nil, that is the entire length of string is searched.

Examples:

(string-search #\? "banana") => NIL
(string-search-char #\a "banana") => 1
(string-search-char #\a "banana" :from-end t) => 5
(string-search-char #\a "banana" :start 1 :end 3) => 1
(string-search-char #\a "banana" :start 1 :end 4 :from-end t) => 3
(string-search-char #\A "banana") => 1

The case-sensitive version of string-search-char is the function:

string-search-exact-char

For a table of related items: See the section "Case-Insensitive String Searches".

sys:%string-search-char char string start end

Function

Performs a low-level string search, possibly more efficiently than the other searching functions. Its only current efficiency advantage is its simplified arguments and minimized type-checking.

string must be an array;
char must be a character;
start, and end must be integers.

Except for this lack of type-coercion, and the fact that none of the arguments is optional, sys:%string-search-char is the same as zl:string-search-char. This func-
tion is documented for the benefit of those who require the maximum possible effi-
ciency in string searching.

Examples:

```lisp
(sys:%string-search-char #\a
(make-array 4 :element-type 'character
:initial-element #\a 2 4) => 2
(sys:%string-search-char #\p "zippy" 0 90) => 2
```

For a table of related items: See the section "Case-Insensitive String Searches".

### function: zl:string-search-char

| char string &optional (from 0) to |

Searches through string starting at the index from, which defaults to the begin-
ning, and returns the index of the first character that is char-equal to char, or nil
if none is found. If the to argument is supplied, it is used in place of (string-
length string) to limit the extent of the search. string is a string or an object that
can be coerced to a string. See the function string.

Example:

```lisp
(zl:string-search #\? "banana") => NIL
(zl:string-search-char #\a "banana") => 1
(zl:string-search-char #\a "banana") => 1
(zl:string-search-char #\a "banana" 1 3) => 1
(zl:string-search-char #\a "banana" 1 4 ) => 1
```

The Symbolics Common Lisp equivalent to zl:string-search-char is the function:

### function: string-search-char

For a table of related items: See the section "Case-Insensitive String Searches".

### function: string-search-exact

| key string &key :from-end (start1 0) :end1 (start2 0) :end2 |

Searches string looking for occurrences of key. The search compares all characters
exactly, using all character fields including character style and alphabetic case.

string-search-exact returns nil, or the position of the first character of key occur-
rning in the (sub)string. To reverse the search, returning the position of the last
occurrence of the initial key character in the (sub)string searched, specify a
non-nil value for :from-end.

key and string must be strings, or objects that can be coerced to a string. See the
function string.

The keywords let you specify the parts of string to be searched, as well as the
parts of key to search for. These keyword arguments must be non-negative integer
indices into the string array.
If a non-nil value is specified, returns the position of the first character of the last occurrence of key in the string or the specified substring.

Specifies the position within key from which to begin the search (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

Specifies the position within key of the first character beyond the end of the search. Default is nil, that is the entire length of key is used.

Work analogously for string.

Examples:

```
(setq a-string (make-string 3 :initial-element #\a)) => "aaa"
(zl:string-search-exact #\a a-string) => 0

(zl:string-search-exact #\a "AAA") => NIL

(zl:string-search-exact #\a "bbbabba") => 3

(zl:string-search-exact #\a "aaabAcBA") => 0

(zl:string-search-exact #\a "abbbacccbaaddda" :from-end 2 ) => 13
```

The case-insensitive version of string-search-exact is the function:

```
string-search
```

For a table of related items: See the section "Case-Sensitive String Searches".

```
(zl:string-search-exact #\a #\a a-string) => 0
```

Function

Searches one string for another, comparing characters exactly and depending on all fields including bits, style, and alphabetic case. Substrings of either argument can be specified.

Examples:

```
(setq a-string (make-string 3 :initial-element #\a)) => "aaa"
(zl:string-search-exact #\a a-string) => 0

(zl:string-search-exact #\a "AAA") => NIL

(zl:string-search-exact #\a "bbbabba") => 3

(zl:string-search-exact #\a "aaabAcBA") => 0
```
The Symbolics Common Lisp equivalent to `zl:string-search-exact` is the function:

```lisp
string-search-exact
```

For a table of related items: See the section "Case-Sensitive String Searches".

```lisp
string-search-exact-char char string &key :from-end (:start 0) :end          Function
```

Searches `string` looking for the character, `char`. The search compares all characters exactly, using all character fields including character style and alphabetic case.

`string-search-exact-char` returns `nil` if it does not find `char`; if successful, it returns the position of the first occurrence of `char` in the string or substring searched. To reverse the search returning the position of the *last* occurrence of `char` in the (sub)string searched, specify a non-nil value for the keyword `:from-end`.

`char` must be a character object.

`string` must be a string, or an object that can be coerced to a string. See the function `string`.

The keywords let you specify the parts of `string` to be searched. These keyword arguments must be non-negative integer indices into the string array.

- **:from-end** If set to a non-nil value, returns the position of the *last* occurrence of `char` in the string or the specified substring.
- **:start** Specifies the position within `string` from which to begin the search (counting from 0). Default is 0, the first character in the string. **:start** must be ≤ **:end**.
- **:end** Specifies the position within `string` of the first character beyond the end of the search. Default is `nil`, that is the entire length of `string` is searched.

Examples:

```
(string-search-exact-char #\a "bbab") => 2
(string-search-exact-char #\a "abbaba") => 0
(string-search-exact-char #\a "bbAAaAab") => 4
(string-search-exact-char #\a "bAcBA") => NIL
(string-search-exact-char #\a "abbababba"
   :from-end 2 :start 3 :end 9) => 8
```

The case-insensitive version of `string-search-exact-char` is the function:

```lisp
string-search-char
```
For a table of related items: See the section "Case-Sensitive String Searches".

**sys:%string-search-exact-char** char string start end

*Function*

Performs a low-level string search, possibly more efficient than the other searching functions. Its only current efficiency advantage is its simplified arguments and minimized type-checking.

The function returns **nil** if unsuccessful, or the position in the string of the character sought for. Count starts at zero.

Examples:

```lisp
(sys:%string-search-exact-char #\a (make-array 4 :element-type 'character :initial-element #\a) 0 9) => 0
(sys:%string-search-exact-char #\i "Garfield" 0 6)  => 4
(sys:%string-search-exact-char #\I "Garfield" 0 6)  => NIL
```

For a table of related items: See the section "Case-Sensitive String Searches".

**zl:string-search-exact-char** char string &optional (from 0) to

*Function*

Searches a string or a substring for the specified character, comparing all fields of the character, including, style, and alphabetic case. Use the optional *to* argument to end the search at the specified position.

**zl:string-search-exact-char** returns:

- The position of the first occurrence of the character in the string.
- **nil** if the character is not contained within the string.

For example:

```lisp
(zl:string-search-exact-char #\a "bbab") => 2
(zl:string-search-exact-char #\A "abattoir") => NIL
(zl:string-search-exact-char #\a "abbaba") => 0
(zl:string-search-exact-char #\a "bbAAAabab") => 4
(zl:string-search-exact-char #\meta-A "bAcBA")  => NIL
```

The Symbolics Common Lisp equivalent to **zl:string-search-exact-char** is the func-
string-search-exact-char

For a table of related items: See the section "Case-Sensitive String Searches".

string-search-not-char char string &key :from-end (start 0) :end

Function

Searches string looking for occurrences of any character other than char. The search uses char-equal, which ignores the character fields for character style and alphabetic case.

string-search-not-char returns nil, or the position of the first occurrence of any character that is not char. To reverse the search, returning the position of the last occurrence of a character other than char in the (sub)string searched, specify t for the keyword argument :from-end.

char must be a character object.

string must be a string, or an object that can be coerced to a string. See the function string.

The keywords let you specify the parts of string to be searched. These keyword arguments must be non-negative integer indices into the string array.

:from-end If it has a non-nil value, returns the position of the last occurrence of a character that does not match char in the string or the specified substring.

:start Specifies the position within string from which to begin the search (counting from 0). Default is 0, the first character in the string. :start must be \leq :end.

:end Specifies the position within string of the first character beyond the end of the search. Default is nil, that is the entire length of string is searched.

Examples:

(string-search-not-char #\E "eel") => 2
(string-search-not-char #\l "oscillate") => 0
(string-search-not-char #\l "oscillate" :start 5) => 6
(string-search-not-char #\l "oscillate" :start 5 :from-end t) => 8
(string-search-not-char #\l "oscillate" :start 2 :end 5 :from-end t) => 3

The case-sensitive version of string-search-not-char is the function:

string-search-not-exact-char

For a table of related items: See the section "Case-Insensitive String Searches".

zl:string-search-not-char char string &optional (from 0) to

Function
Searches through string starting at the index from, which defaults to the beginning, and returns the index of the first character which is not \texttt{char-equal} to \texttt{char}, or \texttt{nil} if none is found. If the \texttt{to} argument is supplied, it is used in place of \texttt{(string-length string)} to limit the extent of the search. \texttt{string} is a string or an object that can be coerced to a string. See the function \texttt{string}. Example:

\begin{verbatim}
(zl:string-search-not-char #\b "banana") => 1
(zl:string-search-not-char #\n "banana" 2) => 3
(zl:string-search-not-char #\n "banana" 2 3) => NIL
(zl:string-search-not-char #\E "eel") => 2
(zl:string-search-not-char #\l "oscillate") => 0
(zl:string-search-not-char #\l "oscillate" 5) => 6
(zl:string-search-not-char #\l "oscillate" 2 5 ) => 2
\end{verbatim}

The Symbolics Common Lisp equivalent to \texttt{zl:string-search-not-char} is the function:

\texttt{string-search-not-char}

For a table of related items: See the section "Case-Insensitive String Searches".

\texttt{string-search-not-exact-char} \texttt{char string} \texttt{&key :from-end (start 0) :end}

\textit{Function}

Searches \texttt{string} looking for occurrences of any character other than \texttt{char}. The search compares all characters exactly, using all character fields including character style and alphabetic case.

\texttt{string-search-not-exact-char} returns \texttt{nil}, or the position of the first occurrence of any character that is not \texttt{char}. To reverse the search, returning the position of the last occurrence of a character other than \texttt{char} in the (sub)string searched, specify \texttt{t} for the keyword argument \texttt{:from-end}.

\texttt{char} must be a character object.

\texttt{string} must be a string, or an object that can be coerced to a string. See the function \texttt{string}.

The keywords let you specify the parts of \texttt{string} to be searched. These keyword arguments must be non-negative integer indices into the string array.

\texttt{:from-end} If it has a non-\texttt{nil} value, returns the position of the last occurrence of a character that does not match \texttt{char} in the string or the specified substring.

\texttt{:start} Specifies the position within \texttt{string} from which to begin the search (counting from 0). Default is 0, the first character in the string. \texttt{:start} must be \leq \texttt{:end}.

\texttt{:end} Specifies the position within \texttt{string} of the first character beyond the end of the search. Default is \texttt{nil}, that is the entire length of \texttt{string} is searched.
Examples:

```lisp
(setq a-string (make-string 3 :initial-element #\a)) => "aaa"
(string-search-not-exact-char #\a a-string) => NIL

(string-search-not-exact-char #\a "AAA") => 0

(string-search-not-exact-char #\a "bbba") => 0

(string-search-not-exact-char #\a "aaabAcBA") => 3

(string-search-not-exact-char #\a
    "abbacccaccca" :from-end 3 :start 2 :end 9) => 8
```

The case-insensitive version of `string-search-not-exact-char` is the function:

`string-search-not-char`

For a table of related items: See the section "Case-Sensitive String Searches".

**zl:string-search-not-exact-char char string &optional (from 0) to Function**

Searches a string or a substring for the first occurrence of any character other than the specified character. It compares all fields of the character, including bits, style, and alphabetic case. Use the optional to argument to end the search at the specified position.

`zl:string-search-not-exact-char` returns:

- The position of the first character in the string that does not match the specified character.
- nil if the string contains only the specified character.

For example:

```lisp
(setq a-string (make-string 3 :initial-element #\a)) => "aaa"
(zl:string-search-not-exact-char #\a a-string) => NIL

(zl:string-search-not-exact-char #\a "AAA") => 0

(zl:string-search-not-exact-char #\a "bbba") => 0

(zl:string-search-not-exact-char #\a "aaabAcBA") => 3
```

The Symbolics Common Lisp equivalent to `zl:string-search-not-exact-char` is the function:

`string-search-not-exact-char`

For a table of related items: See the section "Case-Sensitive String Searches".
**string-search-not-set** char-set string &key :from-end (start 0) :end

Function

Searches *string* looking for a character that is not in *char-set*. The search uses *char-equal*, which ignores the character fields for character style and alphabetic case.

*string-search-not-set* returns `nil`, or the position of the first character that is not *char-equal* to some element of the *char-set*. To reverse the search, returning the position of the last occurrence of a character not in *char-set* in the (sub)string searched, specify `t` for the keyword argument :from-end.

*char-set* is a set of characters which can be represented as a list of characters, an array of characters, or a string of characters.

*string* must be a string, or an object that can be coerced to a string. See the function *string*.

The keywords let you specify the parts of *string* to be searched. These keyword arguments must be non-negative integer indices into the string array.

:from-end

If a non-nil value is specified, returns the position of the last occurrence of a character not in *char-set* in the (sub)string searched.

:start

Specifies the position within *string* from which to begin the search (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end

Specifies the position within *string* of the first character beyond the end of the search. Default is nil, that is the entire length of *string* is searched.

Examples:

```
(string-search-not-set #(\a) "aaa") => NIL
(string-search-not-set '(#\h #\i) "hi") => NIL
(string-search-not-set '(#\a) "bcaab") => 0
(string-search-not-set '(#\a #\b #\c) "abcdefabc") => 3
```

For a table of related items: See the section "Case-Insensitive String Searches".

**zl:string-search-not-set** char-set string &optional (from 0) to

Function

Searches through *string* looking for a character that is not in *char-set*. The search begins at the index `from`, which defaults to the beginning. It returns the index of the first character that is not *char-equal* to any element of *char-set*, or nil if none is found. If the `to` argument is supplied, it is used in place of *(string-length *string*)* to limit the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. *string* is a string or an object that can be coerced to a string. See the function *string*. Example:
The Symbolics Common Lisp equivalent to `zl:string-search-not-set` is the function:

```
string-search-not-set
```

For a table of related items: See the section "Case-Insensitive String Searches".

```
string-search-set char-set string &key :from-end (start 0) :end
```

Function

Searches `string` looking for a character that is in `char-set`. The search uses `char-equal`, which ignores the character fields for character style and alphabetic case.

`string-search-set` returns `nil`, or the position of the first character that is `char-equal` to some element of the `char-set`. To reverse the search, returning the position of the last occurrence of the initial character of `char-set` in the (sub)string searched, set `:from-end` to `t`.

`char-set` is a set of characters which can be represented as a list of characters, an array of characters, or a string of characters.

`string` must be a string, or an object that can be coerced to a string. See the function `string`.

The keywords let you specify the parts of `string` to be searched. These keyword arguments must be non-negative integer indices into the string array.

`:from-end` If set to a non-`nil` value, returns the position of the last occurrence of the first character of `char-set` in the string or the specified substring.

`:start` Specifies the position within `string` from which to begin the search (counting from 0). Default is 0, the first character in the string. `:start` must be ≤ `:end`.

`:end` Specifies the position within `string` of the first character beyond the end of the search. Default is `nil`, that is, the entire length of `string` is searched.

Examples:

```
(string-search-set #(#\a) "aaa") => 0
(string-search-set '(#\h #\i) "hi") => 0
(string-search-set #(#\a) "bcaa") => 2
(string-search-set #(#\a #\b #\c) "abcdefabc") => 0
(string-search-set #(#\a #\b #\c #\h) "ping...ahh...haha") => 4
```

For a table of related items: See the section "Case-Insensitive String Searches".
**zl:string-search-set** char-set string &optional (from 0) to

Searches through *string* looking for a character that is in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character that is *char-equal* to some element of *char-set*, or *nil* if none is found. If the *to* argument is supplied, it is used in place of (**string-length** *string*) to limit the extent of the search.

*char-set* is a set of characters, which can be represented as a list of characters or a string of characters.

*string* is a string or an object that can be coerced to a string. See the function **string**. Example:

\[
\begin{align*}
(zl:string-search-set '(\#\h \#\i) "hi") \Rightarrow 0 \\
(zl:string-search-set '(\#\a) "bcaa") \Rightarrow 2 \\
(zl:string-search-set '(\#\a \#\b \#\c) "abcdefabc") \Rightarrow 0
\end{align*}
\]

The Symbolics Common Lisp equivalent to **zl:string-search-set** is the function: **string-search-set**

For a table of related items: See the section "Case-Insensitive String Searches".

**string-thin** string &key (:start 0) :end (:remove-style t) :remove-bits :error-if :area

Strips the specified character-style information and modifier bits from *string*, and returns the resulting substring. (Hyper, meta, super, and control are bits.) *String* is an array of characters. See the function **string**.

:remove-style removes all of the character-style, but not character-set, information. The default is t.

:remove-bits removes all of the bits.

:error-if is either :fat or :bits. If, after the string has been "thinned" there are still fat characters, and if :error-if :fat is specified, an error is signalled. If, after the string has been "thinned" there are still bits, and if :error-if :bit is specified, an error is signalled.

:area is nil, an area, or :stack.

:start specifies the position within *string* from which to begin to remove the character-style information (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end specifies the position within *string* of the first character beyond character beyond the end of the character-style removing operation. Default is *nil*, that is, the operation continues to the end of the string.

Examples:

\[
(setq string *) \Rightarrow "This is a bold string"
\]
(string-thin string :remove-style t) => "This is a bold string"

(string-thin string :start 0 :end 4 :remove-style t) => "This"

For a table of related items: See the section "String Manipulation".

string-to-ascii lispm-string

Function

Converts lispm-string to an sys:art-8b array containing ASCII character codes. See the section "ASCII Characters".

Example:

(string-to-ascii "hello") => #<(ART-8B-5 24443106>)

For a table of related items: See the section "ASCII Conversion String Functions".

string-trim char-set string

Function

Strips the characters char-set off the beginning and end of string, and returns the resulting substring. string itself is not modified. Under CLOE, string is returned (rather than a copy) if no characters need trimming. In Genera, a copy is always returned.

string is a string or an object that can be coerced to a string. char-set is a set of characters, that can be represented as a list of characters, an array of characters, or a string of characters. See the function string.

