Palo Alto Research Centers

Report on the Programming Language Euclid

Butler W. Lampson
James J. Horning
Ralph L. London
James G. Mitchell
Gerald J. Popek

XEROX

Report on the Programming Language Euclid

By Butler W. Lampson, James J. Horning, Ralph L. London,
James G. Mitchell, and Gerald J. Popek

October 1981

CSL-81-12

This report describes a new programming language called Euclid, intended for the expression of system programs which are to be verified.

XEROX
PALO ALTO RESEARCH CENTERS
3333 Coyote Hill Road / Palo Alto / California 94304

Table of contents

Preface

This report describes a programming language called Euclid, intended for the expression of system programs which are to be verified. Euclid draws heavily on Pascal for its structure and many of its features. In order to reflect this relationship as clearly as possible, the Euclid report has been written as a heavily edited version of the revised Pascal report.

Proof rules for Euclid appear in a separate report [London et al. 1978]. An informal discussion of the language design appears in [Popek et. al. 1977]. Euclid has been implemented (with some omissions) by the Computer Systems Research Group, University of Toronto, Toronto, Canada, and I.P. Sharp Associates, Toronto, Canada [Holt et. al. 1978, Holt and Wortman 1979, Holt et. al. 1980]. The translator is a 70,000 line Euclid program, the largest such program now in existence [Wortman and Cordy 1981].

This is the fourth version of the Euclid report; earlier versions appeared in May 1976, August 1976, and February 1977 (the latter as *SIGPLAN Notices* 12, 2, Feb. 1977).

Acknowledgements

Obviously, we are greatly indebted to Wirth, both for the aspects of the language that are copied from Pascal, and for the structure and some of the wording of the report. We are also much in debt to Hoare's work in the areas of programming language design, axiomatic methods, and program verification. In particular, we have tried to follow his suggestion that "the language designer should be familiar with many alternative features designed by others, and should have excellent judgement in choosing the best. . . One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation." [Hoare 1973]. Experience has shown that we have not been entirely successful.

We have consciously borrowed ideas and features from Alphard [Wulf, London, and Shaw 1976], BCPL [Richards 1969], CLU [Liskov 1977], Gypsy [Ambler et al. 1977], LIS [Ichbiah et al. 1974], Mesa [Geschke and Mitchell 1975], Modula [Wirth 1977], and the SUE System Language [Clark and Horning 1973, Clark and Ham 1974]; other languages and suggestions for language features have also undoubtedly influenced our thinking. We have benefitted greatly from comments and criticisms by numerous individual colleagues on previous versions of the language and report, from the comments of two participants in an early attempt to implement Euclid (Hugh Lauer and Dave Thompson) and from the Toronto implementation team.

We are grateful to Prof. Wirth, and to Springer-Verlag, for permission to use portions of the Pascal report [Wirth 1971, Jensen and Wirth 1975] in this report.

Our work has been significantly aided by the Arpanet, which allowed us to maintain effective and rapid communication in stating and resolving problems, in spite of the wide geographical distribution of the authors.

1. Introduction

"There is no royal road to geometry."
Proclus, *Comment on Euclid*, Prol. G. 20.

The programming language Euclid has been designed to facilitate the construction of verifiable system programs. By a verifiable program we mean one written in such a way that existing formal techniques for proving certain properties of programs can be readily applied; the proofs might be either manual or automatic, and we believe that similar considerations apply in both cases. By system we mean that the programs of interest are part of the basic software of the machine on which they run; such a program might be an operating system kernel, the core of a data base management system, or a compiler.

An important consequence of this goal is that Euclid is not intended to be a general-purpose programming language. Furthermore, its design does not specifically address the problems of constructing very large programs; we believe most of the programs written in Euclid will be modest in size. While there is some experience suggesting that verifiability supports other desired goals, we assume the user is willing, if necessary, to obtain verifiability by giving up some run-time efficiency, and by tolerating some inconvenience in the writing of his programs.

We see Euclid as a (perhaps somewhat eccentric) advance along one of the main lines of current programming language development: transferring more and more of the work of producing a correct program, and verifying its correctness, from the programmer and the verifier (human or mechanical) to the language and its compiler.

The main changes relative to Pascal take the form of restrictions, which allow stronger statements about the properties of the program to be made from the rather superficial, but quite reliable, analysis that the compiler can perform. In some cases new constructions have been introduced, whose meaning can be explained by expanding them in terms of existing Pascal constructions. The reason for this is that the expansion would be forbidden by the newly introduced restrictions, whereas the new construction is itself sufficiently restrictive in a different way. Major extensions to Pascal are prototypes, machine dependent features, counted storage, and modules.

The main differences between Euclid and Pascal are summarized in the following list:

Visibility: Euclid provides explicit control over the visibility of identifiers, by requiring the program to list all the identifiers imported into a routine or module, or exported from a module.

Variables: The language guarantees that two identifiers in the same scope can never refer to the same or overlapping variables. There is a uniform mechanism for binding an identifier to a variable in a procedure call, on block entry (replacing the Pascal with statement), or in a variant record discrimination. The variables referenced or modified by a routine (i.e., procedure or function) must be known in every scope from which the routine is called.

Pointers: This idea is extended to pointers, by allowing dynamic variables to be assigned to collections, and guaranteeing that two pointers into different collections can never refer to the same variable.

Storage allocation: The program can control the allocation of storage for dynamic variables explicitly, in a way that very narrowly confines the opportunity for making a type error. It is also possible to declare that some dynamic variables should be reference-counted, and automatically deallocated when no pointers to them remain.

Types: Types have been generalized to prototypes, which have formal parameters, so that arrays can have bounds that are fixed only when they are created, and variant records can be handled in a type-safe manner. Records are generalized to include constant components.

Modules: A new kind of record, called a module, can contain routine and type components, and thus provides a facility for modularization. The module can include initialization and

finalization statements that are executed whenever a module variable is created or destroyed.

Constants: Euclid defines a constant to be a literal, an identifier whose value is fixed throughout the scope in which it is declared, or an expression whose operands are constant.

For statement: A generator can be declared as a module type, and used in a for statement to enumerate a sequence of values.

Loopholes: features of the underlying machine can be accessed, and the type-checking can be overridden, in a controlled way. Except for the explicit loopholes, Euclid is designed to be type-safe.

Assertions: the syntax allows assertions to be supplied at convenient points.

Deletions: A number of Pascal features have been omitted from Euclid: input-output, reals, multi-dimensional arrays, labels and gotos, and functions and procedures as parameters.

The only new features in the list that can make it hard to convert a Euclid program into a legal Pascal program by straightforward rewriting are prototypes, storage allocation, finalization, and some of the loopholes.

There are a number of other considerations that influenced the design of Euclid:

It was supposed to be based on current knowledge of programming languages and compilers; concepts that were not fairly well understood, and features whose implementation was unclear, were omitted. Implementation experience has shown that certain features escaped this constraint: modules, type equivalence, legality assertions.

Although program portability was not a major goal of the language design, it is necessary to have compilers that generate code for a number of different machines, including mini-computers.

The object code must be reasonably efficient, and the language must not require a highly optimizing compiler to achieve an acceptable level of efficiency in the object program. The Toronto compiler in fact achieves object program efficiency comparable to that of the C compiler for straightforward programs.

Since the total size of each program was to be modest, separate compilation was not required (although it was certainly not ruled out).

The required run-time support must be minimal, since it presents a serious problem for verification.

2. Summary of the language

"Be sure of it; give me the ocular proof."
Othello III, iii, 361.

This section contains a summary of Euclid. The information here is intended to be consistent with the remainder of the report, but in case of conflict the body of the report (sections 3-12) governs. Many details are omitted here, and some general statements are made without the qualifications contained in the body of the report.

An algorithm or computer program consists of two essential parts, a description of *actions* that are to be performed, and a description of the *data* that are manipulated by these actions. Actions are described by *statements*, and data are described by *type definitions*. A data type defines both a set of values and the actions that may be performed on elements of that set.

The data are represented by *values*. A value may be *constant*, or it may be the value of a *variable*. A value occurring in a statement may be represented by a *literal constant*, an identifier which has been declared to be constant, an identifier which has been declared as a variable, or an *expression* containing values. Every identifier occurring in the program must be introduced by a *declaration*. A constant or variable declaration associates with an identifier a data type, and either a value or a variable.

In general, a *definition* specifies a fixed value, type, or routine, and a *declaration* introduces an identifier and associates some properties with it. A data type may either be directly described in the constant or variable declaration, or it may be referenced by a type identifier, in which case this identifier must be introduced by an explicit *type declaration*.

A *constant declaration* associates an identifier with a value; the association cannot be changed within the scope of the declaration. If the value can be determined at compile-time, the constant is said to be *manifest*; the expression defining a manifest constant must contain only literal constants, other manifest constants, and certain built-in operations.

An *enumerated* type definition indicates an ordered set of values, i.e., introduces identifiers standing for each value in the set. The *simple* data types are the enumerated types, the subrange types, and the five *standard simple types*: *Boolean*, *Integer*, *Char*, *StorageUnit* and *AddressType*. For the first three, there is a way of writing literal constants of that type: true and false for Boolean, numbers for Integers, and quotations for Chars. Numbers and quotations are syntactically distinct from identifiers. The set of values of type Char is the character set available in a particular implementation. The type StorageUnit has values which occupy the minimum unit in which storage allocation is done; this may of course differ from one implementation to another. Since no operations are defined on StorageUnit values, nothing more need be said about them. The type AddressType has values that are machine addresses.