Examples:

(string-trim '(#\sp) " Dr. No ") => "Dr. No"
(string-trim #(#\a #\b) "abbafooabb") => "foo"
(string-trim "ab" "abbafooabb") => "abbafooabb"
(string-trim "abcxyz" "abcdefg...uvwxyz") => "defg...uvw"

(string-trim (vector #\Newline #\Space) " abc ") => "abc"

(string-trim (list #\Newline #\Space) " abc ") => "abc"

(setq a-str "abcde")

(setf (aref (string-trim "ae" a-str) 1) #\Q) => #\Q
a-str => "abcde"

(setf (aref (string-trim "gh" a-str) 1) #\Q)
=> #\Q

a-str => "aQcde"

Note in the last example that a-str is altered by setf because string-trim returned a-str itself. This behavior is not a guaranteed in the definition of Common Lisp, is subject to change in CLOE and is not true in Genera.

For a table of related items: See the section "String Manipulation".

**Compatibility Note:** In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol, which are specified by CLtL. Note that this extension is not available under CLOE.

---

### zl:string-trim char-set string

*Function*

Strips the characters in char-set off the beginning and end of string, and returns the resulting substring. string itself is not modified.

string is a string or an object that can be coerced to a string. See the function string.

char-set is a set of characters that can be represented as a list of characters, or a string of characters.

Examples:

(zl:string-trim '(#\sp) "    blank    ") => "blank"
(zl:string-trim "ab" "abbafooabb") => "foo"

The Common Lisp equivalent to zl:string-trim is the function:

string-trim

For a table of related items: See the section "String Manipulation".

---

### string-upcase string &key (start 0) (end nil)

*Function*

Returns a copy of string, with lowercase alphabetic characters replaced by the corresponding uppercase characters. (char-upcase is applied to each character of string.)

string is a string or an object that can be coerced to a string. See the function string.

The keywords let you select portions of the string argument for uppecasing. These keyword arguments must be non-negative integer indices into the string array. The result is always the same length as string, however.
:start Specifies the position within string from which to begin uppercasing (counting from 0). Default is 0, the first character in the string. :start must be ≤ :end.

:end Specifies the position within string of the first character beyond the end of the uppercasing operation. Default is nil, that is, the operation continues to the end of the string.

The destructive version string-upcase is the function nstring-upcase.

Examples:

(string-upcase 'fred) => "FRED"
(string-upcase "window") => "WINDOW"
(string-upcase "mixEd-uP") => "MIXED-UP"
(string-upcase "") => ""
(string-upcase "17.'≤αh") => "17.'≤αH"
(string-upcase "end" :start 1) => "eND"
(string-upcase "middle" :start 2 :end 4) => "miDDle"
(zl:string-upcase a 2 4) => "a STring"
(zl:string-upcase a 5 7) => "a strINg"
(zl:string-upcase a 2 4 nil) => "a STring"
(zl:string-upcase a 5 7 nil) => "a STrINg"
(setq a "a string") => "a string"
(string-upcase a :start 2 :end 4) => "a STring"

Compatibility Note: In the Genera implementation this function is extended to accept character arguments, in addition to the argument types string and symbol which are specified by CLtL. Note that you cannot use this extension in CLOE.

For a table of related items: See the section "String Conversion".

zl:string-upcase string &optional (from 0) to (copy-p t) Function

Replaces lowercase alphabetic characters in argument string with the corresponding uppercase characters. The effect on the original argument depends on the value of copy-p: if copy-p is not nil, a copy of string is returned; if copy-p is nil, string itself is modified and returned.

string is a string, or if copy-p is t, an object that can be coerced to a string. See the function string.

from is the index in string at which to begin uppercasing characters. If to is supplied, it is used in place of (array-active-length string) as the index one greater than the last character to be uppercased.

Examples:
The Common Lisp equivalent to \texttt{zl:string-upcase} are the functions:

\begin{verbatim}
   string-upcase
   nstring-upcase
\end{verbatim}

For a table of related items: See the section "String Conversion".

\textbf{stringp object}

\textit{Function}

Under Genera, determines if \textit{object} is either type of string, returning \texttt{t} if it is, and \texttt{nil} otherwise. Accepts any object as an argument.

A string is a one-dimensional array whose elements can be of type \texttt{string-char} or \texttt{character}; since \texttt{stringp} is a supertype of \texttt{simple-string-p}, it always returns \texttt{t} for any object of which \texttt{simple-string-p} is \texttt{t}.

Unlike arrays of type \texttt{simple-string}, an array of type \texttt{string} can have a fill pointer and displacement (that is, it can be extended, and its contents can be shared with other array objects).

The function \texttt{stringp} is an extension of its Common Lisp counterpart, since it returns \texttt{t} for arrays with elements of type \texttt{character} as well as for arrays of type \texttt{string-char}. In CLOE on the 386, \texttt{stringp} is true only of arrays with elements of type \texttt{string-char}.

Examples:
(stringp "string") => T
(stringp 'symbol) => NIL
(stringp 123) => NIL
(stringp (make-string 3 :initial-element #\a)) => T
(stringp (make-string 3 :initial-element #\a
    :element-type 'character)) => T
(stringp (make-array 5 :element-type 'string-char
    :fill-pointer 8)) => T
(stringp (make-array 4 :element-type 'character
    :fill-pointer 3)) => T
(simple-string-p (make-array 5 :element-type 'string-char
    :fill-pointer 8)) => NIL

Under CLOE,

(stringp "hello") => t
(stringp '#(#\h #\e #\l #\l #\o)) => nil
(stringp "h") => t

In the previous example, the value of the second call to stringp is nil because a vector of characters is not a string. Instead, strings are vectors whose element type is string-char.

(stringp (make-array 3 :element-type 'string-char)) => t
(stringp (make-array 3 :element-type 'character)) => nil

For a table of related items, see the section "String Type-Checking Predicates".

structure &optional (name '*)

Type Specifier

This is the type specifier symbol denoting instances of a structure. When a new structure is defined with defstruct, the name of the structure type becomes a valid type symbol, and individual instances of that structure become valid types of structure that can be tested with typep.

structure is a subtype of t.

Examples:

(defstruct ship
  x-position
  y-position) => SHIP
(setq my-boat (make-ship)) => #S(SHIP :X-POSITION NIL
  :Y-POSITION NIL)
(typep my-boat '(structure ship)) => T
(zl:typep my-boat) => SHIP
(type-of my-boat) => SHIP
(sys:type-arglist 'structure) => (&OPTIONAL (NAME '*)) and T

See the section "Data Types and Type Specifiers". See the section "Structure Macros".
clos:structure-class

The class of classes defined by defstruct.

These classes (objects whose class is clos:structure-class) are provided so users can define methods that specialize on them. They do not support the full behavior of user-defined classes (whose class is clos:standard-class). For example, you cannot use clos:make-instance to create instances of these classes.

zl:sub1 x

(zl:sub1 x) is the same as (- x 1).

The following functions are synonyms of zl:sub1:

```
  1-
  zl:1-$
```

sublis alist tree &rest args &key (:test #'eql) :test-not (:key #'identity)

Makes non-destructive substitutions for objects in a tree (a structure of conses). Returns a tree with the substitutions made. The first argument to sublis is an association list alist. The second argument is the tree in which the substitutions are to be made, as for subst. sublis looks at all the subtrees and leaves of the tree. If a subtree or leaf appears as a key in the association list (that is, the key and the subtree or leaf satisfy the predicate specified by the :test keyword), it is replaced by the datum associated with it. The keywords are:

**:test**

Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

**:test-not**

Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

**:key**

If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

```
(setq exp '((* x y) (+ x y))) => ((* X Y) (+ X Y))

(sublis '(* 100)) exp) => ((* 100 Y) (+ 100 Y))
```

The result may share structure with tree.

```
(setq alist (pairlis '(1 2 3) '(giraffe zebra monkey)))
(setq thing '(spotted 1 (striped 2) fast 3))

(sublis alist thing)
=> (SPOTTED GIRAFFE (STRIPED ZEBRA) FAST MONKEY)
```
Thus, sublis is comparable to several subst operations in parallel. The following example simulates the previous example by executing three sequential subst operations.

```lisp
(setq tmp (subst 'giraffe 1 thing))
(setq tmp (subst 'zebra 2 tmp))
(subst 'monkey 3 tmp)
```

However, not every sublis call can be replaced by sequential calls to subst, as demonstrated in the following example:

```lisp
(setq alist (pairlis '(monkey zebra) '(zebra monkey)))
(setq newthing '(is-taller monkey zebra))

(sublis alist newthing) => (IS-TALLER ZEBRA MONKEY)
```

For a table of related items: See the section "Functions for Modifying Lists".

**zl:sublis alist form**

Makes substitutions for symbols in a tree. The first argument to zl:sublis is an association list alist. The second argument is the tree in which substitutions are to be made. zl:sublis looks at all symbols in the fringe of the tree; if a symbol appears in the association list, occurrences of it are replaced by the object associated with it. The argument is not modified; new conses are created where necessary and only where necessary, so the newly created tree shares as much of its substructure as possible with the old. For example, if no substitutions are made, the result is just the old tree. Example:

```lisp
(zl:sublis '((x . 100) (z . zprime))
  '(plus x (minus g z x p) 4))
=> (plus 100 (minus g zprime 100 p) 4)
```

zl:sublis could have been defined by:

```lisp
(defun zl:sublis (alist sexp)
  (cond ((symbolp sexp)
                (let ((tem (assq sexp alist)))
                  (if tem (cdr tem) sexp)))
    ((listp sexp)
                 (let ((car (sublis alist (car sexp)))
                       (cdr (sublis alist (cdr sexp))))
                   (if (and (eq (car sexp) car) (eq (cdr sexp) cdr))
                       sexp
                       (cons car cdr))))
    (t
     (sexp))))
```

In your new programs, we recommend that you use the function sublis, which is the Common Lisp equivalent of zl:sublis.

For a table of related items: See the section "Functions for Modifying Lists".
Function

In your new programs we recommend that you use the function `compiled-function-p`, which is the Common Lisp equivalent of the function `zl:subrp`.

`zl:subrp` returns `t` if its argument is any compiled code object, otherwise `nil`. Symbolics Common Lisp does not use the term "subr"; the name of this function comes from Maclisp.

Function

Returns the subsequence of `sequence` specified by `start` and `end`. `subseq` always allocates a new sequence for a result, and never shares storage with an old sequence. The result subsequence is always of the same type as `sequence`.

`sequence` can be either a list or a vector (one-dimensional array). Note that `nil` is considered to be a sequence, of length zero.

For example:

```lisp
(subseq '(1 2 3 4 5) 3 5) => #(4 5)
```

Note `start` and `end` are not keywords, since you must specify `start` in order to use the function.

`setf` can be used with `subseq` to destructively replace a subsequence with a sequence of new values. See the function `replace`. See the function `substitute`. For example:

```lisp
(setq num-list '(1 2 3 4 5)) => (1 2 3 4 5)
(setf (subseq num-list 2 4) '(0 0)) => (0 0)
num-list => (1 2 0 0 5)
```

The following example demonstrates a simplified subsequence replacement function defined in terms of `subseq`:

```lisp
(setq seq1 #(a b c d e))
(setq seq2 #(1 2 3 4))

(defun my-replace (sequence1 sequence2 &key start1 end1 start2 end2)
  "real replace must do some extra work"
  (setf (subseq sequence1 start1 end1)
        (subseq sequence2 start2 end2))
  sequence1)

(my-replace seq1 seq2 :start1 1 :end1 4 :start2 0 :end2 3)
=> #(a 1 2 3 E)
```

For a table of related items: See the section "Sequence Construction and Access".
**zl:subset**  \( \text{pred list &rest extra-lists} \) **Function**

Removes from list those elements that do not satisfy \( \text{pred} \). That is, it keeps the elements for which \( \text{pred} \) is true. \( \text{zl:subset} \) does the same thing, but is used if list does not represent a mathematical set.

\( \text{pred} \) should be a function of one argument, if there are no \( \text{extra-lists} \) arguments. If \( \text{extra-lists} \) is present, each element of \( \text{extra-lists} \) (that is, each further argument to \( \text{zl:subset} \) or \( \text{zl:rem-if-not} \)) is a list of objects to be passed to \( \text{pred} \) as \( \text{pred} \)'s second argument, third argument, and so on. The reason for this is that \( \text{pred} \) might be a function of many arguments; \( \text{extra-lists} \) lets you control what values are passed as additional arguments to \( \text{pred} \). However, the list returned by \( \text{zl:subset} \) or \( \text{zl:rem-if-not} \) is still a "subset" of the \( \text{first} \) argument in the various calls to \( \text{pred} \).

For a table of related items: See the section "Functions for Modifying Lists".

**zs: subset-not**  \( \text{pred list &rest extra-lists} \) **Function**

Removes from list those elements that satisfy \( \text{pred} \). A new list is made by applying \( \text{pred} \) to all the elements of list and removing the ones that satisfy it. \( \text{zl:rem-if} \) does the same thing, but is used if list does not represent a mathematical set.

\( \text{zl:subset-not} \) and \( \text{zl:rem-if} \) do the same thing, but they are used in different contexts. \( \text{zl:subset-not} \) refers to the function's action if list is considered to represent a mathematical set.

\( \text{pred} \) should be a function of one argument, if there are no \( \text{extra-lists} \) arguments. If \( \text{extra-lists} \) is present, each element of \( \text{extra-lists} \) (that is, each further argument to \( \text{zl:subset-not} \) or \( \text{zl:rem-if} \)) is a list of objects to be passed to \( \text{pred} \) as \( \text{pred} \)'s second argument, third argument, and so on. The reason for this is that \( \text{pred} \) might be a function of many arguments; \( \text{extra-lists} \) lets you control what values are passed as additional arguments to \( \text{pred} \). However, the list returned by \( \text{zl:subset-not} \) or \( \text{zl:rem-if} \) is still a "subset" of the \( \text{first} \) argument in the various calls to \( \text{pred} \).

For a table of related items: See the section "Functions for Modifying Lists".

**subsetp list1 list2 &key (test #'eql) :test-not (:key #'identity)**  **Function**

A predicate that is true if every element of list1 appears in list2, and false otherwise.

\[
\text{(setq a-list '(loon stork heron)) => (LOON STORK HERON)}
\]

\[
\text{(setq b-list '(loon owl stork eagle heron)) => (LOON OWL STORK EAGLE HERON)}
\]

\[
\text{(subsetp a-list b-list) => T}
\]

\[
\text{(subsetp b-list a-list) => NIL}
\]
The keywords are:

:test
Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not
Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

:key
If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

In the following example, subsetp determines whether or not elements are approved for storage.

```
(unless (subsetp elements-to-be-stored
    elements-checked-ok-for-storage)
  (setq elements-to-be-checked-for-storage
    (set-difference elements-to-be-stored
      elements-checked-ok-for-storage)))
```

For a table of related items: See the section "Predicates that Operate on Lists".

Subst new old tree &rest args &key (:test #'eql) :test-not (:key #'identity)

Function

Makes a copy of tree, substituting new for every subtree or leaf of tree (whether the subtree or leaf is a car or cdr of its parent), such that old and the subtree or leaf satisfy the predicate specified by the :test keyword. It returns the modified copy of tree, and the original tree is unchanged, although it can share with parts of the result tree. For example:

```
(setq bird-list '((waders (flamingo stork) raptors (eagle hawk))) =>
  (WADERS (FLAMINGO STORK) RAPTORS (EAGLE HAWK)))

(subst 'heron 'stork bird-list) =>
  (WADERS (FLAMINGO HERON) RAPTORS (EAGLE HAWK))
```

The keywords are:

:test
Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

:test-not
Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.
:key  

If not nil, should be a function of one argument that will extract the part to be tested from the whole element.

In Common Lisp (unlike MacLisp and Zetalisp), subst does not execute a full copy-tree. If a full copy is necessary, copy-tree may be called before, after, or instead of subst.

The following example uses subst in a parsed English sentence to replace relative pronouns with the appropriate proper nouns. The :key function, car, finds the s-expressions that need replacement.

```
(setq sentence
  '(((SUB (PN . Mork)) (PRED (V . was) (ADJ . young))
    (SUB (RPN . he)) (PRED (V . was) (ADJ . excited))
    (SUB (RPN . he)) (PRED (V . was) (ADJ . happy))))

(subst '(PN . Mork) 'RPN sentence :key #'(lambda(x)(and (consp x)(car x))))
=>
(((SUB (PN . MORK)) (PRED (V . WAS) (ADJ . YOUNG))
  (SUB (PN . MORK)) (PRED (V . WAS) (ADJ . EXCITED))
  (SUB (PN . MORK)) (PRED (V . WAS) (ADJ . HAPPY))))
```

For a table of related items: See the section "Functions for Modifying Lists".

**zl:subst**  

`new old s-exp`  

Function

Substitutes new for all occurrences of old in s-exp, and returns the modified copy of s-exp. The original s-exp is unchanged, as zl:subst recursively copies all of s-exp replacing elements that are equal to old as it goes. Example:

```
(zl:subst 'Tempest 'Hurricane
  '(Shakespeare wrote (The Hurricane)))
=> (Shakespeare wrote (The Tempest))
```

zl:subst could have been defined by:

```
(defun zl:subst (new old tree)
  (cond ((equal tree old) new) ;if item equal to old, replace.
    ((atom tree) tree)       ;if no substructure, return arg.
    ((cons (subst new old (car tree)) ;otherwise recurse.
      (subst new old (cdr tree)))))))
```

Note that this function is not "destructive"; that is, it does not change the car or cdr of any existing list structure.

The old practice of using zl:subst to copy trees is unclear and obsolete. In your new programs use the Common Lisp version of this function, which is subst.

For a table of related items: See the section "Functions for Modifying Lists".

**subst-if**  

`new predicate tree &rest args &key :key`  

Function

Makes a copy of tree, substituting new for every subtree or leaf of tree, such that the subtree or leaf satisfies predicate. It returns the modified copy of tree; the orig-
inal tree is unchanged, although it can share with parts of the result tree. For example:

```lisp
(setq item-list '(numbers (1.0 2 5/3) symbols (foo bar))) =>
(NUMBERS (1.0 2 5/3) SYMBOLS (FOO BAR))

(subst-if '3.1415 #'numberp item-list) =>
(NUMBERS (3.1415 3.1415 3.1415) SYMBOLS (FOO BAR))
```

item-list => (NUMBERS (1.0 2 5/3) SYMBOLS (FOO BAR))

The keyword is:

**:key**

If not **nil**, should be a function of one argument that will extract the part to be tested from the whole element.

The following two calls to `subst-if` use two anonymous functions. The different results are due to the case sensitivity of `string=`.

```lisp
(setq a '("In" "our" "prairie" "home" "we" "read"
"The" "Prairie" "Home" "Companion"))

(subst-if "Gopher"
  #'(lambda (comparator)(string= comparator "Prairie")))
=>
("In" "our" "prairie" "home" "we" "read"
 "The" "Gopher" "Home" "Companion")

(subst-if "Gopher"
  #'(lambda (comparator)(string-equal comparator "Prairie")))
=>
("In" "our" "Gopher" "home" "we" "read"
 "The" "Gopher" "Home" "Companion")
```

For a table of related items: See the section "Functions for Modifying Lists".

**subst-if-not** new predicate tree &rest args &key :key

Function

Makes a copy of tree, substituting new for every subtree or leaf of tree such that old and the subtree or leaf do not satisfy predicate. It returns the modified copy of tree; the original tree is unchanged, although it can share with parts of the result tree. For example:

```lisp
(setq item-list '(numbers (1.0 2 5/3) symbols (foo bar))) =>
(NUMBERS (1.0 2 5/3) SYMBOLS (FOO BAR))

(subst-if-not '3.1415 #'numberp item-list) =>
(3.1415 (1.0 2 5/3) 3.1415 (3.1415 3.1415))
```
The keyword is:

:KEY

If not **nil**, should be a function of one argument that will extract the part to be tested from the whole element.

For a table of related items: See the section "Functions for Modifying Lists".

**substitute newitem olditem sequence &key (:test #'eql) :test-not (:#key #'identity) :from-end (start 0) :end :count**

Returns a sequence of the same type as *sequence* that has the same elements, except that those in the subsequence delimited by :start and :end and satisfying the predicate specified by the :test keyword are replaced by *newitem*. This is a non-destructive operation, and the result is a copy of *sequence* with some elements changed.

For example:

```
(setq letters '(a b c)) => (A B C)
(substitute 'a 'b '(a b c)) => (A A C)
letters => (A B C)
```

```
(substitute 'b 'c letters) => (A B B)
letters => (A B C)
```

*newitem* and *olditem* can be any Symbolics Common Lisp object but must be a suitable element for the *sequence*.

*sequence* can be either a list or a vector (one-dimensional array). Note that *nil* is considered to be a sequence, of length zero.

:TEST specifies the test to be performed. An element of *sequence* satisfies the test if `(funcall testfun item (keyfn x))` is true. Where `testfun` is the test function specified by :test, `keyfn` is the function specified by :key and `x` is an element of the sequence. The default test is `eql`.

For example:

```
(substitute 0 3 '(1 1 4 4 2) :test #'<) => (1 1 0 0 2)
```

:TEST-NOT is similar to :TEST, except that the sense of the test is inverted. An element of *sequence* satisfies the test if `(funcall testfun item (keyfn x))` is false.

The value of the keyword argument :KEY, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```
(substitute 1 2 '((1 1) (1 2) (4 3)) :key #'second) => ((1 1) 1 (4 3))
```

```
(substitute 'a 'b '((a b) (b c) (b b)) :key #'cadr) => (A (B C) A)
```
A non-nil :from-end specification matters only when the :count argument is provided; in that case only the rightmost :count elements satisfying the test are replaced.

For example:

```
(substitute 'hi 'b '(b a b) :from-end t :count 1)
=> (B A HI)
```

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

```
(substitute 'a 'b '(b a b) :start 1 :end 3) => (B A A)

(substitute 'a 'b '(b a b) :end 2) => (A A B)

(substitute 'a 'b '(b a b) :end 3) => (A A A)
```

The :count argument, if specified, limits the number of elements altered. If more than :count elements satisfy the predicate, then only the leftmost :count elements are replaced. A negative :count argument is equivalent to a :count of 0.

For example:

```
(substitute 'a 'b '(b b a b b) :count 3) => (A A A A B)
```

The result of the substitute function can share cells with the argument sequence. A list can share a tail with an input list, and the result can be eq to the input sequence if no elements need to be changed.

See the function subst.

```
(setq realtor-list (list 'lot11 'lot21 'lot34 'lot42 'lot56))

(substitute 'taken "21" realtor-list :test #'string-equal :key #'(lambda(x)(subseq (string x) 3)))
=> (LOT11 TAKEN LOT34 LOT42 LOT56)
```

substitute is the non-destructive version of nsubstitute.

For a table of related items: See the section "Sequence Modification".
substitute-if newitem predicate sequence &key :key :from-end (start 0) :end :count

Returns a sequence of the same type as sequence that has the same elements, except that those in the subsequence delimited by :start and :end and satisfying predicate are replaced by newitem. This is a non-destructive operation, and the result is a copy of sequence with some elements changed.

For example:

```
(setq numbers '(0 1 19)) => (0 1 19)
(substitute-if 1 #'zerop numbers) => (1 1 19)
numbers => (0 1 19)
```

```
(substitute-if 2 #'numberp numbers) => (2 2 2)
numbers => (0 1 19)
```

newitem can be any Symbolics Common Lisp object but must be a suitable element for the sequence.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one argument. This function extracts from each element the part to be tested in place of the whole element.

For example:

```
(substitute-if 1 #'oddp '((1 1) (1 2) (4 3)) :key #'second)
```

```
=> (1 (1 2) 1)
```

A non-nil :from-end specification matters only when the :count argument is provided; in that case only the rightmost :count elements satisfying the test are replaced.

For example:

```
(substitute-if 'hi #'atom '(b 'a b) :from-end t :count 1)
```

```
=> (B 'A HI)
```

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:
(substitute-if 1 #'zerop '(0 1 0) :start 1 :end 3)
=> (0 1 1)

(substitute-if 1 #'zerop '(0 1 0) :start 0 :end 2)
=> (1 1 0)

(substitute-if 1 #'zerop '(0 1 0) :end 1)
=> (1 1 0)

A non-nil :count, if supplied, limits the number of elements altered; if more than
:count elements satisfy the test, then of these elements only the leftmost are re-
placed, as many as specified by :count. A negative :count argument is equivalent
to a :count of 0.

For example:

(substitute-if 'see 'atom '(b b a b b) :count 3)
=> (SEE SEE SEE B B)

substitute-if is the non-destructive version of nsubstitute-if.

For a table of related items: See the section "Sequence Modification".

[substitute-if-not newitem predicate sequence &key :key :from-end (:start 0) :end :count]

Function

Returns a sequence of the same type as sequence that has the same elements, ex-
cept that those in the subsequence delimited by :start and :end that do not satisfy
predicate are replaced by newitem. This is a non-destructive operation, and the re-
sult is a copy of sequence with some elements changed.

For example:

(setq numbers '(0 0 0)) => (0 0 0)
(substitute-if-not 1 #'numberp numbers) => (0 0 0)
numbers => (0 0 0)

(substitute-if-not 2 #'consp numbers)  => (2 2 2)
numbers => (0 0 0)

newitem can be any Symbolics Common Lisp object but must be a suitable element
for the sequence.

predicate is the test to be performed on each element.

sequence can be either a list or a vector (one-dimensional array). Note that nil is
considered to be a sequence, of length zero.

The value of the keyword argument :key, if non-nil, is a function that takes one
argument. This function extracts from each element the part to be tested in place
of the whole element.

For example:
(substitute-if-not 1 #'oddp '((1 1) (1 2) (4 3)) :key #'second)
=> ((1 1) 1 (4 3))

A non-nil :from-end specification matters only when the :count argument is provided; in that case only the rightmost :count elements satisfying the test are replaced.

For example:

(substitute-if-not 'hi #'atom '('b a 'b) :from-end t :count 1)
=> ('B A HI)

Use the keyword arguments :start and :end to delimit the portion of the sequence to be operated on.

:start and :end must be non-negative integer indices into the sequence. :start must be less than or equal to :end, else an error is signalled. It defaults to zero (the start of the sequence).

:start indicates the start position for the operation within the sequence. :end indicates the position of the first element in the sequence beyond the end of the operation. It defaults to nil (the length of the sequence).

If both :start and :end are omitted, the entire sequence is processed by default.

For example:

(substitute-if-not 1 #'zerop '(3 0 2) :start 1 :end 3)
=> (3 0 1)

(substitute-if-not 1 #'zerop '(3 0 2) :start 0 :end 2)
=> (1 0 2)

(substitute-if-not 1 #'zerop '(3 0 2) :end 1)
=> (1 0 2)

A non-nil :count, if supplied, limits the number of elements altered; if more than :count elements satisfy the test, only the leftmost are replaced, as many as specified by :count. A negative :count argument is equivalent to a :count of 0.

For example:

(substitute-if-not 'see 'consp '(b b a b b) :count 3)
=> (SEE SEE SEE B B)

substitute-if-not is the non-destructive version of nsubstitute-if-not.

For a table of related items: See the section "Sequence Modification".

substring string from &optional to (area nil) Function

Extracts a substring of string, starting at the character specified by from and going up to but not including the character specified by to.

string is a string or an object that can be coerced to a string. See the function string.
from and to are 0-origin indices. The length of the returned string is to minus from. If to is not specified it defaults to the length of string. The area in which the result is to be consed can be optionally specified.

The destructive version of substring is the function nsubstring.

Examples:

(substring "Nebuchadnezzar" 4 8) => "chad"
(substring "Nebuchadnezzar" 4) => "chadnezzar"
(substring 'string 1 4) => "TRI"
(setq a "Aloysius") => "Aloysius"
(setq b (substring a 2 4)) => "oy"
(nstring-upcase b) => "OY"
(substring a 0) => "Aloysius"

For a table of related items: See the section "String Access and Information".

**subt**ypep type1 type2

Function

Compares the two type specifiers, type1 and type2. subtypep is true if type1 is definitely a subtype of type2. If the result is nil, however, type1 may or may not be a subtype of type2 (sometimes it is impossible to tell, especially when satisfies type specifiers are involved). A second returned value indicates the certainty of the result; if it is true, then the first value is an accurate indication of the subtype relationship. Thus, subtypep returns one of three possible result combinations:

t t type1 is definitely a subtype of type2.
nil t type1 is definitely not a subtype of type2.
nil nil subtypep could not determine the relationship.

The arguments type1 and type2 must be type specifiers that are acceptable to typep. For standard Symbolics Common Lisp type specifiers, see the section "Type Specifiers".

Examples:

(subtypep 'single-float 'float) => T and T ; subtype and certain
(subtypep 'bit '(number 0 4)) => T and T
(subtypep 'array t) => T and T
(subtypep 'common t) => T and T
(subtypep 'signed-byte 'bit) => NIL and T
(subtypep '(integer 0 (8)) 'integer) => t t
(subtypep 'integer 'float) => nil t

The following example illustrates a second returned value of nil. Note that subtypep can not determine the requirements for a user-defined predicate.
See the section "Data Types and Type Specifiers".

**sum keyword for loop**

```
sum expr (data-type) (into var)
```

Evaluates `expr` on each iteration, and accumulates the sum of all the values. `data-type` defaults to `number`, which for all practical purposes is `notype`. Note that specifying `data-type` implies that both the sum and the number being summed (the value of `expr`) is of that type. When the epilogue of the `loop` is reached, `var` has been set to the accumulated result and can be used by the epilogue code.

It is safe to reference the values in `var` during the loop, but they should not be modified until the epilogue code for the loop is reached.

The forms `sum` and `summing` are synonymous.

Examples:

```lisp
(defun geometric-s (num)
    (loop for i from 1 to num
        sum i into sum-var
        finally (print sum-var))) => GEOMETRIC-S

(geometric-s 5) =>
15 NIL
```

Is equivalent to

```lisp
(defun geometric-s (num)
    (loop for i from 1 to num
        summing i into sum-var
        finally (print sum-var))) => GEOMETRIC-S

(geometric-s 5) =>
15 NIL
```

Not only can there be multiple accumulations in a `loop`, but a single accumulation can come from multiple places within the same `loop` form, if the types of the collections are compatible. `sum` and `count` are compatible.

See the section "Accumulating Return Values for `loop`".

**svref array &rest subscript Function**

Returns the element of the vector selected by `subscript`. The first argument must be a simple vector. The `subscript` must be an integer.
A vector is simple if non-adjustable, has no fill pointer, and can hold elements of any type (that is, has an element-type of t).

\[(\text{svref '}(2 4 6 8) 3) \Rightarrow 8\]

`:swap-hash key value`  
Message  
Does the same thing as `zl:puthash`, but returns different values. If there was an existing entry in the hash table whose key was `key`, it returns the old associated value as its first returned value, and `t` as its second returned value. Otherwise it returns two values, `nil` and `nil`. This message is obsolete; use `swaphash` instead.

`zl:swapf a b`  
Macro  
Exchanges the value of one generalized variable with that of another. `a` and `b` are access-forms suitable for `zl:setf`. The returned value is not defined. `zl:swapf` expands into a `rotatef`, which expands into a `progn`, so there is no danger of the access-forms being evaluated more than once.

Examples:

\[
(\text{zl:swapf a b})
\Rightarrow (\text{rotatef a b})
\Rightarrow (\text{progn (setq a (values (prog1 b (setq b a)))) nil})
\]

\[
(\text{zl:swapf (car (foo)) (car (bar))})
\Rightarrow (\text{rotatef (car (foo)) (car (bar))})
\Rightarrow (\text{progn (let* ((#:g1849 (foo))
(#:g1851 (bar)))
(sys:rplaca2 #:g1849
(values
(prog1 (car #:g1851)
(sys:rplaca2 #:g1851
(values (car #:g1849))))))}))
\]

See the section "Generalized Variables".

`swaphash key value hash-table`  
Function  
Does the same thing as `zl:puthash`, but returns different values. If there was an existing entry in `hash-table` whose key was `key`, it returns the old associated value as its first returned value, and `t` as its second returned value. Otherwise it returns two values, `nil` and `nil`.

For a table of related items: See the section "Table Functions".
**zl:swaphash-equal**  
*key value hash-table*  
*Function*

Does the same thing as **zl:puthash**, but returns different values. If there was an existing entry in **hash-table** whose key was **key**, it returns the old associated value as its first returned value, and **t** as its second returned value. Otherwise it returns two values, **nil** and **nil**. This function is obsolete; use **swaphash** instead.

**sxhash**  
*x*  
*Function*

Computes a hash code of an object, and returns it as a fixnum. A property of **sxhash** is that \((\text{equal } x y)\) always implies \((= (\text{sxhash } x) (\text{sxhash } y))\). The number returned by **sxhash** is always a nonnegative fixnum, possibly a large one. **sxhash** tries to compute its hash code in such a way that common permutations of an object, such as interchanging two elements of a list or changing one character in a string, always changes the hash code.

Under Genera, **sxhash** is the same as **si:equal-hash**, except that **sxhash** returns **0** as the hash value for objects with data types like arrays, stack groups, or closures. As a result, hashing such structures could degenerate to the case of linear search.

\[(\text{sxhash 'key}) \Rightarrow 158428288\]

**symbol**  
*Type Specifier*

**symbol-function**  
*symbol*  
*Function*

Returns the current global function definition named by **symbol**. If **symbol** has no function definition, signals an error. The definition can be a function or an object representing a special form or macro. If the definition is an object representing special form or a macro, it is an error to try to invoke the object as a function. Lexically scoped function definitions, produced by **flet** or **labels**, can not be accessed by **symbol-function**. Only the global value of a named function can be accessed.

\[(\text{defun foo(x y) (list x 'foo y)})\]
\[\text{FOO}\]
\[(\text{symbol-function 'foo})\]
\[\text{#<function:1547434>}\]
\[(\text{funcall (symbol-function 'foo) 'bar 'baz})\]
\[\text{(BAR FOO BAZ)}\]

See the section “Functions Relating to the Function Cell of a Symbol”.

**clos:symbol-macrolet**  
*symbols-and-expressions &body body*  
*Special Form*

Provides the underlying mechanism for substituting expressions for variable names within a lexical scope; both **clos:with-accessors** and **clos:with-slots** are implemented via **clos:symbol-macrolet**.
symbols-and-expressions
A list made up of sublists of the form:

(symbol expression)

The symbol specifies a symbol associated with the form expression.

declarations
The clos:symbol-macrolet syntax allows declarations to appear before the body.

body
Within the body, the symbols are associated with the expressions in the following way: each reference to a symbol as a variable is replaced by expression (not the result of evaluating expression).

When the body of the clos:symbol-macrolet form is expanded, any use of setq to set the value of one of the specified variables is converted to a use of setf.

The values of clos:symbol-macrolet are whatever values are returned by the body.

symbol-name symbol Function
Returns the print name of symbol. Example:

(symbol-name 'xyz) => "xyz"

See the section "Functions Relating to the Print Name of a Symbol".

symbol-package symbol Function
Returns the contents of symbol’s package cell, which is the package that owns symbol, or nil if symbol is uninterned.

(symbol-package 'equal) => #<PACKAGE:LISP>

See the section "The Package Cell of a Symbol".

symbol-plist symbol Function
Returns the list that represents the property list of symbol. Note that this is not the property list itself; you cannot do get on it. You must give the symbol itself to get or use getf.

You can use setf to destructively replace the entire property list of a symbol; however, this is potentially dangerous since it may destroy information that the Lisp system has stored on the property list. You also must be careful to make the new property list a list of even length.

This function is primarily for debugging purposes. We do not recommend the use of setf with symbol-plist unless you recognize the consequences of rendering the old property list inaccessible.
(symbol-plist 'some-symbol)

=> (COLOR RED SPEED MYSTICAL HIT-POINTS 60)

See the section "Functions Relating to the Property List of a Symbol".

symbol-value symbol

Function

Returns the current value of the dynamic (special) variable named symbol. This is the function called by eval when it is given a symbol to evaluate. If the symbol is unbound, symbol-value causes an error. Constant symbols are really variables whose values cannot be changed. You can use symbol-value to get the value of such a constant. symbol-value of a keyword returns that keyword.

symbol-value works only on special variables. It cannot find the value of a lexical variable.

(defconstant *max-alarms* 1000)

(symbol-value '*max-alarms*) => 1000

See the section "Functions Relating to the Value of a Symbol".

symbol-value-globally var

Function

Works like symbol-value but returns the global value of a special variable regardless of any bindings currently in effect (in the current stack group).

symbol-value-globally does not work on local (lexical) variables.

You can use setf with symbol-value-globally to bind the global value of a special variable. (setf (symbol-value-globally var)) ... ) is the same as zl:set-globally and supersedes zl:setq-globally.

See the section "Functions Relating to the Value of a Symbol".

symbol-value-in-closure closure ptr

Function

Returns the binding of symbol in the environment of closure; that is, it returns what you would get if you restored the value cells known about by closure and then evaluated symbol. This allows you to "look around inside" a dynamic or lexical closure. If symbol is not closed over by closure, this is just like symbol-value.

See the section "Dynamic Closure-Manipulating Functions".

symbol-value-in-instance instance symbol &optional no-error-p

Function

Reads, alters, or locates an instance variable inside a particular instance, regardless of whether the instance variable was declared in the defflavor form to be a :readable-instance-variable, :gettable-instance-variable, :writable-instance-variable, :settable-instance-variable, or a :locatable-instance-variable.
instance is the instance to be examined, and symbol is the instance variable. If there is no such instance variable, an error is signalled, unless no-error-p is non-nil, in which case nil is returned.

To read the value of an instance variable:

\[(\text{symbol-value-in-instance instance symbol})\]

To alter the value of an instance variable:

\[(\text{setf (symbol-value-in-instance instance symbol) value})\]

To get a locative pointer to the cell inside an instance that holds the value of an instance variable:

\[(\text{locf (symbol-value-in-instance instance symbol)})\]

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**symbolp arg**

Function

Returns t if its argument is a symbol, otherwise nil.

For example:

\[(\text{symbolp nil}) => t\]
\[(\text{setq foo 12 bar 'foo})\]
\[(\text{symbolp 'foo}) => t\]
\[(\text{symbolp foo}) => \text{nil}\]
\[(\text{symbolp bar}) => t\]

**zl:symeval symbol**

Function

In your new programs, we recommend that you use the function symbol-value, which is the Common Lisp equivalent of the function zl:symeval.

zl:symeval is the basic Zetalisp primitive for retrieving a symbol's value. (zl:symeval symbol) returns symbol's current binding. This is the function called by eval when it is given a symbol to evaluate. If the symbol is unbound, then zl:symeval causes an error.

See the section "Functions Relating to the Value of a Symbol".

**zl:symeval-globally var**

Function

In your new programs, we recommend that you use the function symbol-value-globally, which is the Symbolics Common Lisp equivalent of the function zl:symeval-globally.

Works like zl:symeval but returns the global value regardless of any bindings currently in effect.
**zl:symeval-globally** operates on the *global value* of a special variable; it bypasses any bindings of the variable in the current stack group. It resides in the global package.

**zl:symeval-globally** does not work on local variables.

See the section "Functions Relating to the Value of a Symbol".

**zl:symeval-in-closure** *closure symbol*

*Function*

Use the Symbolics Common Lisp function **symbol-value-in-closure**, which is equivalent to the function **zl:symeval-in-closure**.

This returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the value cells known about by *closure* and then evaluated *symbol*. This allows you to "look around inside" a dynamic or lexical closure. If *symbol* is not closed over by *closure*, this is just like **zl:symeval**. See the section "Dynamic Closure-Manipulating Functions".

**zl:symeval-in-instance** *instance symbol* &optional *no-error-p*

*Function*

In your new programs, we recommend that you use the function **symbol-value-in-instance**, which is the Symbolics Common Lisp equivalent of the function **zl:symeval-in-instance**.

Finds the value of an instance variable inside a particular instance, regardless of whether the instance variable was declared a :readable-instance-variable or a :gettable-instance-variable. *instance* is the instance to be examined, and *symbol* is the instance variable whose value should be returned. If there is no such instance variable, an error is signalled, unless *no-error-p* is non-nil, in which case nil is returned.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

**t**

*Type Specifier*

*t* is the type specifier symbol for the predefined Lisp data type of that name.

The type **t** is a supertype of every type whatsoever. Every Lisp object belongs to type **t**.

Examples:

```
(typep nil 't) => T
(typep 12 't) => T
(constantp t) => T
(equal-typep (not nil) t) => T
(subtypep nil 't) => T
(subtypep 'character 't) => T
(subtypep 'null 't) => T
```
See the section "Data Types and Type Specifiers".

**table-size** *table*  
*Function*

Returns the total number of entries in *table*. Note that this does not include the number of entries that are deleted but not removed from the table.

For a table of related items: See the section "Table Functions".

**tagbody** &body *forms*  
*Special Form*

The body of a *tagbody* form is a series of tags or statements. A tag can be a symbol or an integer; a statement is a list. *tagbody* processes each element of the body in sequence. It ignores tags and evaluates statements, discarding the results. If it reaches the end of the body, it returns nil.

If a (go *tag*) form is evaluated during evaluation of a statement, *tagbody* searches its body and the bodies of any *tagbody* forms that lexically contain it. Control is transferred to the innermost tag that is eql to the tag in the go form. Processing continues with the next tag or statement that follows the tag to which control is transferred.

The scope of the tag is lexical. That is, the go form must be inside the tagbody construct itself (or inside a tagbody form that that tagbody lexically contains), not inside a function called from the tagbody.

**do, prog**, and their variants use implicit *tagbody* constructs. You can provide tags within their bodies and use go forms to transfer control to the tags.

Examples:

```
(let ((x 'hello))
 (tagbody
   (catch 'stuff
     (if (numberp x)
         (princ "a number")
         (go trouble)))
   (return)
   trouble
   (princ "trouble trouble") (terpri))) => trouble trouble
NIL
```

The following two forms are equivalent:

```
(dotimes (i n) (print i))
```
(let ((i 0))
  (when (plusp n)
    (tagbody
     loop
     (print i)
     (setq i (1+ i))
     (when (< i n) (go loop))))))

(let ((i 0)
  (result t))
  (tagbody loop
    (when (and (< i 20) result)
      (unless (= (aref *data-vector-a* i) (aref *data-vector-b* i))
        (setq result nil))
      (go loop))))

For a table of related items: See the section "Transfer of Control Functions".

**tailp** tail list

 tête Function

Returns t if tail is an ending sublist of list (that is, one of the conses that make up list), otherwise returns nil. Another way to look at this is that tailp returns t if (nthcdr n list) returns tail for some n. For example:

(setq item-list '(a b c)) => (A B C)
(tailp (cdr item-list) item-list) => T
(tailp (car item-list) item-list) => NIL
(tailp (nthcdr 2 item-list) item-list) => T
(tailp nil item-list) => T

Tailp could have been defined by:

(defun tailp (tail list)
  (do () ((eq tail list) t)
     (if (atom list)
       (return nil)
       (setf list (cdr list)))))

The following definition returns nil if the second argument is not a sublist of the first argument; otherwise, a copy of the prefix portion of the first argument is returned. This example illustrates how tailp determines whether or not the second argument is actually a sublist of the first argument.

(defun ldiff-if-sublist (list sublist)
  (if (tailp sublist list)
      (do ((old-list list (cdr old-list))
          (new-list nil (cons (car old-list) new-list)))
        ((eq old-list sublist) (nreverse new-list))))
(setq a '(1 2 3 4 5 6 7))
(setq b (cddddr a))

(ldiff-if-sublist a b) => (1 2 3 4)

In the following example, tailp checks that the setf of cdr of one list will not affect another:

(if (tailp list1 list2)
    (setf (cdr (setq list1 (copy-list list1))) foo)
    (setf (cdr list1) foo))

For a table of related items: See the section "Predicates that Operate on Lists".

**tan** *radians* Function

Returns the tangent of *radians*. Examples:

(tan 0) => 0.0
(tan (/ pi 4)) => 1.0d0

For a table of related items: See the section "Trigonometric and Related Functions".

**tand** *degrees* Function

Returns the tangent of *degrees*.

For a table of related items: See the section "Trigonometric and Related Functions".

**tanh** *radians* Function

Returns the hyperbolic tangent of *radians*. Example:

(tanh 0) => 0.0

For a table of related items: See the section "Hyperbolic Functions".

**tenth** *list* Function

Returns the tenth element of *list*. *tenth* is equivalent to

(nth 9 list)

Example:

(setq letters '(a b c d e f g h i j k l)) =>
(A B C D E F G H I J K L)

(tenth letters) => J
For a table of related items: See the section "Functions for Extracting from Lists".

*terminal-io*  
Variable

The value of is ordinarily the stream that connects to the user's console. Under Genera in an "interactive" program, it is the window from which the program is being run; I/O on this stream reads from the keyboard and displays on the terminal. However, in a "background" program that normally does not talk to the user, *terminal-io* defaults to a stream that does not expect to be used. If it is used, perhaps by an error notification, it turns into a "background" window and requests the user's attention.

Although it is common practice to redirect *terminal-io* in Genera, this variable should not be redirected in a CLOE environment. Redirecting some, or even all of the following variables is usually sufficient: *standard-input*, *standard-output*, *error-output*, *trace-output*, *query-io*, and *debug-io*. If the values of any of these variables are changed, they can be restored to write to or get input from the user console by setting their values to synonym streams of *terminal-io*. System and other clean-up functions for CLOE assume that *terminal-io* has not been redirected.

(setq *standard-output* *terminal-io*)

zl:terminal-io  
Variable

In your new programs, we recommend that you use the variable *terminal-io*, which is the Common Lisp equivalent of zl:terminal-io.

The value of zl:terminal-io is the stream that connects to the user's console. In an "interactive" program, it is the window from which the program is being run; I/O on this stream reads from the keyboard and displays on the terminal. However, in a "background" program that does not normally talk to the user, zl:terminal-io defaults to a stream that does not ever expect to be used. If it is used, perhaps by an error notification, it turns into a "background" window and requests the user's attention.

terpri  
&optional output-stream

Function

Outputs a newline to output-stream, and returns nil. It is identical in effect to:

(write-char #\Newline output-stream)

output-stream, which, if unspecified or nil, defaults to *standard-input*, and if t, is *terminal-io*.

(progn (princ 'a) (princ 'b) (terpri) (princ 'c) nil)

AB
C
=> NIL
**zl:terpri** &optional stream

Outputs a carriage return character to stream.

**the type form**

Declares that the value of form is of type type. This allows you to declare the type of a value returned by an unnamed form.

```lisp
(the string (copy-seq x)) ;the result will be a string.
(the integer (+ x 3)) ;the result of + will be an integer.
(+ (the integer x) 3) ;the value of x will be an integer.
```

See the section "Operators for Making Declarations".

The type specifier values can be used to indicate the types of a form that returns multiple values.

```
(the (values integer integer)(floor x y))
```

**thereis** keyword for loop

**thereis expr**

If expr evaluates non-null, the iteration is terminated and that value is returned, without running the epilogue code. If the loop terminates before expr is ever evaluated, the epilogue code is run and the loop returns nil.

**thereis expr** is like (or expr1 expr2 ...). If the loop terminates before expr is ever evaluated, **thereis** is like (or).

If you want a similar test, except that you want the epilogue code to run if expr evaluates non-null, use until.

Examples:

```
(defun loop-thereis (my-list)
  (loop for x in my-list
     finally (print "what are you going to do next ?")
     do
     (princ x) (princ " ")
     do
     and thereis (equal x 'a))) => LOOP-THEREIS

(loop-thereis '(b c a e)) => B C A T

(loop-thereis '(a a)) => A T
```

See the section "Aggregated Boolean Tests for loop".
third list

Function

Takes a list as an argument, and returns the third element of list. third is identical to:

(nth 2 list)

Example:

(setq letters '(a b c d e f g)) =>
(A B C D E F G)

(third letters) =>
C

For a table of related items: See the section "Functions for Extracting from Lists".

throw tag value

Special Form

Used with catch to make nonlocal exits. It first evaluates tag to obtain an object that is the "tag" of the throw. It next evaluates form and saves the (possibly multiple) values. It then finds the innermost catch (or in Genera, *catch) whose "tag" is eq to the "tag" that results from evaluating tag. It causes the catch (or zl:*catch) to abort the evaluation of its body forms and to return all values that result from evaluating form. In the process, dynamic variable bindings are undone back to the point of the catch, and any unwind-protect cleanup forms are executed. An error is signalled if no suitable catch is found.

The scope of the tags is dynamic. That is, the throw does not have to be lexically within the catch form; it is possible to throw out of a function that is called from inside a catch form.

The value of tag cannot be the symbol sys:unwind-protect-tag; that is reserved for internal use.

For example:

(catch 'done
  (ask-database <pattern>
    #'(lambda (x) (when (nice-p x)
        (throw 'done x))))))

Additionally, consider this example:

(catch 'foo (list 'a (catch 'bar (throw 'foo 'b)))) => B

(defvar *input-buffer* nil)

(defun parse (*input-buffer*)
  (catch 'parse-error
    (list 's (parse-np) (parse-vp))))
(defun parse-np (&aux (item (pop *input-buffer*)))
  (if (member item '(a an the))
      '(np (det item) (n ,(pop *input-buffer*)))
      (throw 'parse-error
             (format t "Problem with ~A in noun phrase.~" item))))

(defun parse-vp (&aux (item (pop *input-buffer*)))
  (if (member item '(eats sleeps runs))
      '(vp (v item))
      (throw 'parse-error
             (format t "Problem with ~A in verb phrase.~" item))))

(parse '((a man eats)) => (S (NP (DET A) (N MAN)) (VP (V EATS)))

(parse '((a man walks)) => NIL
  prints: Problem with WALKS in verb phrase.

For more information, see the section "Nonlocal Exit Functions".

zl:*throw tag value
  Function
  An obsolete version of throw that is supported for compatibility with Maclisp. It is
  equivalent to throw except that it causes the catch or zl:*catch to return only
  two values: the first is the result of evaluating form, and the second is the result
  of evaluating tag (the tag thrown to). See the special form throw.

  For a table of related items, see the section "Nonlocal Exit Functions".

zl:times &rest args
  Function
  Returns the product of its arguments. If there are no arguments, it returns 1,
  which is the identity for this operation.

  The following functions are synonyms of zl:times:
    *
    zl:*$
ure out what conditions are being signalled and by what function. You can set this variable to error to trace all error conditions, for example, or you can be more specific.

This variable replaces the zl:errset variable.

*trace-output*  
Variable

The value of *trace-output* is the stream on which the trace function prints its output.

```
(trace function-likely-to-cause-error) ;trace a function

(with-open-file (outstream "myfile" :direction :output)
  (let ((*standard-output* outstream)
        (*trace-output* outstream)) ;redirects *trace-output* to myfile.lisp
    ...
    (function-likely-to-cause-error));capture trace information in file
    ;end of let restores *trace-output*, etc.
    ...
    ;more forms
  ) ;end of with-open-file closes file
```

zl:trace-output  
Variable

In your new programs, we recommend that you use the variable *trace-output*, which is the Common Lisp equivalent of zl:trace-output.

The value of zl:trace-output is the stream on which the trace function prints its output.

flavor:transform-instance  
Generic Function

Offers a way for you to specify code that should be run when an instance is changed to new-flavor. Because flavor:transform-instance is a generic function, you can write a method for it. This generic function is not intended to be called directly; instead, you take advantage of it by writing methods for it. If any methods for the flavor:transform-instance generic function are defined for a given flavor, those methods are applied to an instance in two cases:

- When the function change-instance-flavor is used on the instance.
- When the flavor of the instance has been redefined (with defflavor) and the stored representation of the instance is changed.

It is sometimes desirable to perform some action to update each instance as it is transformed to the new flavor (when change-instance-flavor is used) or as it is transformed to the new definition of the flavor (when defflavor is used to redefine a flavor), beyond the actions the system ordinarily takes. For example, newly added
instance variables are initialized to the same values they would receive in newly created instances. Sometimes this is not the appropriate value, and you need to compute a value for the variable. To do this, you can define a method for the generic function `flavor:transform-instance`, with no arguments.

Note that methods for `flavor:transform-instance` cannot access any instance variables that are deleted. By the time the methods are run, any deleted instance variables have been removed from the instance. In this example, the "old" instance variables are ones that existed both in the old and the new format of the instance.