A type may also be defined as a *subrange* of a simple type by indicating the smallest and the largest value of the subrange.

Structured types are defined by describing the types of their components, and indicating a *structuring method*. The various structuring methods differ in the selection mechanism serving to select the components of a variable of the structured type. In Euclid, there are five basic structuring methods available: array, record, module, set, and collection.

In an *array structure*, all components are of the same type. A component is selected by an array selector, or *computable index*. The index type, which must be simple, is indicated in the array type declaration. It is usually a programmer-defined enumerated type, or a subrange of the type Integer. Given a value of the index type, an array selector yields a variable or constant of the component type. Every array structure can therefore be regarded as a mapping of the index type into the component type.

In a *record structure*, the components (called *fields*) are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing the program), a record selector is not a computable value, but instead is an identifier uniquely denoting the component to be selected. These field identifiers are declared in the record type definition. Records may include constant as well as variable components; manifest constant components, of course, do not need to be stored in each record value.

A record type may be specified as consisting of several *variants*. This implies that different record values, although declared to be of the same type, may assume structures which differ in a certain manner. The difference may consist of a different number and different types of components. The variant which is assumed by a record value is indicated by a constant of some simple type, called the *tag*.

A *module structure* is much like a record, but may include routines and types as components. In this way, the operations that are defined on a data structure can be conveniently packaged with the structure. Module components cannot be accessed outside the module body unless they are explicitly exported. Thus, in a properly written program, it is evident from the lexical structure how the state of a module can be altered.

A *set structure* defines the set of values that is the powerset of its base type, i.e., the set of all subsets of values of the base type. The base type must be a simple type.

Variables declared in explicit declarations are called *static*. The declaration associates an identifier with the variable, and the identifier is used to refer to the variable. The language guarantees that two identifiers that are known in the same scope cannot refer to the same variable, or to overlapping variables. Thus, an assignment to an identifier cannot change the value of any other identifier known in the same scope.

In contrast, variables may also be generated by executable statements. Such a *dynamic* generation yields a *pointer* value (a substitute for an explicit identifier) that subsequently serves to refer to the variable. This pointer may be assigned to other variables, namely variables of type pointer. Each pointer variable may assume values pointing to variables in a single *collection C*, all of whose members are of the same type, or may assume the value *C*.nil, which points to no variable. Because pointer variables may also occur as components of structured variables which are themselves dynamically generated, the use of pointers permits the representation of finite graphs in full generality. Although the language cannot guarantee in general that two pointer variables do not refer to the same variable, it can make this guarantee for two pointers in different collections.

A *zone* can be associated with each collection to provide procedures for allocating and deallocating the storage required by variables in that collection; if the zone is omitted, a standard system zone is used. The program may free a dynamic variable explicitly, in which case the program is responsible for ensuring that no other pointers to the freed variable remain. Alternatively, the collection may be *reference-counted*, in which case each variable is automatically freed when no pointers to it remain. The main advantage of reference-counted variables, as compared with explicit deallocation, is that the correctness of the deallocation does not have to be verified.

Throughout this report, the word *variable* means a container that can hold a value of a specific type. A variable may or may not be associated with an identifier. A *constant*, by contrast, is simply a value of a specific type. The fundamental difference is that assignment to a variable is possible.

A *prototype* may be declared by a type declaration with formal parameters; such a prototype represents a set of types, one of which is specified each time the prototype is referenced and actual parameters are supplied for the formals.

Two types are the same if their definitions are identical after any type identifiers that are not *opaque* have been replaced by their definitions, and any actual parameters and any constant identifiers declared outside the type have been replaced by their values. A type identifier is opaque if it is a module type, or is exported from a module.

The most fundamental statement is the *assignment statement.* It specifies that a newly computed value be assigned to a variable (or a component of a variable). The value is obtained by evaluating an *expression.* Expressions consist of variables, constants, operators and functions operating on the denoted quantities and producing new values. Variables, constants, and functions either are declared in the program or are standard entities. Euclid defines a fixed set of *operators,* each of which can be regarded as describing a mapping from the operand types into the result type. The set of operators is subdivided into groups of:

1. *arithmetic operators* of addition, subtraction, sign inversion, multiplication, division, and computing the remainder **(mod).**
2. *Boolean operators* of negation **(not),** conjunction **(and),** disjunction **(or)** and implication **(−>).**
3. *set operators* of union, intersection, set difference, and symmetric difference **(xor).**
4. *relational operators* of equality, inequality, ordering, set membership and set inclusion. The results of relational operations are of type Boolean.

The *procedure statement* causes the execution of the designated procedure (see below).

There are two kinds of *escape* statements: an *exit statement* is used to terminate a loop, and a *return statement* to terminate a routine. An escape statement qualified by a *when clause* causes termination only if a Boolean expression is true.

Assignment, procedure, and escape statements are the components or building blocks of *structured statements,* which specify sequential, selective, or repeated execution of their components. Sequential execution of statements is specified by the *compound statement;* conditional or selective execution by the *if statement* and the *case statement;* and repeated execution by the *loop statement* and the *for statement.*

The if statement serves to make the execution of a statement dependent on the value of a Boolean expression, and the case statement allows for the selection among many statements according to the value of a selector. The discriminating case statement provides a safe way of discriminating the current variant of a variant record. The for statement is used when a bound on the number of iterations is known beforehand, and the loop statement is used otherwise.

A *block* can be used to associate declarations with statements. The identifiers thus declared have significance only within the block. Hence, the block is called the *scope* of these identifiers, and they are said to be *local* to the block. Since a block may appear as a statement, scopes may be nested. An if, case, for or loop statement, or a module type declaration, also defines a scope in a similar way.

A block can be named by an identifier and be referenced through that identifier. The block is then called a *procedure,* and its declaration a *procedure declaration.* However, an identifier that is not local to a given procedure body is accessible in that body only if it is accessible in the immediately enclosing scope, and

it is *pervasive* in some enclosing scope or

it is explicitly *imported* into the given procedure body.

A procedure has a fixed number of parameters, each of which is denoted within the procedure by an identifier called the *formal parameter,* which is local to the procedure body. Upon an activation of the procedure, an actual quantity has to be indicated for each parameter which can be referenced from within the procedure through the formal parameter. This quantity is called the *actual parameter.* There are two kinds of parameters: constant parameters and variable parameters; routine and type parameters are not allowed. In the first case, the actual parameter is an expression that is evaluated once. The formal parameter represents a local constant whose value is the result of this evaluation. In the case of a variable parameter, the actual parameter is a variable and the formal parameter is *bound* to this variable. Any indices or pointers in the actual parameter expression are evaluated before execution of the procedure.

Functions are declared analogously to procedures; procedures and functions are collectively called *routines.* The main difference lies in the fact that a function yields a result, which may be of any assignable type and must be specified in the function declaration. Functions may therefore be used as constituents of expressions. Variable formal parameters and imported variables are not permitted within function declarations; as a consequence, functions cannot have side effects.

Since Euclid is intended for the writing of programs that are to be verified (either mechanically or by hand), there are a number of explicit interactions between the language and the verifier, in addition to the many aspects of the language that have been motivated by the desire to ease verification. These explicit interactions fall into two main categories:

embedding of *assertions* in the program: the special symbols **assert**, **invariant**, **pre** and **post** introduce assertions. These may be written as comments that are ignored by the compiler. Presumably they will be used by the verifier, which can take advantage of their relationship to the structure of the program. Alternatively, an assertion may be written as a Boolean expression, which is compiled into a run-time check if the **checked** option is enabled.

compiler-generated assertions: in cases where the compiler needs to be able to assume that some condition holds, but is unable to deduce that it does, the compiler may generate a *legality assertion* (in a new listing of the program) for the verifier, and then proceed as though confident of its truth. The legality of the program will then depend on the validity of the compiler-generated assertion. Each case in which such an assertion may be generated is spelled out in this report.

3. Notation, terminology, and vocabulary

"The best words in the best order."
Coleridge.

The syntax is described in a modification of Backus-Naur form, in which syntactic constructs are denoted by English words or phrases, not enclosed in any special marks. These words also suggest the nature or meaning of the construct, and are used in the accompanying description of semantics. Basic symbols of the language are written in boldface or enclosed in quote marks; e.g., **begin** and ";". Possible repetition of a construct is indicated by enclosing the construct within metabrackets { and } . Possible omission of a construct is indicated by enclosing the construct within metabrackets [and] . The word "empty" denotes the null sequence of symbols.

The grammar defining the syntax of Euclid is distributed throughout this report; for convenient reference, it has also been collected in Appendix A.

3.1 Vocabulary

The primitive vocabulary of Euclid consists of basic symbols classified into letters, digits, and special symbols. Note that this vocabulary is *not* the character set. The character set is implementation dependent, and each implementation must define, in its character set, distinct representations for all the basic symbols. Suggestions for doing this in some common cases may be found in section 13.

Each implementation must specify a single *break character* that can be used within an identifier. Two identifiers are *similar* if they are composed of the same sequence of characters, except for changes from upper to lower case letters or vice versa, and for the presence or absence of break characters. The intended use of the break character is to visually separate an identifier into its component parts. In an implementation that can print upper and lower case letters, a transition from lower to upper case can also be used for this separation. It is recommended that this convention be used when possible, in preference to the explicit break character, for implementations that have lower case letters; obviously it cannot be used if the entire identifier is upper case. Thus, an identifier might be represented as

alphaBeta	using 96-character ASCII, with capitalization as the break.
ALPHA BETA	sing the IBM PL/I character set, with as the break.
ALPHA\BETA	using the Model 33 Teletype character set, with \ as the break.

All of these identifiers would be similar to the identifier ALPHABETA. With these conventions, it is possible to convert from one representation to another in a reasonable way (see 13.).

Each time an identifier is used, it must be written in exactly the same way (i.e., with the same capitalization and use of break characters) as it was written when it was declared. However, another identifier that is similar according to the above rules may not be declared in any scope in which the first identifier is known (see 7.3).

The following capitalization convention is used in this report: type, routine and module identifiers begin with a capital letter; other identifiers begin with a small letter. This convention is not part of the definition of Euclid, however.

```
letter      ::= "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"
octalDigit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
digit       ::= octalDigit | "8" | "9"
hexDigit    ::= digit | "A" | "B" | "C" | "D" | "E" | "F"
breakChar   ::= <some implementation-dependent character not a letter or digit>
```

specialSymbol ::= "+" | "-" | "*" | "=" | "<" | ">" | "<=" | ">=" | "->" | "(" | ")" | ".." | "{" | "}" | ":=" | "." | "," | ";" | ":" | "'" | "↑" | "=>" | "$" | "#" | wordSymbol

wordSymbol ::= **abstraction | aligned | and | any | array | assert | at | begin | bind | bits | bound | case | checkable | checked | code | collection | const | converter | counted | decreasing | default | dependent | div | else | elseif | end | exit | exports | finally | for | forward | from | function | if | imports | in | include | initially | inline | invariant | loop | machine | mod | module | not | of | or | otherwise | packed | parameter | pervasive | post | pre | procedure | readonly | record | return | returns | set | then | thus | to | type | unknown | var | when | with | xor**

The construct

"{" <any sequence of symbols not containing "}"> "}"

may be inserted between any two identifiers, literal constants (see 4.), or special symbols. It is called a *comment*, and may be removed from the program text without altering its meaning. The symbols "{" and "}" do not occur otherwise in the language.

Any verification system that accepts Euclid programs as input may define a convention for distinguishing comments that have special meaning for the verifier. One reasonable convention is that every comment in which a certain character appears as the first character after the { is intended for the verifier.

The word *routine* is used as a synonym for the phrase *procedure or function.* The word *note* is used to introduce material that adds no new information, but is intended simply to help the reader understand the implications of rules that have already been stated. The definition of Euclid is not affected by any such notes (although its comprehensibility may be). The words *thus* and *hence* play a similar role.

3.2 Legality assertions

Throughout this report, various restrictions are placed on *legal* Euclid programs. Many of these restrictions cannot be checked syntactically, and in some cases they involve dynamic conditions that are difficult (or impossible) to check statically. Nevertheless, programs that violate them are not considered to be meaningful Euclid programs. It is the responsibility of the compiler to verify as many of these properties as it can, and to produce Boolean expressions called *legality assertions* for those it cannot. Thus, a program is a legal Euclid program, with well-defined semantics, only if it can be verified that no execution leads to falsification of any legality assertion. Note that legality assertions are produced only for conditions specified in this report.

If checking is enabled for a scope (i.e., **checked** is specified and not overridden by an inner **not checked**; see 6.2.3 and 9.2.1), all legality assertions in the block, and all programmer-supplied assertions that are Boolean expressions, are compiled into run-time checks, as an aid in detecting illegal programs before the verification process is complete. See 6.2.6 for the use of a collection's Free procedure in a checked scope.

3.3 Lexical structure

The text of a program is built up out of declarations and statements, collectively called *units*, according to the syntax specified below. The syntax is constructed in such a way that a unit is *always* followed by a semicolon, but in all cases the semicolon is optional. The recommended style is to omit a semicolon at the end of a line.

Commas are used as separators in enumerated types, case label lists, element lists, and parameter lists, and within declarations in identifier lists, bind lists, and import/export lists.

There are several kinds of brackets that are used to group declarations and statements for various purposes. The following list gives the unique closing bracket for each opening bracket.

if ...	**end if**
loop ...	**end loop**
case ...	**end case**
=> ...	**end** caseLabel
begin ...	**end**, or **end** routineIdentifier whenever the block is the body of a routine declaration
code ...	**end** routineIdentifier
record ...	**end record**, or **end** typeIdentifier whenever the record definition is the declaration of a type identifier
module ...	**end module**, or **end** typeIdentifier whenever the module definition is the declaration of a type identifier

4. Identifiers, numbers and strings

"And twenty more such names as these
Which never were nor no man ever saw."
The Taming of the Shrew, Induction, ii, 95.

Identifiers serve to denote constants, variables, types, prototypes, and routines. Their association must be unique within their scope of validity, i.e., within the scope in which they are declared (see 6., 7.3).

```
identifier        ::= letter { letterOrDigit }
letterOrDigit     ::= letter | digit | breakChar
```

Throughout the report, non-terminals of the form *x*Identifier are syntactically equivalent to identifier; the qualifying *x* reflects the intended class of associations.

The usual decimal notation is used for numbers, which are the literal constants of the data type Integer (see 6.1.2.). Numbers may also be written in octal or hexadecimal notation. Note that unsigned numbers are always non-negative; a negative manifest constant can be written as an expression, e.g., -14 or (-32767-1) or SignedInt.first.

```
unsignedNumber    ::= digit { digit } |
                      octalDigit { octalDigit } "#8" |
                      digit { hexDigit } "#16"
```

Examples:

```
1    100    717#8    0CAD1#16    123#16
```

Sequences of characters enclosed by quote marks are called *literal string constants*. They are the literal constants of the standard type String (see 6.2.1). A character code, whether or not it is in the printing character set, can also be represented in a literal string constant as follows:

> $*ddd*, where each *d* stands for a decimal digit, represents the character code with the decimal representation *ddd*. Note: Char.Ord($$*ddd*) = *ddd* (see 6.1.2).

For convenience, $S, $T, $N, $$, $' represent space, tab, newline, $, and ' respectively. The characters $, ', tab and newline may only be represented by an extension. The $*ddd* construction can be used only in a machine-dependent module (see 6.2.3).

```
literalString     ::= "'" { extendedCharacter } "'"
extendedCharacter ::= character | "$" extension
extension         ::= digit digit digit | "S" | "T" | "N" | "$" | "'"
```

Examples:

```
' '    'A'    ';'    ' $' '    'Here comes a null: $000, and there it went'
'Euclid'    'THIS IS A STRING'    'This$Sis$Sa$Sstring'
```

A single character preceded by a dollar sign is a literal constant of the standard type Char (see 6.1.2). The $ convention may also be used in these constants.

```
literalChar       ::= "$" extendedCharacter
```

Examples:

```
$a    $$S {space character}    $$000 {the NUL character}    $"    $$$
```

5. Manifest constants

"One here will constant be,
Come wind, come weather."
Pilgrim's Progress

A manifest constant is

a literal constant;

an identifier declared as a constant with a defining expression that is a manifest constant;

a field designator whose field identifier is a manifest constant (see 7.2.2), or one of the designators listed in the next paragraph;

an expression whose operands are all manifest constants, and which does not involve any functions, except the standard ones defined in this report.

If *T* is an enumerated type, or a subrange type with manifest bounds, then *T*.first and *T*.last are manifest. A type or prototype is manifest if every type, prototype and expression in its definition is manifest. For any manifest type *T* except Integer, *T*.size and *T*.alignment are manifest.

Thus, the value of a manifest constant can be computed in a straightforward way at compile time. A general constant, by contrast, has a value that is fixed during the lifetime of the scope in which it is defined, but may be computed in an arbitrary way at the beginning of that lifetime. Note that a constant formal parameter is *not* a manifest constant.

```
literalConstant      ::= unsignedNumber | literalString | literalChar | enumeratedValueIdentifier
manifestConstant     ::= literalConstant | manifestExpression
manifestExpression   ::= expression
```

Examples:

```
-100      $a      'Euclid'      red      3*Color.Ord(Color.last)
```

6. Types

"What is written without effort is in general read without pleasure."
Johnson

A data *type* determines the set of values which variables and constants of that type may assume, and the set of basic operations that may be performed on them. A type declaration associates an identifier with the type. The type identifier denotes the same type as its definition, unless it is declared as a module type (see 6.2.3) or is exported from a module, in which case the identifier denotes a different type; type equivalence is discussed in detail in 6.4. Prototypes are introduced in 6.3.

An identifier must be *declared* before it is used. When there are mutually recursive routines, types or prototypes, however, it is impossible to give the *definition* of every identifier before its use. In this situation, a definition of **forward** may be given instead, and later in the same closed scope another declaration, of the form **type** *T*=... (or **procedure** *P*=..., or **function** *F*=...) must appear to provide the true definition. If an identifier declared with **forward** has parameters, these must appear in the forward definition and must *not* be repeated in the true definition. Between its forward and true definitions, or within its own definition, a type may only be used as the object type of a collection.

A type definition may not contain calls to functions which import variables, or variable identifiers that are free, except for variables appearing

in an import list,

as the collection variable of a pointer type,

as the zone of a collection type, or

within a nested, closed scope.

Hence, a type identifier denotes the same type throughout the scope in which it is declared, a type component in a module denotes the same type in every value of the module type, and the type denoted by a prototype application depends only on the values of the actual parameters. Note, however, that the same type identifier may denote different types in different scopes, e.g., in different instantiations of a routine.