```lisp
(defun (flavor:transform-instance my-flavor) ()
  (unless (variable-boundp new-instance-variable)
    (setq new-instance-variable
      (f old-instance-variable-1 old-instance-variable-2))))
```

By default, `flavor:transform-instance` uses `:daemon` method combination. You can specify a different type of method combination for this generic function by giving the `:method-combination` option to the `defflavor` of the flavor involved. If you want all the methods defined by the various component flavors to run, you can either specify `:progn` method combination or use `:after` methods with the default `:daemon` method combination.

Note: You should be careful to allow for your method being called more than once, if the flavor is redefined several times. A method intended to be used for one particular redefinition of the flavor remains in the system and is used for all future redefinitions, unless you use Kill Definition (`m-X`) or `fundefine` to remove the definition of the method.

Depending on the purpose of the method, it might be necessary to redefine the flavor before compiling the method for `flavor:transform-instance`. For example, a method that initializes a new instance variable cannot be compiled until the flavor is redefined to contain that instance variable.

Note that if an instance is accessed after its flavor has been redefined and before you have defined a method for `flavor:transform-instance`, the method is not executed on that instance.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

### math:transpose-matrix

**Function**

`math:transpose-matrix matrix &optional into-matrix`

Transposes `matrix`. If `into-matrix` is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. `matrix` must be a two-dimensional array. `into-matrix`, if provided, must be two-dimensional and must be the size of the transpose of `matrix`.

### tree-equal

**Function**

`tree-equal x y &key test test-not`
Returns t if x and y are isomorphic trees with identical leaves, that is, if x and y are atoms that satisfy the predicate specified by the :test keyword, or if they are both conses and their cars are tree-equal and their cdrs are tree-equal. Thus tree-equal recursively compares conses, but not any other objects that have components. The equal function compares certain other structured objects, such as strings. For example:

\[(\text{tree-equal } '(a b c) '(a b c)) \Rightarrow T\]

\[(\text{tree-equal } '(a b c) '(b c a)) \Rightarrow \text{NIL}\]

The keywords are:

: **test**

Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If :test is not supplied, the default operation is eql.

: **test-not**

Similar to :test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

\[(\text{tree-equal } 1 1) \Rightarrow (\text{eql } 1 1) \Rightarrow t\]

\[(\text{tree-equal } 1 '(1)) \Rightarrow \text{NIL}\]

The second form in the previous example is false because the structure of the two arguments to tree-equal are different. For the next few examples, we define a, b and c as follows:

\[(\text{setq } a '("root" "leaf1" "leaf2"))\]

\[(\text{setq } b '("root" "leaf1" "leaf2"))\]

\[(\text{setq } c '("Root" "Leaf1" "Leaf2"))\]

The first of the following forms is false, because the leaves are not eql. The second form is true because the test changed to string=. The third form is false, because string= is case sensitive, and the fourth is true because string-equal ignores case differences.

\[(\text{tree-equal } a b) \Rightarrow \text{NIL}\]

\[(\text{tree-equal } a b : \text{test } #\text{’string=} \Rightarrow t\]

\[(\text{tree-equal } a c : \text{test } #\text{’string=} \Rightarrow \text{NIL}\]

\[(\text{tree-equal } a b : \text{test } #\text{’string-equal}) \Rightarrow t\]

For a table of related items: See the section "Predicates that Operate on Lists".

**true &rest ignore**

Function

Takes no arguments and returns t. See the section "Functions and Special Forms for Constant Values".

**:truename**

Message
Returns the pathname of the file actually open on this stream. This can be different from what :pathname returns because of file links, logical devices, mapping of "newest" version to a particular version number, and so on. For some systems (such as ITS) the truename of an output stream is not meaningful until after the stream has been closed, at least on an ITS file server.

**truncate number &optional (divisor 1)**

Function

Divides number by divisor, and truncates the result toward zero. The truncated result and the remainder are the returned values.

number and divisor must each be a noncomplex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are Q and R, then (+ (* Q divisor) R) equals number. If divisor is 1, then Q and R add up to number. If divisor is 1 and number is an integer, then the returned values are number and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, the second returned value is always a number of the same type as the argument.

Examples:

- (truncate 5) => 5 and 0
- (truncate -5) => -5 and 0
- (truncate 5.2) => 5 and 0.19999981
- (truncate -5.2) => -5 and -0.19999981
- (truncate 5.8) => 5 and 0.8000002
- (truncate -5.8) => -5 and -0.8000002
- (truncate 5 3) => 1 and 2
- (truncate -5 3) => -1 and -2
- (truncate 5 4) => 1 and 1
- (truncate -5 4) => -1 and -1
- (truncate 5.2 3) => 1 and 2.1999998
- (truncate -5.2 3) => -1 and -2.1999998
- (truncate 5.2 4) => 1 and 1.1999998
- (truncate -5.2 4) => -1 and -1.1999998
- (truncate 5.8 3) => 1 and 2.8000002
- (truncate -5.8 3) => -1 and -2.8000002
- (truncate 5.8 4) => 1 and 1.8000002
- (truncate -5.8 4) => -1 and -1.8000002

For a table of related items: See the section "Functions that Divide and Convert Quotient to Integer".
Gets the next character from the stream and returns it. For example, if the next character to be read in by the stream is a "C", the following form returns #\C:

(send s :tyi)

Note that the :tyi operation does not "echo" the character in any fashion; it only does the input. The zl:tyi function echoes when reading from the terminal.

The optional eof argument to the :tyi message tells the stream what to do if it reaches the end of the file. If the argument is not provided or is nil, the stream returns nil at the end of file. Otherwise it signals an error and prints out the argument as the error message. Note that this is not the same as the eof-option argument to read, zl:tyi, and related functions.

The :tyi operation on a binary input stream returns a nonnegative number, not necessarily to be interpreted as a character.

An EOF can be forced into the currently selected I/O buffer with the keystrokes FUNCTION END. The next :tyi message sent to a window taking input from that I/O buffer will return nil.

The EOF indicator is not "sticky", in that the next :tyi will take the next character from the I/O buffer. The reason for this is that some programs which read only from the terminal might not be prepared to encounter an EOF, and might loop trying to read input.

This EOF feature makes it possible to fully test programs which use the :line-in, :string-in, and :string-line-in operations by taking input from a window instead of from a file. Typing FUNCTION END causes each of these operations to return. This is especially important when debugging programs which use the :string-in operation, since :string-in returns only when its buffer is full or an EOF is encountered.

FUNCTION END activates any input buffered in the input editor, since there is no representation for the EOF indicator within text strings.

zl:tyi &optional stream eof-option

Function

Inputs one character from stream and returns it. The character is echoed if stream is interactive, except that Rubout is not echoed. The Control, Meta, and so on shifts echo as prefix c-, m-, and so on.

The :tyi stream operation is preferred over the zl:tyi function for some purposes. Note that it does not echo. See the message :tyi.

(This function can take its arguments in the other order, for Maclisp compatibility only)

:tyi-no-hang &optional eof

Message

Identical to :tyi except that if it would be necessary to wait in order to get the character, returns nil instead. This lets the caller efficiently check for input being available and get the input if there is any. :tyi-no-hang is different from :listen
because it reads a character and because it is not simulated by the default handler
for streams that do not support it.

:tyipeek &optional eof

Message

On an input stream, returns the next character that is about to be read, or nil if
the stream is at end-of-file. The eof argument has the same meaning as it does for
:tyi. :tyipeek is defined to have the same effect as a :tyi operation, followed by a
:untyi operation if end-of-file is not reached. Note that this means that you cannot
read some character, do a :tyipeek to look at the next character, and then :untyi
the original character.

zl:tyipeek &optional peek-type stream eof-option

Function

Provided mainly for Maclisp compatibility; the :tyipeek stream operation is usually
preferred.

What zl:tyipeek does depends on the peek-type, which defaults to nil. With a peek-
type of nil, zl:tyipeek returns the next character to be read from stream, without
actually removing it from the input stream. The next time input is done from
stream the character is still there; in general, (= (zl:tyipeek) (zl:tyi)) is t. See the
message :tyipeek.

If peek-type is an integer less than 1000 octal, zl:tyipeek reads characters from
stream until it gets one equal to peek-type. That character is not removed from the
input stream.

If peek-type is t, zl:tyipeek skips over input characters until the start of the print-
ed representation of a Lisp object is reached. As above, the last character (the one
that starts an object) is not removed from the input stream.

The form of zl:tyipeek supported by Maclisp, in which peek-type is an integer not
less than 1000 octal, is not supported, since the readable formats of the Maclisp
reader and the Symbolics Common Lisp reader are quite different.

Characters passed over by zl:tyipeek are echoed if stream is interactive.

:tyo char

Message

Puts the char into the stream. For example, if s is bound to a stream, then the
following form will output a "B" to the stream:

    (send s :tyo #\B)

For binary output streams, the argument is a nonnegative number rather than
specifically a character.

zl:tyo char &optional stream

Function

Outputs the character char to stream.
**sys:type-arglist**  
*Function*

Takes a data type as its argument and checks whether *type* is a defined Common Lisp type.

**sys:type-arglist** returns two values: if *type* is a defined Common Lisp type, the first value is the lambda-list for specifiers for that *type*, if any, or **nil**; the second value is **t**. If *type* is not a defined Common Lisp type, both values are **nil**.

**sys:type-arglist** is useful if you are building software to run on top of the Common Lisp type system.

Examples:

```lisp
(sys:type-arglist 'integer) => (&OPTIONAL (LOW '*)) (HIGH '*)) and T
(sys:type-arglist 'array) => (&OPTIONAL (ELEMENT-TYPE '*)) (DIMENSIONS '*)) and T
(sys:type-arglist 'single-float) => NIL and T
(sys:type-arglist 'foo) => NIL
```

See the section "Data Types and Type Specifiers".

**type-of**  
*Function*

Returns a type of which *object* is a member. **type-of** returns the most specific type that can be conveniently computed and is likely to be useful to the user. If the argument is a user-defined structure created by **defstruct**, then **type-of** returns the name of that structure. If the argument is a user-created structure created by **defflavor** then **type-of** returns the type **symbol**. (**type-of instance**) returns the symbol that is the name of the instance's flavor.

Examples:

```lisp
(type-of 4) => FIXNUM
(type-of "Ariadne's thread") => STRING
(type-of 5/7) => RATIO
```

The following CLOE Runtime example begins with a request to make a 10 element vector of floats. Then, the type of **new-array**, and its initialized elements, is requested.

```lisp
(setq new-array (make-array 10 :element-type 'float))
(type-of new-array) => ARRAY
(type-of (aref new-array 0)) => NULL
(array-element-type new-array) => T
```

The returned type specifier is simply **array**, rather than (**array float (10)**), and the array elements were initialized to **nil**. Application of **array-element-type** on **new-array** reveals that there is no restriction on the type of the contents.

See the section "Data Types and Type Specifiers".
**typecase object &body body**

*Special Form*

This is a conditional that chooses one of its clauses by examining the type of an object. Structurally `typecase` is much like `cond` or `case`, and it behaves like them in selecting one clause and then executing all consequences of that clause. It differs in the mechanism of clause selection.

Its form is as follows:

```
(typecase form
  (type consequent consequent ...)
  (type consequent consequent ...)
  ...
)
```

The following example approximates a possible implementation of `zl-user:constantp` using `zl:typecase`.

```
(defun constantp (object)
  (typecase object
    (consp (eq (car object) 'quote))
    ((not symbol) t)
    (null t)
    ((satisfies #'(lambda(x)(eq x t))) t)
    ((satisfies keywordp) t)
    ((satisfies defined-constant-p) t)
    (otherwise nil)))
```

First `typecase` evaluates `form`, producing an object. `typecase` then examines each clause in sequence. The `type` that appears in each clause is a type specifier, which is not evaluated. If the object is of that type, the consequents are evaluated and the result of the last one is returned (or `nil` if there are no consequents in that clause). Otherwise, `typecase` moves on to the next clause. If no clause is satisfied, `typecase` returns `nil`.

For an object to be of a given type means that if `typep` is applied to the object and the type, it returns `t`. That is, a type is something meaningful as a second argument to `typep`. To specify more than one type in a clause, use the type specifier `$or$`:

```
(typecase form
  (type consequent consequent ...)
  ((or type type ...) consequent consequent ...)
  ...
)
```

See the section "Data Types and Type Specifiers".

As a special case, the `type` can be `otherwise`; in this case, the clause is always executed, so this should be used only in the last clause.

It is permissible for more than one clause to specify a given type, particularly if one is a subtype of another; the earliest applicable clause is chosen. Thus, for `typecase`, the order of the clauses can affect the behavior of the construct.
For a table of related items: See the section "Conditional Functions".

CLOE Note: \texttt{zl:typecase} is a macro in CLOE.

\texttt{zl:typecase} \texttt{object} \&\texttt{body} \texttt{body} \quad \textit{Special Form}

Selects various forms to be evaluated depending on the type of some object. It is something like \texttt{select}. A \texttt{zl:typecase} form looks like:

\begin{verbatim}
(zl:typecase form
  (types consequent consequent ...)
  (types consequent consequent ...)
  ...
)
\end{verbatim}

\textit{form} is evaluated, producing an object. \texttt{zl:typecase} examines each clause in sequence. \textit{types} in each clause is either a single type (if it is a symbol) or a list of types. If the object is of that type, or of one of those types, the consequents are evaluated and the result of the last one is returned. Otherwise, \texttt{zl:typecase} moves on to the next clause. As a special case, \textit{types} can be \texttt{otherwise}; in this case, the clause is always executed, so this should be used only in the last clause. For an object to be of a given type means that if \texttt{zl:typep} is applied to the object and the type, it returns \texttt{t}. That is, a type is something meaningful as a second argument to \texttt{zl:typep}.

Examples:

\begin{verbatim}
(defun tell-about-car (x)
  (zl:typecase (car x)
    (string "string") => TELL-ABOUT-CAR
    (tell-about-car '("word" "more")) => "string"
    (tell-about-car '(a 1)) => NIL

(defun tell-about-car (x)
  (zl:typecase (car x)
    (fixnum "number.")
    ((or string symbol) "string or symbol.")
    (otherwise "I don't know.") => TELL-ABOUT-CAR
    (tell-about-car '(1 a)) => "number."
    (tell-about-car '(a 1)) => "string or symbol."
    (tell-about-car '("word" "more")) => "string or symbol."
    (tell-about-car '(1.0)) => "I don't know."
\end{verbatim}

For a table of related items: See the section "Conditional Functions".

See the special form \texttt{typecase}.

\texttt{typep object type} \quad \textit{Function}

\texttt{typep} returns \texttt{true} if the \texttt{object} is of \texttt{type}. Otherwise it returns \texttt{false}.
The predicate is true if object is of type type, and is false otherwise. Note that an object can be "of" more than one type, since one type can include another, or the types can overlap without inclusion.

_type_ can be any of the type specifiers discussed in the chapter on Data Types. See the section "Type Specifiers". The exception is that _type_ cannot be or contain a type specifier list whose first element is _function_ or _values_. A specifier of the form _satisfies fn_ is handled simply by applying the function _fn_ to _object_ (see _funcall_); the _object_ is considered to be of the specified type if the result is not _nil_.

(typenp instance 'flavor-name) returns t if the flavor of instance is named _flavor-name_ or contains that flavor as a direct or indirect component; it returns _nil_ otherwise.

Examples:

```lisp
(typep 'my-dog-rover 'common) => T
(typep 'a 'atom) => T
(typep 0 'bit) => T

(defstruct ship
  x-position
  y-position) => SHIP

(setq my-boat (make-ship)) => #S (SHIP :X-POSITION NIL :Y-POSITION NIL)

(typep my-boat '(structure ship)) => T
(typep my-boat 'vector) => T

(typep #(a b c) 'vector) => T
(typep #*1010 'bit-vector) => T
(typep 4 'number) => T
(typep #c(3 4) 'complex) => T
(typep 4 'bit-vector) => nil

(typep 12 'integer) => t
(typep 12 '(integer 0 7)) => nil
(typep 12 '(satisfies integerp)) => t
(typep "a" 'character) => nil
(typep "a" 'string) => t
(typep \\a 'character) => t
(typep "a" 'array) => t
```

See the section "Type-checking Differences Between Symbolics Common Lisp and Zetalisp". See the section "Data Types and Type Specifiers".

zl:typenp x &optional type

Function
This function is really two different functions. With one argument, \texttt{zl:typep} is not
really a predicate; it returns a symbol describing the type of its argument. With
two arguments, \texttt{zl:typep} is a predicate that returns \texttt{t} if \texttt{x} is of type \texttt{type}, and \texttt{nil}
otherwise. Note that an object can be "of" more than one type, since one type can
be a subset of another.

The symbols that can be returned by \texttt{zl:typep} of one argument are:

\begin{itemize}
  \item \texttt{:symbol} \hspace{1cm} \texttt{x} is a symbol.
  \item \texttt{:fixnum} \hspace{1cm} \texttt{x} is a fixnum (not a bignum).
  \item \texttt{:bignum} \hspace{1cm} \texttt{x} is a bignum.
  \item \texttt{:rational} \hspace{1cm} \texttt{x} is a ratio.
  \item \texttt{:single-float} \hspace{1cm} \texttt{x} is a single-precision floating-point number.
  \item \texttt{:double-float} \hspace{1cm} \texttt{x} is a double-precision floating-point number.
  \item \texttt{:complex} \hspace{1cm} \texttt{x} is a complex number.
  \item \texttt{:list} \hspace{1cm} \texttt{x} is a cons.
  \item \texttt{:locative} \hspace{1cm} \texttt{x} is a locative pointer.
  \item \texttt{:compiled-function} \hspace{1cm} \texttt{x} is the machine code for a compiled function.
  \item \texttt{:closure} \hspace{1cm} \texttt{x} is a closure.
  \item \texttt{:select-method} \hspace{1cm} \texttt{x} is a select-method table.
  \item \texttt{:stack-group} \hspace{1cm} \texttt{x} is a stack-group.
  \item \texttt{:character} \hspace{1cm} \texttt{x} is a character.
  \item \texttt{:string} \hspace{1cm} \texttt{x} is a string.
  \item \texttt{:array} \hspace{1cm} \texttt{x} is an array that is not a string.
  \item \texttt{:random} \hspace{1cm} Returned for any built-in data type that does not fit into one of
                     the above categories.
  \item \texttt{foo} \hspace{1cm} An object of user-defined data type \texttt{foo} (any symbol). The prime-
                     tive type of the object could be array, or instance.
\end{itemize}

\textbf{(zl:typep instance)} returns the symbol that is the name of the instance's flavor.

\textbf{(zl:typep instance \texttt{flavor-name})} returns \texttt{t} if the flavor of \texttt{instance} is named \texttt{flavor-
name} or contains that flavor as a direct or indirect component, \texttt{nil} otherwise.

Examples:
The `type` argument to `zl:typep` of two arguments can be any of the above keyword symbols (except for :random), the name of a user-defined data type (either a named structure or a flavor), or one of the following additional symbols:

- **:atom** Any atom (as determined by the `atom` predicate).
- **:fix** Any kind of fixed-point number (fixnum or bignum).
- **:float** Any kind of floating-point number (single- or double-precision).
- **:number** Any kind of number.
- **:non-complex-number** Any noncomplex number.
- **:instance** An instance of any flavor.
- **:null** `nil` is the only value that has this type.
- **:list-or-nil** A cons or `nil`.

Examples:

```lisp
(zl:typep 3 :number) => T
(zl:typep nil :null) => T
(zl:typep '(a b c) :list-or-nil) => T
```

Note that (`zl:typep nil`) => :symbol, and (`zl:typep nil :list`) => nil; the latter might be changed.