All types (except Integer) automatically acquire components when they are declared. For example, an array type *T* has the component *T*.IndexType (see 6.2.1). Any component of a type is automatically also a component of every constant or variable of that type. Thus if *v* is a constant or variable of type *T*, *v*.IndexType is the same as *T*.IndexType.

A type *T* is *assignable* unless it is a collection type (see 6.2.6), an opaque type for which assignment is not exported (see 6.2.3), or a structured type with a variable component whose type is not assignable (see 6.2). If *T* is not assignable, variables of type *T* may not appear on the left side of an assignment statement, and constants and constant parameters of type *T* may not be declared.

This report specifies the *standard representations*, in terms of bits, for the values of certain types. These specifications are given so that machine-dependent records and machine-code procedures can be sensibly defined, and so that the effect of an explicit type conversion can be predicted. The following contexts are defined as *sensitive*:

a variable component of a machine-dependent record (see 6.2.4)

a variable declared at a fixed address (see 7.)

an actual parameter or result of a machine-code routine (see 10.)

the actual parameter or result of an explicit type converter (see 6.5)

A type may not appear in a sensitive context unless its representation is specified, either in this report, or explicitly by the implementation. Except in these sensitive contexts, there is no way for a Euclid program to determine the representation of any value, and an implementation is therefore free to use any representation, provided that it converts each value to the standard representation when it appears in a sensitive context.

type ::= simpleType | structuredType
typeDeclaration ::= **type** typeIdentifier = preAssertion typeDefinition | prototypeDeclaration
typeDefinition ::= type | **forward**

There are two components implicitly declared for each type *T* other than Integer:

T.size — the result, of type Integer, is the number of StorageUnits (see 6.1.2) required for the representation of a variable of type *T*.

T.alignment — the result, of type Integer, is the required alignment of variables of type *T*, in StorageUnits. Thus, if *p* : *Ptype* is a pointer to such a variable, and *PtypeToAddressType* is a converter from *Ptype* to AddressType, then *PtypeToAddressType*(*p*) **mod** *Ptype*.alignment = 0.

There is a component implicitly declared for each variable or constant:

x.ItsType — the type of *x*

6.1. Simple types

There are no type variables in Euclid. However, a type may be a component of a module (see 6.2.3), and hence may be referenced by a field designator (see 7.2.2), as well as by an identifier.

simpleType ::= enumeratedType | standardSimpleType | subrangeType | derivedSimpleType | simpleTypeAppl
derivedSimpleType ::= [containingVariable "."] simpleTypeIdentifier

6.1.1. Enumerated types

An enumerated type defines an ordered set of values by enumeration of the identifiers which denote these values. There must be at least two such identifiers. The identifiers are declared as constants in the current scope. If the current scope is a type declaration of type *T*, for which the enumerated type is the definition, however, the identifiers are declared in the enclosing scope instead, i.e., the scope in which *T* is declared. As with all other declarations, the identifiers may not be used for any other purpose in the scope in which they are declared.

The standard representation of the *i*th identifier (counting from 0) in the enumeration is the same as the representation of the unsigned Integer *i*. Thus, if *T* is an enumerated type, *T*.first is represented exactly like the Integer 0.

enumeratedType ::= "(" enumeratedValueIdentifier { "," enumeratedValueIdentifier } ")"

Examples:

type *Color* = (*red, green, blue, orange, yellow, purple*)
type *Suit* = (*club, diamond, heart, spade*)
type *Day* = (*Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday*)
type *SexType* = (*female, male*)
type *Classification* = (*confidential, secret, topSecret*)
type *Device* = (*disk, display, keyboard, printer, tape*)

Components implicitly declared for each enumerated and subrange (see 6.1.3) type *T* are (with *x*: *T* and *y*: Integer):

T.first	the first value (in the enumeration).
T.last	the last value (in the enumeration).
T.Succ(x)	the value succeeding x (in the enumeration). *T*.Succ(*T*.last) is undefined.
T.Pred(x)	the value preceding x (in the enumeration). *T*.Pred(*T*.first) is undefined.
T.Ord(x)	an Integer which is the ordinal number of the value x in the enumeration of *T*. Thus, *T*.Ord(*T*.first) = 0.
T.Val(y)	a value of type *T*, such that *T*.Val(*T*.Ord(x)) = x and *T*.Ord(*T*.Val(y)) = y.
T.Max(x_1, x_2)	the x_i for which *T*.Ord(x_i) is larger.
T.Min(x_1, x_2)	the x_i for which *T*.Ord(x_i) is smaller.

For instance, *Suit*.last is *spade*, and *Day*.first is *Monday*.

6.1.2. Standard simple types

The following types are standard in Euclid, and are pervasive throughout all programs (see 7.3):

Integer

Its values are the positive and negative integers, in the mathematical sense. It is not possible to declare a variable to be of type Integer. Instead, variables can be declared to be of some suitable subrange type. Constants of type Integer may be declared, however, and numbers are literal constants of type Integer. Note that no standard components are defined for Integer.

SignedInt, UnsignedInt

Every implementation has these two standard types; they are ordinary subranges of Integer which are pre-defined for convenience. The intention is that they should be large subranges of Integer type that can be handled efficiently by the machine, and that contain:

for SignedInt, equal numbers of positive and negative numbers, or perhaps one more negative number.

for UnsignedInt, 0 and no negative numbers.

If *T* is defined by **type** *T* = *m* .. *n*, for any manifest constants *m*, $n \geq 0$, and *i* is a value of type *T*, the standard representation of *i* is the ordinary binary representation of the integer *i*, filled out on the left with any number of extra zeros. The standard representation of a signed integer must be defined by the implementation, but is not defined in this report.

There are two standard functions defined on any subrange of Integer:

Abs(x)	returns an Integer which is the absolute value of x.
Odd(x)	returns a Boolean which is true if and only if x is odd.

Boolean

Its values are the truth values denoted by the identifiers false and true. It is defined by **type** Boolean = (false, true).

Char

It is an enumerated type whose values are a set of characters determined by the implementation, but including at least the 26 capital letters and the 10 decimal digits. They are denoted by the characters themselves preceded by a dollar sign (see 4.).

There is a standard function Chr identical to Char.Val.

The ordering of the values of type Char is implementation dependent. Use of this ordering in comparisons of Chars, subranges of Chars, or the Chr and Char.Ord functions, will in general result in non-portable programs. Unlike other machine-dependent features of Euclid, this one is not restricted to machine-dependent modules. However, the ordering must obey

$A<$B<...<$Z, $a<$b<...<$z, and $1=Char.Succ($0),..., $9=Char.Succ($8).

StorageUnit — It is the basic unit for storage allocation (see 6.3). There are no distinguishable values or constants of this type, and no operations are defined on this type. Thus, a StorageUnit variable simply serves to occupy a known amount of space. The standard representation of a StorageUnit is not defined.

There is a standard component of the type StorageUnit:

sizeInBits — an Integer constant which defines the number of bits in a StorageUnit.

There is a standard component of every StorageUnit variable:

address — the machine address (of type AddressType) of the variable.

AddressType — It is a non-negative subrange of Integer, large enough to hold a full machine address; i.e., a value of *x*.address, if *x* is a StorageUnit variable.

6.1.3. Subrange types

A type *S* may be defined as a subrange of another simple type *T* by indication of the smallest and the largest value in the subrange. The first constant specifies the lower bound (*S*.first), and the second the upper bound (*S*.last). If *S*.first>*S*.last there are no values of this type. When a record or array component has a subrange type, the type must be manifest; this restriction ensures that the packing of array and record components is manifest.

If type *A* is a subrange of type *B*, and type *B* is a subrange of type *C*, we say that *A* is also a subrange of *C*. The Succ, Pred, Ord, Val, Max, Min, first and last components are defined for all subrange types. If *A* is a subrange of *B*, and *a* is of type *A* and *b* is of type *B*, and *a*=*b*, and the standard representation of *b* is defined, then it is the same as the standard representation of *a*.

```
subrangeType    ::= constantSum ".." constantSum
constantSum     ::= sum
```

Examples:

```
type OneToOneHundred = 1 .. 100
type SymmetricRange = -10 .. 10
type Primary = red .. blue {the values of a Primary are red, green, and blue}
type ScreenPosition = 1 .. 480 {y coordinate for display screen}
```

6.2. Structured types

A *structured type* is characterized by the type(s) of its components and by its structuring method. Moreover, a structured type definition may contain an indication that a packed data representation is preferred: if a definition is prefixed with the symbol **packed**, this is a hint to the compiler that storage should be economized even at the price of some loss in efficiency of access, and even if this may expand the code necessary for accessing components of the structure. Adding occurrences of **packed** may make a legal program into an illegal one (because of type compatibility (see 6.4) or if a component of the structure has been renamed as an entire variable (see 7.4)), but will not otherwise change the meaning of the program.

```
structuredType          ::= [ packed ] unpackedStructuredType | derivedStructuredType |
                            structuredTypeAppl
unpackedStructuredType  ::= arrayType | recordType | moduleType | mdRecordType |
                            setType | collectionType | pointerType
derivedStructuredType   ::= [containingVariable "."] structuredTypeIdentifier
```

6.2.1. Array types

An array type is a structure consisting of a fixed number of components that are all of the same type, called the *component type*. The elements of the array are designated by indices, values belonging to the *index type*. The array type definition specifies both the component type and the index type. The index type may not be StorageUnit. If the component type is a subrange, the bounds must be manifest.

The standard representation of an unpacked array A of type **array** I **of** C is defined as follows. Let $t = C.\text{size} + (C.\text{alignment} - 1)$, and $s = t - (t \bmod C.\text{alignment})$. Then successive components of A occupy successive groups of s StorageUnits, with no unoccupied StorageUnits in between. $A(I.\text{first})$ occupies the first s StorageUnits, i.e., the ones with the smallest machine addresses, and $A(I.\text{last})$ occupies the last s StorageUnits, i.e., the ones with the largest machine addresses. Each component is aligned in the same way as a variable of type C. The entire array thus occupies $\text{Max}(0, s*(I.\text{last} - I.\text{first} + 1))$ StorageUnits. The standard representation of a packed array is not defined.

```
arrayType       ::= array indexType of componentType
indexType       ::= simpleType
componentType   ::= type
```

There are two standard components of an array type T:

T.IndexType	the index type
T.ComponentType	the component type

Like other components of types, they are also components of any variable or constant of the type (see 6.). Thus if a is a variable of type T, then a.IndexType is the same as T.IndexType, and likewise for ComponentType.

Examples:

```
type Array0 = array 1 .. 100 of SignedInt
type Array1 = array -10 .. 10 of 0 .. 99
type Array2 = array Boolean of Color
type NameTable = array OneToOneHundred of String(50)
```

There are two standard pervasive identifiers having to do with strings, declared as follows.

```
type StringIndex = 1 .. stringMaxLength
type String(length: StringIndex) = packed array 1 .. length of Char
```

Literal string constants are of type String, with *length* equal to the number of characters. Routines can readily be defined to extract substrings, do pattern matching, or perform any other desired operations on strings (see 10. and 11. for examples). Furthermore, these routines might be machine-coded for efficiency. The value of the pervasive manifest constant stringMaxLength is implementation-defined.

6.2.2. Record types

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a *field*, its type and an identifier which denotes it. The scope of these *field identifiers* is the record definition itself. They are also accessible within a field designator (see 7.2) referring to a variable or constant of this type. Record

components may be constants or variables. Constant components are accessible within a field designator referring to the type itself, as well as in a designator referring to a variable. Neither a variable binding nor a module type definition may appear in a record. If the type of a component is a subrange, the bounds must be manifest.

For the syntax of constant and variable declarations, see 7. If the record type appears as the definition of a type identifier, it must end with the clause **end** identifier; otherwise it must end with **end record.**

The size of a record containing an unpacked array is equal to some constant value, independent of the array size, plus the size of the array. The standard representation of a record is not defined.

```
recordType           ::= record fieldList endRecord
endRecord            ::= end record | end identifier
fieldList            ::= [ recordDeclaration ";" ] [ variantPart ] ";"
recordDeclaration    ::= recordDeclarationPart { ";" recordDeclarationPart }
recordDeclarationPart::= constantDeclaration | var variableDeclarer
```

A record type may have several *variants.* In this case a constant identifier of some manifest simple type must be used as a selector in a **case** construction that enumerates the possible variants. This identifier is called the *tag,* and its value indicates which variant is assumed by the record variable at a given time. Each variant structure is identified by a *case label,* which is a set of manifest constants of the type of the tag. The tag must be a manifest constant or a formal parameter of a prototype declaration in which the **case** appears (see 6.3). When the prototype is applied, however, the actual parameter corresponding to such a formal must be a manifest constant, or **any.** In the latter case, the default clause specifies the variant to which values of the type will be initialized.

A variant record *r* has a standard component

r.itsTag	the value of the tag.

The case label lists must be disjoint. Furthermore, the union of the lists must exhaust the enumerated type of the tag, unless there is an **otherwise** variant, in which case all the tag values not mentioned explicitly are lumped under that variant. The case label following the **end** of each variant must be one of the labels specified by its case label list.

```
variantPart          ::= case tag [default manifestConstant] of variant { ";" variant }
                         [ otherwiseVariant ] ";" end case
variant              ::= caseLabelList "=>" recordDeclaration ";" end caseLabelEnd | empty
caseLabelList        ::= caseLabel { "," caseLabel }
caseLabelEnd         ::= literal | manifestConstantIdentifier | "(" caseLabel ")"
caseLabel            ::= manifestConstant | subrangeType
tag                  ::= identifier
otherwiseVariant     ::= otherwise "=>" recordDeclaration
```

Examples:

```
type Date = record
      var day: 1 .. 31
      var month: (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)
      var year: 1900 .. 2100
      end Date

type Stream (dev: Device) = record
      case dev default keyboard of
       display =>
            var first, last: DisplayControlBlock {not defined in this report}
            var height: ScreenPosition := ScreenPosition.first
            var nLines: 0 .. (ScreenPosition.last) div 8
```

```
        end display
    tape, disk =>
        var file: FileHandle {not defined in this report}
        var position: UnsignedInt := 0
        var buffer: array 0 .. 255 of Char
        end tape
    keyboard =>
        var buffer: array 0 .. 20 of Char
        var bufFirst, bufLast: buffer.IndexType := buffer.IndexType.first
        end keyboard
    otherwise => {null fieldList}
    end case
end Stream
```

6.2.3 Module types

A module type is a generalization of a record type. Module components may be declared as constants, variables, routines, types, or prototypes. These declarations have the same form and the same meaning as the declarations in a block. Thus, a module serves as a package for a collection of related objects. A module type or prototype declaration is always opaque; i.e., it is not the same as any other type (see 6.4). Thus, two different module declarations always define different types, even if the definitions are textually identical. Except for this consideration and the visibility restrictions, however, a module definition is equivalent to a record definition containing the module's constants and variables, together with separate declarations of the module's routines and types. References to the constant and variable components within the routines are explained in 6.2.3.2.

The optional identifier following **module** in the type definition is declared as a variable in the scope of the definition, and its type is the module type. It may be used within the definition to name the entire value, i.e., for self-referencing.

The standard representation of a module is not defined.

```
moduleType      ::= [ machine dependent ] module [ identifier ]
                    importClause exportClause moduleBody endModule
endModule       ::= end module | end identifier
moduleBody      ::= checkedClause declaration ";" initialAction invariant finalAction
checkedClause   ::= checked | not checked | empty
declaration     ::= empty | pervasive declarationPart { ";" pervasive declarationPart }
declarationPart ::= constantDeclaration | variableDeclaration | typeDeclaration |
                    procedureDeclaration | functionDeclaration | converterDeclaration |
                    assert assertion
pervasive       ::= pervasive | empty
invariant       ::= [abstraction functionDeclaration ";"] invariant assertion ";" | empty
assertion       ::= "(" expression ")" | empty
```

The following example outlines how one might package floating-point numbers and operations on them in a module. An example of a complete module may be found in section 12.

```
var Real: machine dependent module
    exports (
        Add, Subtract, Times, Divide, Greater, zero, Number,
        Value with (:=, =, sign, exponent, mantissa) )

    type Exp = -80#16 .. 7F#16;
    type Mant = 0 .. 0FFFFFF#16
    type Value = machine dependent record
        var sign (at 0 bits 0 .. 0): 0 .. 1 := 0
        var exponent (at 0 bits 1 .. 7): Exp := 0
```

```
        var mantissa (at 0 bits 8 .. 31): Mant := 0
        end Value
const zero: Value := (0, 0, 0)

inline function Number(m: SignedInt, ex: SignedInt) returns num: Value =
        imports (Mant)
        begin
        if m<0 then num.sign := 0 else num.sign := 1 end if
        num.mantissa := Abs(m); num.exponent := ex
        end Number

function Add(const l, r: Value) returns sum: Value =
        code ... end Add

converter ValToInt(Value) returns SignedInt

inline function Greater(const l, r: Value) returns g: Boolean =
        imports (ValToInt)
        begin {use type converters to compare l & r as SignedInts}
        return (ValToInt(l)>ValToInt(r))
        end Greater
...
end Real
```

6.2.3.1 Exporting

The scope of a module's field identifiers is the module definition itself. Those which are *exported* by the module definition are also accessible within a field designator referring to a variable or constant of the module type, in any scope in which the module type is accessible. Furthermore, in such a scope constant and type components are accessible within a field designator referring to the type itself. The packaging supplied by the module thus provides protection against improper use of components that are intended to be known only within the module definition.

A constant identifier is always exported as a constant; this may be specified by a binding condition of **const**, or the binding condition may be omitted. A variable identifier may be exported as a variable using the binding condition **var**, or as a readonly variable using the binding condition **readonly** or omitting the binding condition (see 7.). A variable may not be exported as a constant. No variable in the explicit import or export list of a module may overlap any other such variable (see 7.4).

A constant or variable identifier *i* may be exported only if

> all identifiers used in the type definition in the declaration of *i* are exported or accessible in the scope S enclosing the module definition; and
>
> *i* is a field identifier of the module, i.e., it is declared at the outermost level in the module.

Similarly, a routine identifier *R* may not be exported unless all identifiers used in the type definitions of its formal parameter list and return type (if any) are exported or accessible in S, and *R* is declared at the outermost level.

The ItsType component of an exported constant or variable is automatically exported.

When a type *T* is exported from a module *M*, it is always opaque outside its defining module; i.e., it is not the same as any other type (see 6.4). Thus all operations on values of type *T* and literal constants of type *T* are inaccessible outside *M*, except that :=, =, ↑, subscripting, field identifiers, and enumerated value identifiers may be explicitly exported by a **with** clause appended to the export item for the type. Anything not specified is not exported. Thus:

> *T* is assignable outside *M* if and only if assignment is exported.

If *T* is an array type, subscripting is permitted outside *M* if and only if the IndexType component is exported.