```lisp
(zl:typep nil :list) => NIL
```

```
defflavor ship
  (name x-velocity y-velocity z-velocity mass)
  () ; no component flavors
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables) => SHIP
(setq my-ship
  (make-instance 'ship :name "Enterprise"
                :mass 4534
                :x-velocity 24
                :y-velocity 2
                :z-velocity 45)) => #<SHIP 43004623>
```

```lisp
(zl:typep my-ship :instance) => T
(zl:typep my-ship) => SHIP
(type-of my-ship) => SHIP
```
See the section "Type-checking Differences Between Symbolics Common Lisp and Zetalisp".

`unbreakon` &optional `function` (condition `t`)  

Function

Turns off a breakpoint set by `breakon`. If `function` is not provided, all breakpoints set by `breakon` are turned off. If `condition` is provided, it turns off only that condition, leaving any others. If `condition` is not provided, the entire breakpoint is turned off for that function.

For a table of related items: See the section "Breakpoint Functions".

`:unclaimed-message` `operation` &rest `arguments`  

Message

When an `operation` is performed on a flavor instance, whether the operation is a generic function or a message, the Flavors system checks to be sure that a method exists for performing the operation on the object. If no method is found, it checks for a method for the `:unclaimed-message` message. If such a method exists, it is invoked with arguments `operation` and any arguments that were given to the operation.

This is equivalent to using the `:default-handler` option to `defflavor`.

`flavor:vanilla` does not provide a method for `:unclaimed-message`. If no method for `:unclaimed-message` exists, and the `:default-handler` option was not used, then the default action of the Flavors system is to signal an error.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

`undefine-global-handler` `name`  

Function

Removes a global handler defined with `define-global-handler`.

`name` is the name of the global handler to be removed.

`undefine-global-handler` returns `t` if it finds the named handler. Otherwise it signals a proceedable error, and, if the condition proceeds, returns `nil`.

Examples:

```
(define-global-handler infinity-is-three sys:divide-by-zero
  (error)
  (values :return-values '(3)))

(undefine-global-handler infinity-is-three)
```

For a table of related items: See the section "Basic Forms for Global Handlers".
**undefun function-spec**

Undoes the definition of `function-spec` and returns `function-spec`. If `function-spec` has a saved previous basic definition, this interchanges the current and previous basic definitions, leaving the encapsulations alone. This undoes the effect of a **defun**, **compile**, and so on. (See the function **uncompile**.) If `function-spec` has no previous definition, **undefun** is equivalent to **fundefine**. If **undefun** does not find a definition for `function-spec`, it returns **nil**.

**si:unencapsulate-function-spec function-spec &optional encapsulation-types**

Takes one function spec and returns another. If the original function spec is undefined, or has only a basic definition (that is, its definition is not an encapsulation), then the original function spec is returned unchanged.

If the definition of `function-spec` is an encapsulation, its debugging info is examined to find the uninterned symbol that holds the encapsulated definition, and also the encapsulation type. If the encapsulation is of a type that is to be skipped over, the uninterned symbol replaces the original function spec and the process repeats.

The value returned is the uninterned symbol from inside the last encapsulation skipped. This uninterned symbol is the first one that does not have a definition that is an encapsulation that should be skipped. Or the value can be `function-spec` if `function-spec`'s definition is not an encapsulation that should be skipped.

The types of encapsulations to be skipped over are specified by **encapsulation-types**. This can be a list of the types to be skipped, or **nil**, meaning skip all encapsulations (this is the default). Skipping all encapsulations means returning the uninterned symbol that holds the basic definition of `function-spec`. That is, the definition of the function spec returned is the basic definition of the function spec supplied. Thus:

```
(fdefinition (si:unencapsulate-function-spec 'foo))
```

returns the basic definition of `foo`, and:

```
(fdefine (si:unencapsulate-function-spec 'foo) 'bar)
```

sets the basic definition (just like using **fdefine** with carefully supplied as t).

**encapsulation-types** can also be a symbol, which should be an encapsulation type; then we skip all types that are supposed to come outside of the specified type. For example, if **encapsulation-types** is **trace**, we skip all types of encapsulations that come outside **trace** encapsulations, but we do not skip **trace** encapsulations themselves. The result is a function spec that is where the **trace** encapsulation ought to be, if there is one. Either the definition of this function spec is a **trace** encapsulation, or there is no **trace** encapsulation anywhere in the definition of `function-spec`, and this function spec is where it would belong if there were one. For example:

```
(let (((tem (si:unencapsulate-function-spec spec 'trace)))
     (and (eq tem (si:unencapsulate-function-spec tem '(trace))
          (si:encapsulate tem spec 'trace '(...body...))))
```
finds the place where a trace encapsulation ought to go, and makes one unless there is already one there.

(let ((tem (si:unencapsulate-function-spec spec 'trace)))
  (fdefine tem (fdefinition (si:unencapsulate-function-spec tem '(trace)))))

eliminates any trace encapsulation by replacing it by whatever it encapsulates. (If there is no trace encapsulation, this code changes nothing.)

These examples show how a subsystem can insert its own type of encapsulation in the proper sequence without knowing the names of any other types of encapsulations. Only the si:encapsulation-standard-order variable, which is used by si:unencapsulate-function-spec, knows the order.

unexport symbols &optional package

Function

symbols should be a list of symbols or a single symbol. If symbols is nil, it is treated like an empty list. These symbols become internal symbols in package. package can be a package object or the name of a package (a symbol or a string). If unspecified, package defaults to the value of *package*. Returns t. It is an error to unexport a symbol from the keyword package.

=> (multiple-value-bind (symbol status) (find-symbol "exp-symbol")
    (when (eq status ' :external))
      (unexport symbol)))

=> T

unintern sym &optional (pkg (symbol-package si:sym))

Function

Removes sym from pkg and from pkg's shadowing-symbols list. If pkg is the home package for sym, sym is made to have no home package. In some circumstances, sym may continue to be accessible by inheritance. unintern returns t if it removes a symbol and nil if it fails to remove a symbol. unintern should be used with caution since it changes the state of the package system and affects the consistency rules. (See the section "Consistency Rules for Packages").

Compatibility Note: Symbolics Common Lisp under Genera specifies that this function's second argument defaults to symbol-package; CLtL and CLOE specify that this function's second argument defaults to *package*.

In the following example, the symbol whose print name is "one-symbol" is uninterned. In the second attempt to unintern the symbol, it is not found, and nil is returned.

=> (setq symbol (find-symbol "one-symbol"))
ONE-SYMBOL
=> (unintern symbol)
T
=> (unintern symbol)
nil
union list1 list2 &key (test #eql) test-not (key #identity)  \[\text{Function}\]

Takes two lists and returns a new list containing everything that is an element of either list1 or list2. If there is a duplication between the two lists, only one of the duplicate instances is in the result. If either of the arguments has duplicate entries within it, the redundant entries may or may not appear in the result. There is no guarantee that the order of the elements in the result will reflect the ordering of the arguments in any particular way. The keywords are:

: test
  Any predicate that specifies a binary operation on a supplied argument and an element of a target list. The item matches the specification only if the predicate returns t. If : test is not supplied, the default operation is eql.

: test-not
  Similar to : test, except that item matches the specification only if there is an element of the list for which the predicate returns nil.

For all possible ordered pairs consisting of one element from list1 and one element from list2, the predicate is used to determine whether they match. For every matching pair, at least one element of the pair will be in the result. Moreover, any element from either list that matches no element of the other will appear in the result.

\[
\begin{align*}
\text{union '('a b c) 'f a d a))} & \Rightarrow (D F A B C) \\
\text{union '(((x 5) (y 6) (x 3)) '((z 2) (x 4)) :key #'car) } & \Rightarrow ((Z 2) (X 5) (Y 6) (X 3))
\end{align*}
\]

In the following example, union returns the list of lists of all new and tenured professors and the courses they are teaching:

\[
\begin{align*}
\text{setq professors-with-tenure '('("Jones" CS101 CS242)("smith" CS202 CS231) }
& ("parks" CS221)("hunter" CS216 CS232))) \\
\text{setq new-professors '('("Able" CS101 CS244)("Cain" CS101 CS331) }
& ("Parks" CS221)("adams" CS215 CS222))) \\
\text{union professors-with-tenure new-professors :test #'string-equal :key #'car) } & \Rightarrow \\
& ("Jones" CS201 CS242)("smith" CS202 CS231) \\
& ("Hunter" CS216 CS232)("Able" CS203 CS244) \\
& ("Cain" CS212 CS331)("Parks" CS221) \\
& ("adams" CS215 CS222))
\end{align*}
\]

For a table of related items: See the section "Functions for Comparing Lists".

zl:union &rest lists \[\text{Function}\]
Takes any number of lists that represent sets and returns a new list that repre-
sents the union of all the sets. \texttt{zl:union} uses \texttt{eq} for its comparisons. You cannot
change the function used for the comparison. \texttt{zl:union} with no arguments returns\texttt{nil}.

For a table of related items: See the section "Functions for Comparing Lists".

\textbf{unless condition &rest body} \hspace{1cm} \textit{Macro}

The forms in \texttt{body} are evaluated when \texttt{condition} returns \texttt{nil}. It returns the value
of the last form evaluated. When \texttt{condition} returns something other than \texttt{nil},
\texttt{unless} returns \texttt{nil}.

Examples:

\begin{verbatim}
(unless) => error
(unless nil "rain, rain, rain") => "rain, rain, rain"
(unless (eq 1 1) (setq a b) "foo") => NIL
(unless (eq 1 2) (setq a 4) "foo") => “foo”
(a => 4

(defun make-even (integer)
  (unless (evenp integer) (setf integer (+ integer 1))))

(defvar *my-int* 5)
(make-even *my-int*) => 6
(make-even *my-int*) => nil
\end{verbatim}

Note that the following forms are equivalent, and that the \texttt{unless} version of these
may be more readable:

\begin{verbatim}
(if test nil (progn form1 form2 form3))
(when (not test) form1 form2 form3)
(unless test form1 form2 form3)
\end{verbatim}

When \texttt{body} is empty, \texttt{unless} always returns \texttt{nil}.

For a table of related items: See the section "Conditional Functions".

See the section \texttt{loop Conditionalization}.

\textbf{unless keyword for loop}

\textbf{unless expr}

If \texttt{expr} evaluates to \texttt{t}, the following clause is skipped, otherwise not. This
is equivalent to \texttt{when (not expr)}.

Examples:
(defun loop1 ()
  (loop for i from 0 to 9
       unless (> i 5) collect i
       finally (print " so long, goodbye...."))) => LOOP1
(loop1) =>
"so long, goodbye...." (0 1 2 3 4 5)

While the keyword **when** would do the following.

(defun loop1 ()
  (loop for i from 0 to 9
       when (> i 5) collect i
       finally (print " so long, goodbye...."))) => LOOP1
(loop1) =>
" so long, goodbye...." (6 7 8 9)

Multiple conditionalization clauses can appear in sequence. If one test fails, any following tests in the immediate sequence, and the clause being conditionalized, are skipped.

In the typical format of a conditionalized clause such as

    when expr1 keyword expr2

expr2 can be the keyword **it**. If that is the case, then a variable is generated to hold the value of expr1, and that variable gets substituted for expr2. Thus, the composition:

    when expr return it

is equivalent to the clause:

    thereis expr

and you can collect all non-null values in an iteration by saying:

    when expression collect it

If multiple clauses are joined with **and**, the **it** keyword can only be used in the first. If multiple **whens**, **unlesses**, and/or **if s** occur in sequence, the value substituted for **it** is that of the last test performed. The **it** keyword is not recognized in an **else**-phrase.

Conditionals can be nested.

See the section "loop Conditionalization".

**unread-char** character &optional input-stream Function

Puts character onto the front of input-stream. character must be the same character that was most recently read from input-stream. input-stream backs up over this character, so that when a character is next read from input-stream it will be the specified character. Successive calls to **read-char** will pick up the previous contents of input-stream, as it was before the call to **unread-char**. unread-char returns nil.
You can apply `unread-char` only to the character most recently read from `input-stream`. Moreover, you can not invoke `unread-char` twice consecutively without an intervening `read-char` operation. The result is that you can back up only by one character, and you can not insert any characters into the input stream that were not already there.

If unspecified or `nil`, `input-stream` defaults to `*standard-input*`. A value of `t` for `input-stream` indicates `*terminal-io*.

```lisp
(let ((c (read-char)))
  (unread-char c)
  (list c (read))) abc
=> (#\a ABC)
```

### unsigned-byte

`unsigned-byte` is the type specifier denoting the set on non-negative integers that can be represented in a byte of size `n` bits. It is the same as the type `(integer 0 *)`, the set of non-negative integers.

### until Keyword for loop

#### until `expr`

If `expr` evaluates to `t`, the loop is exited, performing exit code (if any), and returning any accumulated value. The test is placed in the body of the loop where it is written. It can appear between sequential `for` clauses.

Examples:

```lisp
(defun trivial-loop ()
  (loop for i from 0 until (= i 12)
        do
       (princ i) (princ " "))
 => TRIVIAL-LOOP
 (TRIVIAL-LOOP) => 0 1 2 3 4 5 6 7 8 9 10 11 NIL
```

See the section "End Tests for `loop`."

### :untyi char

The stream will remember the character `char`, and the next time a character is input, it will return the saved character. In other words, `:untyi` means "put this character back into the input source". For example:
(setq *my-stream* (make-string-input-stream "This is a test"))
(send *my-stream* :tyi) ==> #\T
(setq *char* (send *my-stream* :tyi)) ==> #\h
(send *my-stream* :untyi *char*) ==> 1
(send *my-stream* :tyi) ==> #\h

This operation is used by read, and any stream that supports :tyi must support :untyi as well. Note that you are allowed to :untyi only one character before doing a :tyi, and you can :untyi only the last character you read from the stream. Some streams implement :untyi by saving the character, while others implement it by backing up the pointer to a buffer. You also cannot :untyi after you have peeked ahead with :tyipeek.

:untyo mark

Message

Used by the grinder in conjunction with :untyo-mark. It takes one argument, which is something returned by the :untyo-mark operation of the stream. The stream should back up output to the point at which the object was returned.

:untyo-mark

Message

Used by the grinder if the output stream supports it. See the special form grinddef. It takes no arguments. The stream should return some object that indicates where output has reached in the stream.

unuse-package packages &optional pkg

Function

packages should be a list of packages or package names, or a single package or package name. These packages are removed from the use-list of pkg, and their external symbols are no longer accessible, unless they are accessible through another path. pkg can be a package object or the name of a package (a symbol or a string). If unspecified, pkg defaults to the value of *package*. Returns t.

=> (package-use-list *package*)
(TURBINE-PACKAGE GENERATOR-PACKAGE LISP)
=> (unuse-package 'turbine-package)
T
=> (package-use-list *package*)
(GENERATOR-PACKAGE LISP)

See the section "Interpackage Relations".

unwind-protect protected-form &rest cleanup-forms

Special Form

Sometimes it is necessary to evaluate a form and make sure that certain side-effects take place after the form is evaluated. A typical example is:
(progn
  (turn-on-water-faucet)
  (hairy-function 3 nil 'foo)
  (turn-off-water-faucet))

The nonlocal exit facility of Lisp creates a situation in which the above code does not work. However, if hairy-function should do a throw to a catch that is outside of the progn form, (turn-off-water-faucet) is never evaluated (and the faucet is presumably left running). This is particularly likely if hairy-function gets an error and the user tells the Debugger to give up and abort the computation.

In order to allow the above program to work, it can be rewritten using unwind-protect as follows:

(unwind-protect
  (progn (turn-on-water-faucet)
         (hairy-function 3 nil 'foo))
  (turn-off-water-faucet))

If hairy-function does a throw that attempts to quit out of the evaluation of the unwind-protect, the (turn-off-water-faucet) form is evaluated in between the time of the throw and the time at which the catch returns. If the progn returns normally, then the (turn-off-water-faucet) is evaluated, and the unwind-protect returns the result of the progn.

Examples:

(tagbody
  (let ((num 4))
    (unwind-protect
      (if (= num 4) (go home))
      (princ "reach out")))
  home
  (princ " and ") => reach out and NIL

(unwind-protect
  (progn (start-car)
         (drive-car))
  (stop-car))

The general form of unwind-protect looks like:

(unwind-protect protected-form
  cleanup-form1
  cleanup-form2
  ...
)

protected-form is evaluated, and when it returns or when it attempts to quit out of the unwind-protect, the cleanup-forms are evaluated. To ensure that unwind-protect does not return without completely executing its cleanup forms, the macro sys:without-aborts is automatically and atomically wrapped around all cleanup-forms, preventing them from being aborted by user action. (To cancel out the effect of a sys:without-aborts invocation, see the macro sys:with-aborts-enabled.)
unwind-protect catches exits caused by return-from or go as well as those caused by throw. The value of the unwind-protect is the value of protected-form. Multiple values returned by the protected-form are propagated back through the unwind-protect.

The cleanup forms are run in the variable-binding environment that you would expect: that is, variables bound outside the scope of the unwind-protect special form can be accessed, but variables bound inside the protected-form cannot be. In other words, the stack is unwound to the point just outside the protected-form, then the cleanup handler is run, and then the stack is unwound some more.

Note: It is almost never adequate to do something of the form

```
(unwind-protect (progn (foo) ... code ...) (undo-foo))
```

Nearly always you should write

```
(let ((old-foo-state (read-foo-state)))
  (unwind-protect (progn (foo) ... code ...) (set-foo-state old-foo-state)))
```

You should also consider that other processes may see your data structure in the modified state. If you have a shared structure, you may need to use a lock to only allow one process to use it while it is modified.

```
(defmacro bind ((form value) &body body)
  "a powerful binding primitive guaranteed to restore the old value"
  (let ((old-value-var (gensym)))
    '(let ((,old-value-var ,form))
      '(let ((,old-value-var ,form))
        (unwind-protect (progn (setf ,form ,value)
                              ,body)
                        (setf ,form ,old-value-var))))))
```

For a table of related items, see the section "Nonlocal Exit Functions".

unwind-protect-case (&optional aborted-p-var) body-form &rest cleanup-clauses

Macro

body-form is executed inside an unwind-protect form. The cleanup forms of the unwind-protect are generated from cleanup-clauses. Each cleanup-clause is considered in order of appearance and has the form (keyword forms ...). keyword can be :normal, :abort or :always. The forms in a :normal clause are executed only if body-form finished normally. The forms in an :abort clause are executed only if body-form exited before completion. The forms in an :always clause are always executed. The values returned are the values of body-form, if it completed normally.

To ensure that unwind-protect-case does not return without completely executing its cleanup forms, the macro sys:without-aborts is automatically and atomically wrapped around all cleanup-forms, preventing them from being aborted by user action.
aborted-p-var, if supplied, is t if the body-form was aborted, and nil if it finished normally. aborted-p-var can be used in forms within cleanup-clauses as a condition for executing abort instead of normal cleanup code. It can be set within body-form, but should be done so with great care. It should only be set to nil if the remaining subforms of body-form do not need protecting.

For a table of related items, see the section "Nonlocal Exit Functions".

clos:update-instance-for-different-class previous current &rest initargs

Generic Function

Provides a mechanism for users to specialize the behavior of updating an instance when its class is changed by clos:change-class. This generic function is called by clos:change-class and should not be called by users.

Note that the usual way for users to customize the behavior of updating an instance for a different class is to specialize clos:update-instance-for-different-class by writing after-methods. A user-defined primary method would override the default method, and thus could prevent the usual slot-filling behavior.

The value of clos:update-instance-for-different-class is ignored by its caller, clos:change-class.

previous

A copy of the instance before its class was changed. The purpose of this argument is to enable methods to access the old slot values. It has dynamic extent within clos:change-class.

current

The instance whose class has been changed.

initargs

Alternating initialization argument names and values. Note that no initialization arguments are provided by the caller, clos:change-class. They can be supplied by one method to another method, using clos:call-next-method.

The set of valid initialization argument names includes:

- Symbols declared by the :initarg slot option to clos:defclass, which are used to initialize the value of a slot.
- Keyword arguments accepted by any applicable methods for clos:update-instance-for-different-class or clos:shared-initialize.
- The keyword :allow-other-keys. The default value for :allow-other-keys is nil. If you provide t as its value, then all keyword arguments are valid.

The default method for clos:update-instance-for-different-class does the following:

1. Checks the validity of the initargs and signals an error if an invalid initialization argument name is detected.
2. Calls the `clos:shared-initialize` generic function with the instance, a list of the newly added local slots, and any initialization arguments provided. The second argument indicates that only the newly added local slots are to be initialized from their initforms.

See the section "Changing the Class of a CLOS Instance".

clos:update-instance-for-redefined-class `instance added-slots discarded-slots property-list &rest initargs`  

*Generic Function*

Provides a mechanism for users to specialize the behavior of updating instances when a class is redefined.

This generic function should not be called directly by users; it is called by the system when a class is redefined or when `clos:make-instances-obsolete` is called. It is not necessarily called immediately in these cases; it is called at some time before a slot of that instance is read or written.

Note that the usual way for users to customize the behavior of updating instances when a class is redefined is to specialize `clos:update-instance-for-redefined-class` by writing after-methods. A user-defined primary method would override the default method, and thus could prevent the usual slot-filling behavior.

The value of `clos:update-instance-for-redefined-class` is ignored by its caller.

*instance*  
The instance being updated due to class redefinition.

*added-slots*  
A list of the names of slots added to the instance. An added slot is a local slot defined by the new class for which there was no slot of the same name defined in the previous class.

*discarded-slots*  
A list of the names of slots removed from the instance. A discarded slot is a slot that was defined by the previous class but not by the new class. Included in this list are slots defined as local in the previous class and shared in the new class.

*property-list*  
A property list containing the slot names and values for each discarded slot that had a value.

*initargs*  
Alternating initialization argument names and values. Note that no initialization arguments are provided by the caller. They can be supplied by one method to another method, using `clos:call-next-method`.

The set of valid initialization argument names includes:

- Symbols declared by the `:initarg` slot option to `clos:defclass`, which are used to initialize the value of a slot.

- Keyword arguments accepted by any applicable methods for `clos:update-instance-for-redefined-class` or `clos:shared-initialize`. 

The keyword :allow-other-keys. The default value for :allow-other-keys is nil. If you provide t as its value, then all keyword arguments are valid.

The default method for clos:update-instance-for-redefined-class does the following:

1. Checks the validity of the initargs and signals an error if an invalid initialization argument name is detected.

2. Calls the clos:shared-initialize generic function with the instance, the added-slots, and any initialization arguments provided. The second argument indicates that only the newly added local slots are to be initialized from their initforms.

See the section "Redefining a CLOS Class".

### upper-case-p char

Function

Returns t if char is an uppercase letter.