If *T* is a pointer type, dereferencing is permitted outside *M* if and only if "↑" is exported.

T's field identifiers (including standard ones, like size) are accessible outside *M* if and only if the **with** clause specifies that they should be exported.

If *T* is an enumerated type, its enumerated value identifiers are accessible outside *M* if and only if the **with** clause specifies that they should be exported; subrangeType (with manifest bounds) is permitted in an exportItem to facilitate this. Note that like all exported identifiers, they may only be used in suitable field designators.

Values of the module type itself may not be assigned or compared for equality unless := or =, respectively, appear in the export list. Furthermore, := and = may not appear in the export list if the module imports any variables, and := may not appear if it has a final action.

```
exportClause        ::= exports exportList ";" | empty
exportList          ::= "(" exportItem { "," exportItem} ")"
exportItem          ::= bindingCondition identifier [ with exportList ] | ":=" | "=" | "↑" |
                        subrangeType
bindingCondition    ::= const | readonly | var | empty
```

6.2.3.2 Importing

A module is a closed scope; i.e., identifiers declared outside the module are not known inside unless they are known in the immediately enclosing scope, and either are pervasive or are imported into the module (by the import clause in the module definition, or because they appear in the formal parameter list of the module declaration). Importing is discussed in detail in 7.3. Note that variables imported into a module may not be used except in the import lists of routines or modules declared in the module, or as collection variables or zones (see 6.). Hence, they cannot affect any variables, constants or functions declared in the module. A module may not import a routine declared with **forward** if the routine's true declaration has not yet appeared; hence mutual recursion of routines and modules is impossible. Note that mutual recursion of routines in two modules is impossible because a forward type may not appear in an import list. See 10. for the use of **forward** in an import list.

A routine *R* declared in a module *M* may import a constant or variable identifier *x* of type *T* declared in *M*. The meaning of *x* in *R* is explained as follows. Replace *R* with another routine *R'* which does not import *x*, but instead has an additional formal parameter *x*: *T* with the same binding condition. Replace every call *Mi.R*(arguments) with *Mi.R'*(*Mi.x*, arguments), where *Mi* is an expression of type *M*; export *x* if necessary to make this legal. Replace every call *R*(arguments) inside *M* with *R'*(*x*, arguments). Importing a routine or type identifier declared in *M* is equivalent to duplicating its definition within *R*.

```
importClause        ::= singleImportClause { ";" singleImportClause } | empty
singleImportClause  ::= imports importList [thus importList] ";"
importList          ::= "(" importItem { "," importItem } ")" | "(" ")"
importItem          ::= [ forward ] pervasive bindingCondition identifier
```

6.2.3.3 Machine-dependency

A module type may be prefixed with the symbols **machine dependent**, which simply means that it may contain

other machine-dependent module and record declarations,

fixed addresses in variable declarations (see 7.),

extended characters of the form $*ddd* (see 4.),

type converters involving types with implementation-dependent representations (see 6.5),

uses of the address component of a StorageUnit variable (see 6.1.2),

machine-code routine declarations (see 10. and 11.).

A module that is not machine-dependent may not contain any of these things. Thus it is possible to tell from the heading of a module type whether it is allowed any machine dependencies or not. Note that machine-dependent identifiers exported from a machine-dependent module, as well as the machine-dependent module type itself, may be used in other modules that are not machine-dependent. Since exported types can only be manipulated by exported routines, the routines serve to encapsulate the machine-dependencies.

6.2.3.4 Initial and final actions

A module may include an *initial action*, which is executed whenever a new variable of the module type is created, and a *final action*, which is executed whenever such a variable is destroyed. These actions are considered to be parameterless procedures declared in the module. If several module variables are declared, as in the program fragment

begin *x*:*M1*; *y*:*M2*; *z*:*M1*; ... **end**

the order of creation is *x*, *y*, *z*, and the order of destruction is *z*, *y*, *x*. This follows from the fact that the three declarations start three nested scopes, all of which end at the **end** (see 7.3). If an array *a* of modules is declared, the order of creation is *a*(*a*.IndexType.first) .. *a*(*a*.IndexType.last), and the order of destruction is the reverse. If a record containing modules is declared, the order of creation is the order of the components, and the order of destruction is the reverse.

The meaning of final actions is precisely explained by the following rewriting rule. If *M* is a module type with a final action, let *M'* be the same type except that the final action is replaced by an exported procedure named Finally with the same routine definition. Then if S is any statement,

begin var *x*:*M*; S **end**

is short for

begin var *x*:*M'*; **begin** S' **end** ; *x*.Finally **end**

where S' is the same as S except that every occurrence of

exit when B;

not in a nested reepetitive statement, is replaced by

if B **then** *x*.Finally; **exit end if**;

and likewise for each **return** not in a nested closed scope. Note that since *M* is not assignable, an identifier of type *M* cannot be introduced by a constant parameter or function result declaration.

The extension to arrays and records containing such modules is straightforward, using the rules of the previous paragraph. A collection of such modules is also possible; *x*.Finally is called when *x* is freed (see 6.2.6).

initialAction ::= **initially** routineDefinition ";" | empty
finalAction ::= **finally** routineDefinition ";" | empty

6.2.3.5 Invariants and abstractions

A module may also specify an *invariant* that is supposed to be true during the lifetime of the module variable (i.e., after the execution of the initial action and before the execution of the final action), except perhaps when one of the procedures of the module has been called and has not yet returned. Like other assertions, this one may be empty or a Boolean expression. In the former case

the content of the assertion is supplied in a comment. This comment is of course ignored by the compiler, but presumably is interpreted by the verifier. In the latter case the assertion must be a legal Boolean expression; code is generated to check it, whenever checking is enabled for the scope enclosing the module type definition (see 9.2.1):

on every entry to or exit from any exported procedure in the module;

on every assignment to an exported variable outside the module;

after the initial action and before the final action.

The *abstraction function*, necessary for one approach to the verification of modules [Hoare 1972, London et al. 1977], maps the values inside the module into a value of the module type. The need for this mapping in verification arises because a variable of the module type may be represented inside the module by different variables. The body of the abstraction function may contain constructs outside Euclid in the same way that assertions may, i.e., within comments. The abstraction function is not callable from a Euclid program, and is treated as a comment by the compiler.

6.2.4 Machine-dependent records

A machine-dependent record type is a restricted kind of record type that allows (in fact, requires) the programmer to specify the exact position and size of each variable field. The position is specified in StorageUnits, where the first StorageUnit of the record is numbered 0, and then in bits, where the first bit of the specified StorageUnit is numbered 0, and the bit numbering continues to successive StorageUnits in the obvious way. The ordering of bits in a StorageUnit is implementation-defined. If the **bits** clause is omitted, the field occupies an integral number of StorageUnits, and its size is computed from the size of its type; the value is right-justified in the field. The compiler's responsibility is to check that fields do not overlap and that each field is at least large enough to hold values of its type. The size of a value of machine-dependent record type is equal to the maximum value of

$$1+a+((b.\text{last}+\text{StorageUnit.sizeInBits}-1)\ \textbf{div}\ \text{StorageUnit.sizeInBits}$$

over all the variable components, where a is the value following **at**, and b is the simple type following **bits**. An implementation may place restrictions on how fields overlap natural storage boundaries. The type specifying **bits** must be a subrange of Integer, and must be manifest except possibly for the last component of the record.

A machine-dependent record may have constant components like an ordinary record. It may not be the definition of a prototype. All its variable components must have position specifications, and they cannot be exported, passed as variable parameters, or bound to. Furthermore, they must all have types whose standard representation is specified. Note that the standard representation of a machine-dependent record is specified, but the standard representation of an ordinary record is not.

An alignment clause, **aligned mod** *a*, in a machine-dependent record declaration forces a value of the record type to be allocated so that the machine address of its first StorageUnit is 0 **mod** *a*; *a* must be a power of 2, and an implementation may limit its maximum value.

The module in which a machine-dependent record type appears must be a machine-dependent module.

```
mdRecordType        ::= machine dependent record [ alignmentClause ]
                        [ mdDeclarationPart { ";" mdDeclarationPart } ] ";" endRecord
mdDeclarationPart   ::= constantDeclaration |
                        var identifier "(" at manifestConstant [ bits subrangeType ] ")"
                        ":" typeDefinition [ initialization ]
alignmentClause     ::= aligned mod manifestConstant
```

Example:

```
type InterruptWord = machine dependent record aligned mod 8
    var device (at 0 bits 0 .. 2): DeviceNumber;
    var channel (at 0 bits 3 .. 5): 0 .. 7;
    var stopCode (at 0 bits 6 .. 7): (finishedOk, errorStop, powerOff);
    var command (at 1 bits 0 .. wordSize): ChannelCommand
    end InterruptWord
```

6.2.5. Set types

A set type defines the range of values which is the powerset of its base type. Base types must be manifest simple types other than StorageUnit. An implementation may restrict the cardinality of a base type (see 13.). Operators applicable to all set types are:

+	union
−	set difference: *i* appears in *a*−*b* if and only if it appears in *a* and not in *b*.
xor	symmetric difference: *i* appears in *a* **xor** *b* if and only if it appears in exactly one of *a* and *b*.
*	intersection
in	membership
<=, >=	set inclusion

Sets can be built up from values of the base type as described in 8. The standard representation of a set *S*: **set of** *B* is defined if the standard representation of values of *B* is defined. It is a sequence of *n* significant bits, where $n=(B.\text{last}-B.\text{first}+1)$, preceded by any number of insignificant zero bits. If the significant bits are numbered 0, 1, ..., $n-1$, then bit *i* is one if and only if *x* is in *S* and $B.\text{Ord}(x)=i$.

setType	::=	**set of** baseType
baseType	::=	simpleType

There is one standard component of a set type *T*:

T.BaseType	the base type

Examples:

```
type Hue = set of Color
type SubtractivePrimaries = set of red .. green
type SymSet = set of −5 .. +5
type EntriesInUse = set of Array1.IndexType
```

6.2.6. Pointer and collection types

A variable which is declared in a program (see 7.) is denoted by its identifier. The variable exists during the entire lifetime of the scope to which it is local, and such a variable is therefore called static. In contrast, variables may also be generated dynamically, i.e., without much correlation to the structure of the program. These *dynamic* variables are generated by the standard procedure component New described below; since they do not occur in an explicit variable declaration, they cannot be denoted by an identifier. Instead, they may be referred to by a *pointer* value which is provided by New when the dynamic variable is generated. A pointer type thus consists of an unbounded set of values pointing to elements of the same type. No operations are defined on pointers except the test for equality, the pointer following operator ↑, which yields the variable referred to by the pointer, and the standard function component Index, which converts a pointer into an Integer.

The standard representation of a pointer is the same as the standard representation of AddressType.

A dynamic variable must be an element of a *collection*. A collection is not a type: it is a variable

that behaves very much like an array variable. Just as an element of an array variable *A* can be referenced by subscripting *A* with an index whose type is the index type of *A* (*A*.IndexType), so an element of a collection *C* can be referenced by subscripting *C* with a pointer whose type is the pointer type of *C* (↑*C*). There are two differences:

> No two collections have the same pointer type. Hence the pointer alone is sufficient to specify the collection, and we allow *p*↑ as shorthand for *C*(*p*), where *p* is of type ↑*C*.
>
> There are no operations that produce pointer results, except explicit type conversion and the standard procedure *C*.New that creates a new variable. Hence the storage allocation strategy for collections can be quite different from the strategy for arrays.

The reason for having collections is that two pointers to different collections are guaranteed to point to different variables; two pointers to the same collection are either equal, and point to the same variable, or unequal, and point to non-overlapping variables. Hence collections are a means by which the programmer can express some of his knowledge about the ways in which his program is using pointers. If he prefers not to do this, or has no knowledge about pointers to variables of type *T* which can be expressed in this way, he can simply declare a single collection of *T*s and use it everywhere.

There are no operations on collections. A collection may not be assigned to another collection. In fact, there is nothing to do with a collection except to subscript it, or to pass it as a **var** actual parameter. A collection type may not be the component type of an array or record or the object type of a collection.

Associated with every collection is a *zone*, which provides storage for its variables. A zone is a module variable with three special components (and possibly other components):

> a variable *storageBlocks*, which is a collection of a record type containing a special variable component (and possibly other components):
>
> > *theStorage*, a StorageUnit
>
> a procedure *Allocate*(*size*, *alignment*: UnsignedInt, **var** *pointer*: ↑*storageBlocks*)
>
> a procedure *Deallocate*(*pointer*: ↑*storageBlocks*, *size*: UnsignedInt)

These components must be exported; they are intended only for use by the standard procedures New and Free. A zone must be a machine-dependent module variable. Note that a collection *C*'s zone must be imported as a variable into any scope in which *C*.New or *C*.Free is called. If *C* is reference-counted, *C*.zone must also be imported as a variable into any scope in which a non-local variable of type ↑*C* is assigned to.

A collection declared without a zone will use a standard zone called systemZone. This zone is not pervasive (since it is a variable), but must be imported where it is needed.

A collection *C* can be *reference-counted*, in which case a variable in *C* will be freed automatically when no pointers to it remain and no identifiers are bound to it (see 7.4); note that inaccessible circular structures will not necessarily be freed. The optional manifest constant is an UnsignedInt that gives the maximum reference count that should be maintained; if more than this number of pointers to a variable in *C* ever exist at one time, the program is illegal. The Free procedure does not exist for a reference-counted collection. Space is allocated for the reference count in each variable in *C*.

A collection *C* must be *checkable* if *C*.Free is to be used in a checked scope. A checkable collection incurs checking overhead in *all* scopes, whether checked or unchecked. Aside from execution speed, the size of the compiled program, and the amount of space required in the system zone, the presence or absence of **checkable** has no effect. In particular, it cannot be determined by the program itself. See 14.12 for an implementation note.

This is the only situation in which checking a scope requires any change elsewhere in the program. The resulting inconvenience can perhaps be reduced by converting back and forth between **checkable** and { **checkable** } in declarations, with the help of a text editor.

There is one standard component of a dynamic variable *v* in a counted or checkable collection:

v.refCount	the number of pointers to *v*, plus the number of identifiers bound to *v*.

Note that assignment of arrays containing pointers to counted or checkable collections will result in loops to do the necessary refCount adjustments. Likewise, the assignment of records with variant parts containing such pointers will result in tests of the tag to conditionally do the necessary adjustments.

collectionType	::=	countControl **collection of** objectType [**in** zoneIdentifier]
countControl	::=	**counted** [manifestConstant] | **checkable** | **empty**
objectType	::=	type
pointerType	::=	"↑" collectionVariable

There are six standard components of a collection variable *C*:

C.nil	a pointer that points to no variable at all.
C.ObjectType	the object type. If there are any **unknowns** in the collection definition, *C*.ObjectType is a prototype with formal parameters corresponding to the **unknowns**.
C.zone	the zone (see above).
C.Index(*obj*: ↑C)	a function that takes a pointer to *C* and returns an Integer. This function has only one defined property: it is one-to-one.
C.New(**var** *p*: ↑C)	allocates a new variable *v* in collection *C* and assigns a pointer to *v* to the pointer variable *p*. *C*.New imports *C* as a variable. This procedure works as follows: It first calls *Allocate* for the pointer's zone with some $s \geq T$.size, and *T*.alignment, as parameters, where $T = C$.ObjectType. It gets back a ↑*storageBlocks*, and uses the *theStorage* component in this block as the first StorageUnit for the newly created variable. (It is up to the verifier of the zone to ensure that a sequence of at least *n* free StorageUnits begins there if *Allocate*(*n*, *i*, *p*) was called, and that the storage allocated does not overlap with that of any other variable.) Last, any initialization specified by the type of *v* is performed. If *C*.ObjectType is a prototype, then values for the parameters corresponding to **unknowns** must be supplied as additional parameters to New, in the same order in which they appear in the prototype's formal parameter list, so that the variable being created will have a definite type.
C.Free(**var** *p*: ↑C)	frees the variable *v* pointed to by *p* and sets *p* to *C*.nil. *C*.Free imports *C* as a variable, and works as follows: Any finalization specified by the type of *v* is performed. Then the *Deallocate* procedure for *C*'s zone is called with a pointer to the *storageBlocks* variable from which *v* was originally allocated by *C*.New, and the size that was given to *C*.New. This procedure is not defined for reference-counted collections. *C*.Free(*p*) is illegal if there is any other pointer value in existence equal to *p*. Thus the legality assertion for *C*.Free(*p*) is *p*.refCount = 1. It is illegal to use *C*.Free in a checked scope unless *C* is checkable.

Note that New and Free are procedures that violate the strict type checking of Euclid. These procedures, explicit type conversions (see 6.5) and machine code routines (see 10), are the only ways of doing so.

Examples:

```
var myStreams: collection of Stream(unknown) in ioZone
type StreamRef = ↑myStreams
var userInput, userOutput, anyStream: StreamRef
...
myStreams.New(userInput, keyboard)          {create & initialize input stream}
myStreams.New(userOutput, display)          {also an output stream}
var stringStorage: collection of String(unknown)
type StringRef = ↑stringStorage
```

6.3 Prototypes

It is possible to declare a prototype by including a formal parameter list in a type declaration:

```
type T(a: SignedInt, b: Color) = U
```

Every use of *T*, except in an import or export list, must be an *application*, with an actual parameter list that supplies values for all the formal parameters. Thus, a prototype is a template, from which a number of types can be obtained by supplying actual parameters for the formals. The formal parameters of a prototype must be constant, and hence of an assignable type. An identifier *x* that is accessible as a field identifier of *U* may not also identify one of *T*'s formal parameters; this rule prevents *x* from being ambiguous in the context *v.x*, where *v* is of type *T*(...).

When a prototype application appears in the formal parameter list of a procedure, an actual parameter of the application can be a previous formal parameter of the procedure (see 10.). Thus, procedures can be written to accept actual parameters whose type is any application of a prototype.

The built-in type constructors *i..j* (subrange), **case** *c* **of** ... (variant part of a record type), and ↑*C* (pointer) also take parameters. In fact, the first two can take parameters of any simple type, and the last can take any collection variable. Thus all three are unlike user-defined prototypes, in which the types of the parameters are specified in the formal parameter list. For subrange and **case** the actual parameters must be constants, but need not be manifest. Thus, textually identical occurrences of one of these constructors, like identical prototype applications, do not necessarily produce the same type.

The **case** constructor is normally used in the declaration of a prototype *T* in which its parameter is in turn declared to be a (necessarily constant) formal parameter of *T*. Note that when *T* is applied, actuals must be supplied for all its formals, even though some of the formals may be used only in a variant that is not selected. Furthermore, the actual parameter supplied for the tag must be a manifest constant, **any**, or **unknown**.