```lisp
(upper-case-p #\A) => T
(upper-case-p #\a) => T
```

For a table of related items, see the section "Character Predicates".

### use-package packages &optional pkg

Function

packages should be a list of packages or package names, or a single package or package name. These packages are added to the use-list of pkg if they are not there already. All external symbols in the packages to use become accessible in pkg. pkg can be a package object or the name of a package (a symbol or a string). If unspecified, pkg defaults to the value of *package*. Returns t.

The following function first checks if a package to be added to the use-list of another package is already on the list, before calling use-package.

```lisp
(defun add-to-use-list( package package-to-use )
  (unless (member package-to-use
                   (package-use-list package))
    (use-package package package-to-use)))
```

See the section "Interpackage Relations".

### zl:value-cell-location sym

Function

This function is obsolete on local and instance variables; use sys:variable-location instead.

zl:value-cell-location returns a locative pointer to sym’s internal value cell. See the section "Cells and Locatives". It is preferable to write:
(locf (zl:symeval sym))

instead of calling this function explicitly.

(zl:value-cell-location 'a) is still useful when a is a special variable. It behaves slightly differently from the form (sys:variable-location a), in the case that a is a variable "closed over" by some closure. See the section "Dynamic Closures". zl:value-cell-location returns a locative pointer to the internal value cell of the symbol (the one that holds the invisible pointer, which is the real value cell of the symbol), whereas sys:variable-location returns a locative pointer to the external value cell of the symbol (the one pointed to by the invisible pointer, which holds the actual value of the variable).

See the section "Functions Relating to the Value of a Symbol".

values &rest args

Function

Returns values, its arguments. This is the primitive function for controlling return values. It returns exactly one value for each form in its argument list. In this way you can assure that a function returns only one value. For example,

(floor 9 2) => 4 1

(values (floor 9 2)) => 4

floor returns two values. However, values returns only the first value produced by each form, so it returns the 4 and ignores the 2.

It is valid to call values with no arguments; it returns no values in that case.

(defun foo-pos (foo) (values (foo-x foo)(foo-y foo)))

In the next example, the call to add-to-end-just-for-effect returns no values.

(defun add-at-end-just-for-effect (list item)
 (setf (cdr (last list)) (cons item nil))
 (values))

(setq x '(a b c))

(add-to-end-just-for-effect x 'd)

x => (A B C D)

(defun add-at-end-return-old-and-new (list item &aux (old-list (copy-list list)))
 (setf (cdr (last list)) (cons item nil))
 (values list old-list))

(add-at-end-return-old-and-new x 'e) => (A B C D E) (A B C D)

See the section "Primitives for Producing Multiple Values".
values

values-list list

Returns multiple values, the elements of the list. (values-list '(a b c)) is the same as (values 'a 'b 'c). list can be nil, the empty list, which causes no values to be returned.

In the following example, the let returns as many values as original-list contained numbers greater than 5.

(let ((mylist '()))
  (dolist (item original-list)
    (when (> item 5) (push item mylist)))
  (values-list mylist))

See the section “Primitives for Producing Multiple Values”.

flavor:vanilla

This flavor is included in all flavors by default. flavor:vanilla has no instance variables, but it provides several basic useful methods, some of which are used by the Flavor tools.

Every flavor has flavor:vanilla as a component flavor, unless you specify not to include flavor:vanilla by providing the :no-vanilla-flavor option to defflavor. It is unusual to exclude flavor:vanilla.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section “Summary of Flavor Functions and Variables”.

variable-boundp variable

Returns t if the variable is bound and nil if the variable is not bound. variable should be any kind of variable (it is not evaluated): local, special, or instance. Note: local variables are always bound; if variable is local, the compiler issues a warning and replaces this form with t.

If a is a special variable, (boundp 'a) is the same as (variable-boundp a). See the section “Functions Relating to the Value of a Symbol”.

sys:variable-location variable

Returns a locative pointer to the memory cell that holds the value of the variable. variable can be any kind of variable (it is not evaluated): local, special, or instance.

sys:variable-location should be used in almost all cases instead of zl:value-cell-location; zl:value-cell-location should only be used when referring to the internal value cell. For more information on internal value cells: See the section “What is a Dynamic Closure?”. 
You can also use `locf` on variables. `(locf a)` expands into `(sys:variable-location a)`. See the section "Functions Relating to the Value of a Symbol".

**variable-makunbound** `variable`  
*Special Form*

Makes the variable be unbound and returns `variable`. `variable` should be any kind of variable (it is not evaluated): local, special, or instance. Note: since local variables are always bound, they cannot be made unbound; if `variable` is local, the compiler issues a warning.

If `a` is a special variable, `(makunbound 'a)` is the same as `(variable-makunbound 'a)`. See the section "Functions Relating to the Value of a Symbol".

**vector** &optional (`element-type '*'`) (`size '*'`)  
*TypeSpecifier*

`vector` is the type specifier symbol for the predefined Lisp structure of that name.

The type `vector` is a *subtype* of the type `array`: for all types of `x`, the type `(vector x)` is the same as the type `(array x (*)�)菜肴`.

The types `vector` and `list` are *disjoint subtypes* of the type `sequence`.

The type `vector` is a supertype of the types `string`, `bit-vector`, `simple-vector`;

- `string` means `(vector string-char)`, or `(vector character)`
- `bit-vector` means `(vector bit)`
- `simple-vector` means `(simple-array t (*))`

The types `vector t`, `string`, and `bit-vector` are *disjoint*.

This type specifier can be used in either symbol or list form. Used in list form, `vector` allows the declaration and creation of specialized one-dimensional arrays whose elements are all of type `element-type` and whose lengths match `size`. This is entirely equivalent to

`(array (element-type size))`  

`element-type` must be a valid type specifier, or unspecified. For standard Symbolics Common Lisp type specifiers: See the section "Type Specifiers".

`size` can be a non-negative integer, or it can be a list of non-negative integers, or it can be unspecified.

The specialized types `(vector string-char)` and `(vector bit)` are so useful that they have the special names `string` and `bit-vector`.

Examples:

```
(typep #(a b c) 'vector) => T
(subtypep 'vector 'array) => T and T
(subtypep 'vector 'sequence) => T and T
(sys:type-arglist 'vector)
=> (&optional (ELEMENT-TYPE '*)) (SIZE '*)) and T
```
(vectorp #()) => T
(typep #\010 '(vector bit 3)) => T

See the section "Data Types and Type Specifiers". See the section "Arrays".

vector &rest objects  

Creates a simple vector with specified initial contents and with the order given. For example:

(vector 12 'foo 42.9) => #( 12 FOO 42.9)

For a table of related items: See the section "Operations on Vectors".

sys:vector-bitblt alu size from-array from-index to-array to-index  

Copies a linear portion of from-array of length size starting at from-index into a linear portion of to-array starting at to-index. The value stored can be a Boolean function of the new value and the value already there, under the control of alu. This function is a one-dimensional bitblt. See the function bitblt.

from-array and to-array are allowed to be the same array. If size is negative, then the processing is done backwards, using (abs size) as the number of elements. For arrays of different elements it works bitwise, and size is in units of to-array.

sys:vector-bitblt might not work well if from-array is indirceted with an index-offset.

vector-pop array &optional default  

Decreases the fill pointer by one and returns the vector element designated by the new value of the fill pointer. array must be a one-dimensional array with a fill pointer. If the fill pointer is 0, nil is returned.

Symbolics Common Lisp provides the optional argument default, which might not work in other implementations of Common Lisp.

(setq some-vector (make-array 4 :initial-contents (list 12 18 (list 'a 'b) 'C) :fill-pointer t))

(vector-pop some-vector) => C

(vector-pop some-vector) => (A B)

(fill-pointer some-vector) => 2

For a table of related items: See the section "Operations on Vectors". Also: See the section "Adding to the End of an Array".

vector-push new-element vector  


 Stores new-element in the element designated by the fill pointer and increments
the fill pointer by one. vector must be a one-dimensional array with a fill-pointer,
and new-element can be any object allowed to be stored in the array.

If the fill pointer does not designate an element of the array (specifically, when it
gets too big), it is unaffected and vector-push returns nil. Otherwise, the two ac-
tions (storing and incrementing) happen uninterruptibly, and vector-push returns
the former value of the fill pointer, that is, the array index in which it stored new-
element.

For a table of related items: See the section "Operations on Vectors". Also: See the
section "Adding to the End of an Array".

vector-push-extend new-element vector &optional extension

Function

Stores new-element in the element designated by the fill pointer and increments
the fill pointer by one. If the vector is too small, vector-push-extend extends the
vector, it is adjustable. Note that under CLOE, only vectors specified to be ad-
justable in the call to make-array are in fact adjustable.

vector-push-extend returns the index in vector where new-element was stored.

(setq astring (make-array 12 :element-type 'string-char :fill-pointer t
   :adjustable t :initial-element \\).))
=> "............"

(fill-pointer astring) => 12
(array-dimension astring 0) => 12

(vector-push-extend \\a astring 10) => 12
astring => "...........a"
(fill-pointer astring) => 13
(array-dimension astring 0) => 22

(vector-push-extend \\b astring 100) => 13
astring => "...........ab"
(fill-pointer astring) => 14
(array-dimension astring 0) => 22

Note in the previous example that we use the extension argument of the first call
to vector-push-extend because only this call actually adjusts the array. The second
call places an element within the bounds of the newly adjusted array.

For a table of related items: See the section "Operations on Vectors". Also: See the
section "Adding to the End of an Array".

vector-push-portion-extend to-array from-array &optional (from-start 0) from-end

Function
Copies a portion of one array to the end of another, updating the fill pointer of the second to reflect the new contents. The destination array must have a fill-pointer. The source array need not.

**vector-push-portion-extend** returns the to-array and the index of the next location to be filled.

Example:

```lisp
(setq to-string
  (vector-push-portion-extend
   to-string from-string (or from 0) to))
```

If the optional arguments are not provided, the default is to copy all of from-array to the end of to-array.

For a table of related items: See the section "Operations on Vectors".

**vectorp object**

Tests whether the given object is a vector. A vector is a one-dimensional array. See the type specifier vector.

```lisp
(vectorp (make-array 5 :element-type 'bit :fill-pointer 2)) => T

(vectorp (make-array '(5 2))) => NIL

(vectorp '#(foo bar baz)) => t

(vectorp (make-array '(2 3)
  :initial-element 'foo)) => nil
```

For a table of related items: See the section "Operations on Vectors".

**warn optional-options optional-condition-name format-string &rest args**

If the flag *break-on-warnings* is nil, prints a warning message without entering the Debugger.

If the flag *break-on-warnings* is not nil, warn enters the Debugger and prints the warning message. If you continue from the error, warn returns args.

format-string is an error message string.

format-args are additional arguments; these are evaluated only if a condition is signalled.

Examples:
(defun sum-numbers (list-of-numbers)
  (when (< (length list-of-numbers) 2)
    (warn "You are trying to only add ~D number~:P." (length list-of-numbers)))
  (reduce #'+ list-of-numbers))

(sum-numbers '(1))
=> Warning: You are trying to only add 1 number.

(setq *break-on-warnings* t) => T

(sum-numbers '(1))=>
Warning: You are trying to only add 1 number

SUM-NUMBERS:
Arg 0 (LIST-OF-NUMBERS): (1)
Debugger was entered because *BREAK-ON-WARNINGS* is set
s-A, <RESUME>: Return from WARN
s-B: Proceed without any special action
s-C, <ABORT>: Return to Lisp Top Level in Dynamic Lisp Listener 1
            → Return from WARN

1

For a table of related items: See the section "Condition-Checking and Signalling Functions and Variables".

**what-files-call** symbol-or-symbols &optional how

<table>
<thead>
<tr>
<th>how</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Returns all ways to call symbol, a keyword, meaning only find symbol called as keyword, or a list of keywords.</td>
</tr>
<tr>
<td>variable</td>
<td>Uses symbol as a variable.</td>
</tr>
<tr>
<td>function</td>
<td>Calls symbol as a function.</td>
</tr>
<tr>
<td>microcoded-function</td>
<td>Calls symbol as an instruction. This is used on 3600-family machines only.</td>
</tr>
<tr>
<td>constant</td>
<td>Uses symbol as a constant.</td>
</tr>
<tr>
<td>instance-variable</td>
<td>Uses symbol as an instance variable.</td>
</tr>
<tr>
<td>macro</td>
<td>Uses symbol as a macro or optimized function.</td>
</tr>
<tr>
<td>defined-constant</td>
<td>Uses symbol as an open coded (defconstant) constant.</td>
</tr>
</tbody>
</table>
:condition Establishes a condition handler for symbol.

:flavor-component A dependent flavor of symbol.

:generic-function Calls symbol as a generic function.

:constructor Is a constructor function for symbol.

:setf Calls the setf function for symbol.

:locf Calls the locf function for symbol.

:presentation-translator-from A presentation translator from symbol.

:presentation-translator-to A presentation translator to symbol.

:defines-instance-variable A flavor that defines symbol as an instance variable.

**when condition &rest body**

Macro

The forms in body are evaluated when condition returns non-nil. In that case, it returns the value(s) of the last form evaluated. When condition returns nil, when returns nil.

Examples:

```
(when) => error
(when t "Climb Tree") => "Climb Tree"
(when (atom 'x) (setq a 1) "foo") => "foo"
  a => 1
(when (eq 1 2) "day" "night") => NIL
(defun make-even (integer)
  (when (oddp integer) (setf integer (+ integer 1))))

(make-even *my-int*) => 6
(make-even *my-int*) => nil
```

Note that the following forms are equivalent, and the when version of these may be more readable:

```
(if test (progn form1 form2 form3))
(unless (not test) form1 form2 form3)
(when test form1 form2 form3)
```

When body is empty, when always returns nil.

For a table of related items: See the section "Conditional Functions".
when keyword for loop

when expr

If expr evaluates to nil, the following clause is skipped, otherwise not.

Examples:

```
(defun loop1 ()
  (loop for i from 1 to 10
      when (= i 5 ) return i
  finally (print "Finally triggered"))) => LOOP1
(loop1) => 5
```

```
(defun loop1 ()
  (loop for i from 1
      when (> i 5 ) collect i
      until (> i 20))) => LOOP1
(loop1) => (6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21)
```

Multiple conditionalization clauses can appear in sequence. If one test fails, any following tests in the immediate sequence, and the clause being conditionalized, are skipped.

In the typical format of a conditionalized clause such as

```
when expr1 keyword expr2
```

expr2 can be the keyword it. If that is the case, then a variable is generated to hold the value of expr1, and that variable gets substituted for expr2. Thus, the composition:

```
when expr return it
```

is equivalent to the clause:

```
thereis expr
```

and one can collect all non-null values in an iteration by saying:

```
when expression collect it
```

If multiple clauses are joined with and, the it keyword can only be used in the first. If multiple whens, unlesses, and/or ifs occur in sequence, the value substituted for it is that of the last test performed. The it keyword is not recognized in an else-phrase.

Conditionals can be nested.

See the section "loop Conditionalization".

where-is pname

Function

Finds all symbols named pname and prints on *standard-output* a description of each symbol. The symbol's home package and name are printed. If the symbol is present in a different package than its home package (that is, it has been imported), that fact is printed. A list of the packages from which the symbol is accessible
is printed, in alphabetical order. **where-is** searches all packages that exist, except for invisible packages.

If `pname` is a string it is converted to uppercase, since most symbols’ names use uppercase letters. If `pname` is a symbol, its exact name is used.

**where-is** returns a list of the symbols it found.

The **find-all-symbols** function is the primitive that does what **where-is** does without printing anything.

### :which-operations

The object should return a list of the messages and names of generic functions for which it has methods.

The :**which-operations** method supplied by **flavor:vanilla** generates the list once per flavor and remembers it, minimizing consing and compute time. The list is regenerated when a new method is added.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

### while Keyword for loop

**while expr**

If `expr` evaluates to **nil**, the loop is exited, performing exit code (if any), and returning any accumulated value. The test is placed in the body of the loop where it is written. It can appear between sequential **for** clauses.

Examples:

```lisp
(defun x-power (x)
  (loop for stepper = x then (* stepper x)
    while (< stepper 100)
    do
      (print stepper))) => X-POWER
(x-power 3) =>
3
9
27
81 NIL
```

**who-calls** `symbol` &optional `how`

Tries to find all the functions in the Lisp world that call `symbol`. 
how may be nil, meaning all ways to call the symbol, a keyword, meaning only find symbol called as keyword, or a list of keywords. The permitted keywords are:

:variable Uses symbol as a variable.
:function Calls symbol as a function.
:microcoded-function Calls symbol as an instruction. This is used on 3600-family machines only.
:constant Uses symbol as a constant.
:instance-variable Uses symbol as an instance variable.
:macro Uses symbol as a macro or optimized function.
:defined-constant Uses symbol as an open coded (defconstant) constant.
:condition Establishes a condition handler for symbol.
:flavor-component A dependent flavor of symbol.
:generic-function Calls symbol as a generic function.
:constructor Is a constructor function for symbol.
:setf Calls the setf function for symbol.
:locf Calls the locf function for symbol.
:presentation-translator-from A presentation translator from symbol.
:presentation-translator-to A presentation translator to symbol.
:defines-instance-variable A flavor that defines symbol as an instance variable.

who-calls takes a single symbol as its argument.

who-calls prints one line of information for each caller it finds. It also returns a list of the names of all the callers.

who-calls works only on bound symbols. To locate unbound symbols: See the function si:who-calls-unbound-functions.

The compiler records, as part of its debugging-info property, which macros were expanded and which functions were optimized away, with the exception of basic parts of the language, such as car and when. This information is used by who-calls and similar functions. Thus you can use who-calls for macros. who-calls can also find callers of open-coded functions, such as substitutable functions.

The who-calls database is created at site configuration time using the function si:enable-who-calls. See the function si:enable-who-calls.

After you create the database, you should run si:compress-who-calls-database. See the function si:compress-who-calls-database.
The editor has a command, List Callers (m-X), that is similar to who-calls. There is also a Command Processor command:

See the section "Show Callers Command".

si:who-calls-unbound-functions

Function

Searches the compiled code for any calls through a symbol that is not currently defined as a function. This is useful for finding errors such as functions whose names you misspelled or forgot to write.

&whole

Lambda List Keyword

Used with macros only. It should be followed by a single variable that is bound to the entire macro-call form or subform. This variable is the value that the macro-expander function receives as its first argument. &whole and its following variable should appear first in the lambda-list, before any other parameter or lambda-list keyword.

with keyword for loop

with var1 {data-type} (= expr1} {and var2 {data-type} (= expr2)}...

The with keyword can be used to establish initial bindings, that is, variables that are local to the loop but are only set once, rather than on each iteration.

The optional argument, data-type, is reserved for data type declarations. It is currently ignored.

If no expr is given, the variable is initialized to the appropriate value for its data type, usually nil. with bindings linked by and are performed in parallel; those not linked are performed sequentially. That is:

(loop with a = (foo) and b = (bar) and c ...)

binds the variables like:

((lambda (a b c) ...) (foo) (bar) nil)

whereas:

(loop with a = (foo) with b = (bar a) with c ...)

binds the variables like:

((lambda (a)
   ((lambda (b)
     ((lambda (c) ...) nil))
     (bar a)))
   (foo))
All expr's in with clauses are evaluated in the order they are written, in lambda-expressions surrounding the generated prog. The loop expression:

```lisp
(loop with a = xa and b = xb
   with c = xc
   for d = xd then (f d)
   and e = xe then (g e d)
   for p in xp
   with q = xq
   ...)
```

produces the following binding contour, where t1 is a loop-generated temporary:

```lisp
(((lambda (a b)
   ((lambda (c)
     ((lambda (d e)
       ((lambda (p t1)
         ((lambda (q) ...
          xq))
       nil xp))
   xd xe))
   xc))
   xc))
xa xb)
```

Because all expressions in with clauses are evaluated during the variable-binding phase, they are best placed near the front of the loop form for stylistic reasons.

For binding more than one variable with no particular initialization, one can use the construct:

```lisp
with variable-list {data-type-list} {and ...}
```
as in:

```lisp
with (i j k t1 t2) (fixnum fixnum fixnum) ...
```
A slightly shorter way of writing this is:

```lisp
with (i j k) fixnum and (t1 t2) ...
```
These are cases of destructuring which loop handles specially. See the section "Destructuring".

Examples:
(defun loop1 ()
  (loop for x from 0 to 3
    with (a b)
    with c = '(its constant)
    with d = '(another constant)
    do
      (setq a (+ x 10))
      (setq b (+ x 20))
      (print (list a b c d)))) => LOOP1
(loop1) =>
  (10 20 (ITS CONSTANT) (ANOTHER CONSTANT))
  (11 21 (ITS CONSTANT) (ANOTHER CONSTANT))
  (12 22 (ITS CONSTANT) (ANOTHER CONSTANT))
  (13 23 (ITS CONSTANT) (ANOTHER CONSTANT)) NIL

See the macro loop.

sys:with-aborts-enabled (&rest identifiers) &body body  Macro
Cancels the effect of one or more invocations of sys:without-aborts.
Each of the identifiers is a symbol that relates this invocation of sys:with-aborts-enabled to a matching invocation of sys:without-aborts. The innermost sys:without-aborts with a matching identifier is nullified for the duration of body. The identifier unwind-protect identifies the automatic sys:without-aborts created by unwind-protect. It is not possible to nullify a sys:without-aborts without an identifier.
Use sys:with-aborts-enabled when an operation that is generally unsafe to abort contains an interval during which the state is consistent and aborting is safe, especially if an error can be signalled during that interval. In the case of an error, sys:with-aborts-enabled allows the user to abort without having to interact further with the Debugger.
You also use sys:with-aborts-enabled when you don’t need the automatic sys:without-aborts created by unwind-protect. For example,

  (unwind-protect (do-something)
    (sys:with-aborts-enabled (unwind-protect)
      (clean-up-something)))

If the cleanup form contained an explicit sys:without-aborts, to specify a specific reason why it should not be aborted instead of the default generic reason, the sys:with-aborts-enabled must specify the identifiers of both the explicit and the implicit sys:without-aborts. For example,
(unwind-protect (do-something)
  (sys:without-aborts
    (foo "The floor is being cleaned up.
    Aborting now could leave a serious mess that will cause
    trouble if you enter this room again later.")
    (do-something-not-abortable)
    (sys:with-aborts-enabled (foo unwind-protect)
      (do-something-abortable))))

See the function sys:without-aborts.
For a table of related items, see the section "Nonlocal Exit Functions".

clos:with-accessors slot-entries form &body body                Macro
Creates a lexical environment in which accessors can be called as if they were
variables. A reader can be called by using the variable, and a writer can be called
by using setf or setq with the variable.

slot-entries Each slot-entry is a list of the form:
  (variable-name reader-name)
  The reader-name is the name of a reader generic function, and
  variable-name is the name of a variable which will call the
  reader. Note that setf or setq may also be used with this vari-
  able, to call the corresponding writer.

form A form that evaluates to the object whose accessors should be
made available.

declarations The clos:with-accessors syntax allows declarations to appear
before the body.

body Within the lexical context of the body, the variables can be
used to call the accessors.

clos:with-added-methods                                      Special Form
Symbolics CLOS does not support clos:with-added-methods.

dbg:with-erring-frame (frame-var condition) &body body       Macro
Sets up an environment with appropriate bindings for using the rest of the func-
tions that examine the stack. It binds frame-var with the frame pointer to the
stack frame that signalled the error.

frame-var is always a pointer to an interesting stack frame.

condition is the condition object for the error, which was the first argument given
to the condition-bind handler.
(defun my-handler (condition-object)
  (dbg:with-erring-frame (frame-ptr condition-object)
    body...))

Inside body, the variable frame-var is bound to the frame pointer of the frame that got the error.

Sometimes, you might want to use the special variable dbg:*current-frame* as frame-var because some functions expect this special variable to be bound to the stack frame that signalled the error.

You would use this special variable if you are sending the :bug-report-description message to the condition object, which calls stack-examination routines that depend on the idea of a current frame, in addition to the other things that dbg:with-erring-frame sets up. :bug-report-description is the message that generates the text that the :Mail Bug Report command (c-M) puts in the mail composition window. See the generic function dbg:bug-report-description.

For a table of related items: See the section "Functions for Examining Stack Frames".

**sys:with-indentation** *(stream-var relative-indentation) &body body*  
Function

Within the body of sys:with-indentation, any output to stream-var is preceded by a number of spaces. At every recursion, the additional indentation is specified by relative-indentation. The macro does not work this way with the :item message used to display mouse-sensitive items; the items appear, but without indentation. (See the section "Interactive Streams and Mouse-Sensitive Items".)

(defun traced-factorial (n)
  (format t "Argument: ~D" n)
  (sys:with-indentation (*standard-output* 2)
    (let ((value (if (≤ n 1)
              1
              (* n (traced-factorial (1- n))))))
      (format t "Value: ~D" value)
      value))
  (traced-factorial 5)
Within the `body`, the variable `env` will be bound to an interpreter environment for the specified `instance`. The primary use of this is to create a listener loop like that of the debugger when examining a method, in which you can reference an instance’s instance variables and internal functions directly.

**clos:with-slots** `slot-entries form &body body`  
Macro

Creates a lexical environment in which slots can be accessed as if they were variables. The access to these slots is accomplished by calling `clos:slot-value`. The slots can be read (by using the variable) or written (by using `setf` or `setq` with the variable).

- **slot-entries**: Each slot-entry is one of the following:
  - `slot-name`
  - `(variable-name slot-name)`
  The slot-name is the name of a slot. If it is given alone, then it can be accessed by the variable with the same name as the slot. If it is given in the list format, then it can be accessed by the given variable-name.

- **form**: A form that evaluates to the object whose slots should be made available.

- **declarations**: The clos:with-slots syntax allows declarations to appear before the body.

- **body**: Within the lexical context of the body, the variables can be used to call clos:slot-value to access the slots.

**sys:with-table-locked** `(table) &body body`  
Function
Locks a table around body.

```
sys:without-aborts ([optional-identifier] reason &rest format-args) &body body
```

**Function**

Encloses code that should not be aborted. `sys:without-aborts` intercepts abort attempts by user action (such as `c-ABORT`), but not abort attempts by program action (such as `throw`).

When the macro is activated, it uses `reason`, a format-control string, and `format-args`, additional arguments, to display an explanation of why it is sensitive to the current abort request and what the consequences of aborting now would be. Phrase this explanation so that it is as useful and meaningful as possible to the user who is trying to abort the program. Giving the user the information needed to decide whether to leave the program running or to force it to abort is more important than conciseness. See the example given below.

`optional-identifier` is optional and usually omitted. If present, `optional-identifier` is a symbol that relates this invocation of `sys:without-aborts` to a matching invocation of `sys:with-aborts-enabled`. See the macro `sys:with-aborts-enabled`.

Use `sys:without-aborts` to protect those parts of your program, such as manipulations of global data structures, that cannot be aborted partway through their execution without damaging the program. You don't need `sys:without-aborts` if aborting the program would not cause a future execution of it to operate incorrectly.

If a program remains unsafe to abort for only a brief time, `c-ABORT` simply waits until the program leaves the `body` of `sys:without-aborts` and then aborts it. `c-ABORT` displays `reason` and queries the user only if the program remains inside `sys:without-aborts` for too long.

If a program enters the Debugger while inside `sys:without-aborts`, and you invoke a restart option that would throw through the `sys:without-aborts`, aborting the execution of `body`, the Debugger displays `reason` and queries you. In this case waiting until the program leaves `body` is not possible because the program is already stopped and sitting in the Debugger.

`sys:without-aborts` is automatically wrapped around all `unwind-protect` cleanup forms; this decreases the probability of leaving an `unwind-protect` without completely executing its cleanup forms. When `sys:without-aborts` is invoked during an `unwind-protect`, `optional-identifier` is `unwind-protect` and `reason` is a generic explanation supplied by the system.

You can specify a more precise description of why the cleanup forms of this `unwind-protect` are not safe to abort by invoking `sys:without-aborts` explicitly. You can also specify that the cleanup forms are safe to abort by invoking `sys:with-aborts-enabled` with `unwind-protect` as an identifier.

The function `process-abort`, used by the various abort keys, respects `sys:without-aborts`, waiting until the process is abortable, and asking the user what to do if the process is still not abortable after a timeout. See the section "Obsolete Process Functions".
Example:

(sys:without-aborts
   ("The ^.R widget data base is being ~(\^A\^-)d.\^E
    Aborting this could leave the data base in an inconsistent state,\^E
    and future operations on widgets might fail in unpredictable ways."
    2 :update)
(+ 1 'foo))

Trap: The second argument...

s-A, <RESUME>: Supply replacement argument
s-B: Return a value from the +-INTERNAL instruction
s-C: Retry the +-INTERNAL instruction
s-D, <ABORT>: Return to Dynamic Lisp Top Level in Dynamic Lisp Listener 2
s-E: Restart process Dynamic Lisp Listener 2

-->Abort Abort

Return to Dynamic Lisp Top Level in Dynamic Lisp Listener 2

The program cannot safely be aborted at this time.
   The second widget data base is being updated.
   Aborting this could leave the data base in an inconsistent state,
   and future operations on widgets might fail in unpredictable ways.
Do you want to Skip or Abort? (press <HELP> for help) <HELP>

The current program operation is one that the programmer expected
to run to completion. Aborting this operation partway through
could leave the program in an inconsistent state and interfere
with its proper operation.

Your choices are:

   Skip Abandons this attempt to abort the program.
   Abort Aborts the program by force, accepting the risk of damage.

Do you want to Skip or Abort? Abort

Back to Dynamic Lisp Top Level in Dynamic Lisp Listener 2.

The example assumes the user of this program knows what widgets are and what
a widget data base is. If this is not the case, the reason string should include a
brief explanation.

In this example, the Debugger offers you two choices. If you select Skip, you can
use one of the first two proceed options to correct the error in the program and
continue execution. If you select Abort, you accept the possibility that the program
won't work correctly in the future.

If the program had been aborted with c-ABORT, you would have been offered addi-
tional choices, as follows:

Skip Abandons this attempt to abort the process.
Wait waits until the process reaches a point where it can safely be aborted. Offers these choices again if five seconds elapse and it still cannot be aborted.

Wait indefinitely keeps waiting for as long as it takes. Another attempt to abort stops waiting and offers these choices again.

Abort aborts the process by force, accepting the risk of damage.

Debug enters the Debugger for detailed investigation.

For a table of related items, see the section "Nonlocal Exit Functions".

**without-floating-underflow-traps &body body**  
*Special Form*

Inhibits trapping of floating-point exponent underflow traps within the body of the form. The result of a computation which would otherwise underflow is a denormalized number or zero, whichever is closest to the mathematical result.

Example:

```lisp
(describe (without-floating-underflow-traps (expt .1 40))) =>
1.0e-40 is a single-precision floating-point number.
   Sign 0, exponent 0, 23-bit fraction 213302 (denormalized)
1.0e-40
```

*Function*

The printed representation of `object` is written to the output stream specified by `:stream`, which defaults to the value of `*standard-output*`, or `*terminal-io*` if `:stream` is `t`.


**write** returns `object`. For example:

```lisp
(writeln "A simple string") => "A simple string"
"A simple string"
```

(let ((*print-escape* t) (s "foo"))
  (terpri)
  (write s)
  (write-char \\Space)
  (prin1 s)
  (write-char \\Space)
  (princ s)
  nil)
"foo" "foo" foo
=> NIL

(let ((*print-escape* nil) (s "foo"))
  (terpri)
  (write s)
  (write-char \\Space)
  (prin1 s)
  (write-char \\Space)
  (princ s)
  nil)
foo "foo" foo
=> NIL

**write-byte**

```
write-byte integer binary-output-stream
```

Writes one byte, the value of `integer` to `binary-output-stream`. It is an error if `integer` is not of the type specified as the `:element-type` argument to `open` when the stream was created. **write-byte** returns `integer`.

```lisp
(with-open-file (s "data.file"
    :direction :output
    :element-type '(unsigned-byte 2))
  (write-byte 1 s)
  (write-byte 3 s)
  (write-byte 2 s))
=> 2
```

```lisp
(with-open-file (s "data.file"
    :direction :input
    :element-type '(unsigned-byte 2))
  (list (read-byte s) (read-byte s) (read-byte s)))
=> (1 3 2)
```

**write-char**

```
write-char character &optional output-stream
```

Outputs `character` as a printing character to `output-stream`, and returns `character` as a character object. `character` must be a character object. For example:
(write-char \a) => a
\a

output-stream, which, if unspecified or nil, defaults to *standard-input*, and if t, is *terminal-io*.

(with-output-to-string (s)
  (princ "foo" s)
  (write-char \Space s)
  (princ "bar" s))
=> "foo bar"

**write-line** string &optional output-stream &key (start 0) end  
    
Function

Writes the characters of the specified substring of string to output-stream, followed by a newline. The :start and :end parameters delimit a substring of string. **write-line** returns string. For example:

  (write-line "hello") => hello
  "hello"

(setq stream (make-string-output-stream))
=> #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 35643762>

(write-line "two words" stream :start 4)
=> "two words" ;returns the full string

(get-output-stream-string stream)
=> "words"
  ;writes the substring plus NEWLINE to the stream

output-stream, which, if unspecified or nil, defaults to *standard-input*, and if t, is *terminal-io*.

(with-output-to-string (s)
  (write-line "foo" s)
  (write-line "bar" s)
  (write-line "baz" s))
=> "foo
  bar
  baz"

**write-string** string &optional output-stream &key (:start 0) :end  
    
Function

Writes the characters of the specified substring of string to output-stream, without a following newline. The :start and :end parameters delimit a substring of string. **write-string** returns string. For example:
(write-string "hello") => hello"hello"

(setq s (make-string-output-stream))
=> #<LEXICAL-CLOSURE CLI::STRING-OUTPUT-STREAM 14372772>

(write-string "two words" s :start 4)
=> "two words" ;returns the full string

(get-output-stream-string s)
=> "words" ;writes the substring to the stream output-stream, which, if unspecified or nil, defaults to *standard-input*, and if t, is *terminal-io*.

(with-output-to-string (s)
  (write-string "foo" s)
  (write-char #\Space s)
  (write-string "bar" s))
=> "foo bar"


Function

The object is printed as if by write, and the characters that would be output are made into a string, which is returned. The other keyword arguments specify values used to control the generation of the printed representation. See the function write and see CLtL 384.

For example:

  (write-to-string '|red|) => "|red|"

(let ((*print-escape* t))
  (list (write-to-string #\A)
     (progn (setq *print-escape* nil) (write-to-string #\A))))
=> (#\A "A")

xcons y x

Function

Creates an "exchanged cons", one whose car is x and whose cdr is y. Example:

  (xcons 'a 'b) => (b . a)

xcons is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Constructing Lists and Conses".
**xcons-in-area** \( x \ x \ area \)

Function

Creates an "exchanged cons", one whose car is \( x \) and whose cdr is \( y \), in the specified \( area \). (Areas are an advanced feature of storage management. See the section "Areas").

**xcons-in-area** is a Symbolics extension to Common Lisp.

For a table of related items: See the section "Functions for Constructing Lists and Conses".

**zerop** \( number \)

Function

Returns \( t \) if \( number \) is zero, otherwise \( nil \). If \( number \) is not a number, **zerop** signals an error.

For floating-point numbers, this only returns \( t \) for exactly \( 0.0 \), \( -0.0 \), \( 0.0d0 \) or \( -0.0d0 \); there is no "fuzz". For complex numbers, both real and imaginary parts must be zero.

\[
\begin{align*}
(\text{zerop} \ 0.0) & \Rightarrow t \\
(\text{zerop} \ #c(0 \ 0)) & \Rightarrow t
\end{align*}
\]

For a table of related items, see the section "Numeric Property-checking Predicates".