Parameters of a prototype application may be referenced like record components; thus after

```
type T(p: color, q: Boolean) . . .; var x: T(red, true)
```

the expression *x.p=red* is true. Note that the parameters of a module type need not be exported, since they are declared outside the module definition.

When a prototype *T* is applied (e.g., in the declaration of a variable), the actual parameters are substituted for the formals, producing a type. If the definition of *T* contains references to further prototypes, these in turn must have actual parameters which are substituted for these formals. This expansion process proceeds recursively until finally a type definition with no prototypes results. In certain cases, however, Euclid makes it possible to defer fixing the value of a parameter. In particular:

- If a formal parameter is the tag of a variant, its actual may be the symbol **any**; then a variable can be changed from one variant to another during execution, by assigning values of different variants to the variable.

If an application of *T* is used as the object type of a collection, one or more actuals may be the symbol **unknown**, and fixed only when a variable in that collection is created.

If an application of *T* is the type of a formal parameter of a routine, one or more actuals may be the symbol **parameter**. This is a shorthand which indicates that they are to be passed as additional parameters of the routine.

The first two cases are described in detail below; for the third, see 10.

The symbol **any** may be used as an actual parameter of a prototype application, provided that the corresponding formal is used only as the tag of a variant. Suppose *V* is such a prototype, with a formal parameter *s*, of enumerated type *T*, used as a tag (there might be other formals, but they are omitted in this example). Then *V*(**any**) is a type whose values are the union of the values of *V*(*i*) as *i* ranges over all the elements of *T*. It differs from any particular *V*(*i*) in two important ways:

If *x* is declared to be of type *V*(**any**), only those components of *x* that are outside the case constructor with tag *s* can be referenced. A discriminating case statement (see 9.2.2.2) can be used to bind *x* to an identifier *y* whose type is *V*(*i*), and then all the components of *y* can be referenced in the scope of the discrimination.

The value of the tag *x*.itsTag, and hence the choice of variant, can be changed during execution by assignment to *x* (but not, of course, to *y* if *y* is of type *V*(*i*)). This is the only case in which any property of a variable which is determined by the parameters of its type can be changed after the variable has been created.

Note that **any** is not a value; *x*.itsTag will never be **any**, but will be equal to the current value of the tag, and *x.s* is undefined.

The symbol **unknown** may be used as an actual parameter in a prototype application which appears as the object type of a collection. A variable in the collection can only be created by the standard procedure New, however (see 6.2.6), and when New is called, actual parameters must be supplied for all the **unknowns** in the object type; note that **any** is not a legitimate actual parameter in this case. Hence a type never involves **unknown** except in the object type of a collection. In this case the standard representation of variables in the collection is undefined.

When a pointer to **collection of** *T*(..., **unknown**, ...) is dereferenced to yield a variable *v*, that variable has type *T*(..., *x*, ...), where *x* is the value that was supplied to New when *v* was created. As in other cases where the parameters of types are not manifest constants, the compiler may have to generate legality assertions to ensure that the type of a dereferenced pointer has some property demanded by the context in which it is used. If the **unknown** parameter is only used as the tag of a variant, a discriminating case statement can be used to bind a referenced variable to an identifier of known type, just as is done with **any**.

Note that all actual parameters in an object type other than **any** and **unknown** are evaluated when the collection is declared, *not* when a variable in the collection is created.

```
prototypeDeclaration ::= type typeIdentifier typeFormalList = preAssertion typeDefinition
typeFormalList       ::= "(" typeFormalSection { "," typeFormalSection } ")"
typeFormalSection    ::= identifier { "," identifier } ":" indexType
simpleTypeAppl       ::= derivedSimpleType typeActualList
structuredTypeAppl   ::= derivedStructuredType typeActualList
typeActualList       ::= "(" typeActualParameter { "," typeActualParameter } ")"
typeActualParameter  ::= expression | any | unknown | parameter
```

Examples of type definitions:

```
type FamilyMember(sex: SexType) = forward
var members: collection of FamilyMember(unknown)
type FamilyMember = record
```

```
        var age: 0 .. 100
        var mother, father, sibling: ↑members
        var oldestChild: ↑members
                case sex default female of
                        female      => var husband: ↑members end female
                        male        => var wife: ↑members end male
                        end case
        end FamilyMember

type Subject = FamilyMember(any)
type Family = ↑members
```

6.4 Type compatibility

This section defines the conditions under which two types are the same, and describes the rules for type compatibility in the language. The basic idea is this: a type identifier is an abbreviation for its definition. After all such abbreviations have been removed, two types are the same if their definitions look the same. However, a module type identifier, or any type identifier exported from a module, is considered to be different from any other type, hence operations on such a type are restricted to these exported from the module.

Two types are the *same* if their *expanded definitions* are equal. The expanded definition of a type is obtained by the following algorithm:

Start with the type.

Replace each type or prototype identifier by its definition, unless the definition is a module type, or the identifier was exported from a module. During this replacement, substitute any actual parameters for the corresponding formals.

Replace *x*.ItsType by the type of *x*.

If *x* is a module variable and *T* an exported type, replace *x* in *x.T* by the type of *x*.

Repeat these replacements until there are no more to be done.

The result is the expanded definition.

Two expanded definitions are equal if,

when all *extended parameters* are removed, they are identical sequences of basic symbols;

the values of corresponding extended parameters in the two sequences are equal.

The extended parameters of an expanded definition are the constant expressions that appear as

actual parameters of unexpanded prototype identifiers,

parameters of subrange or **case** constructors,

case labels in a variant,

constant definitions,

following **at** or **aligned** in a machine-dependent record declaration.

If the compiler cannot determine whether or not two types are the same (e.g., because their extended parameters are not manifest), and they must be the same for the program to be legal, then the compiler will assume that they are the same, and generate a legality assertion guaranteeing this fact.

When a value is assigned to a variable, or a variable is bound to an identifier, the types must be *compatible* according to the following rules:

In an assignment, both types must be the same, except that when the left side is a variable *v* of a simple type, then

> If *v* is of a subrange type, its range may differ from the range of the expression on the right side (note, however, that other parameters of types, such as array bounds, may *not* differ). In a legal Euclid program, the actual value being stored will be within the range of the variable. Where the compiler cannot verify the legality of an assignment, it will generate one or more legality assertions concerning the range of the actual value.
>
> Occurrences of **any** as an actual parameter in the type of the variable may correspond to occurrences of any value in the type of the right side. Thus, a *T*(*red*) may be assigned to a *T*(**any**), but not the reverse.

In a binding (see 7.4), the type T_v of the variable must be the same as the type T_i of the identifier. If the binding is part of a procedure or function call, however, actual parameters in the specification of T_i may be other formal parameters of the procedure or function (see 10.).

The following table summarizes the transitions which are possible:

To (formal or left side) / From (actual or right side)	*T*(red)	*T*(**any**)
T(red)	bind assign	assign
T(**any**)	discriminate	bind assign

6.5 Explicit type conversions

In recognition of the fact that controlled breaches of the type system are sometimes necessary, Euclid provides a mechanism for specifying such breaches. It takes the form of a class of pseudo-functions called type-converters. A type-converter is declared by a converter declaration, which specifies an explicit conversion from one type (the source) to another (the target). The two types must have the same size. The function takes a value of the source type as its single argument, and produces a value of the target type. No code is generated by the function, except perhaps for code supplied by the implementation to bring the representations of the argument and result into the standard form specified in this report.

It is possible to specify **procedure** or **function** as the source type, so that a program can get hold of the starting address for a routine in order to link to it from a machine code body (see 10.)

If either source or target type has an implementation-dependent representation, the converter declaration can only appear in a machine-dependent module.

converterDeclaration ::= **converter** functionIdentifier "(" sourceType ")" **returns** targetType
targetType ::= typeIdentifier
sourceType ::= typeIdentifier | **procedure** | **function**

Examples (from Appendix B):

converter *MakeCellPtr* (*AddressType*) **returns** *CellPtr*
converter *CellPtrToSBPtr* (*CellPtr*) **returns** *SBPtr*

7. Constants and variables

"Declare, if thou hast understanding."
Job 38, 4

A constant is a literal constant, or an identifier declared as a constant, or an expression whose operands and actual parameters are all constants.

constant ::= expression

A constant declaration consists of an identifier denoting the new constant, followed optionally by its type, and then by an expression that defines its value. The defining expression is evaluated, and its value becomes the value of the constant, which can never change thereafter. The type of the constant, if specified, must be assignable, and assignment-compatible with the type of the defining expression (see 6.4); otherwise its type is taken from the expression.

The defining expressions for constants are evaluated when

a scope is entered, or

a record or module type is defined.

A structured constant may be used to define a constant of a record or array type. The constants within the parentheses are the values of the components of the structured value. For a record, the order is the order in which the components appear in the definition; note that the tag of a variant is not a component and hence may not appear in a structured constant. For an array a, the order is a(*a*.IndexType.first) to a(*a*.IndexType.last). If the structured type contains other structured types as components, their values are in turn represented as nested structured constants. All the simple constants appearing in a structured constant must be manifest.

```
constantDeclaration  ::= const identifierList [ ":" type ] ":=" expression |
                         const identifierList ":" type ":=" structuredConstant
identifierList       ::= identifier { "," identifier }
structuredConstant   ::= "(" [ constantItem { "," constantItem } ] ")"
constantItem         ::= manifestConstant | structuredConstant
```

A variable declaration consists of a list of identifiers denoting the new variables, followed by their type and optional initialization, or it consists of a binding. The initialization is exactly equivalent to an assignment statement executed immediately after the declaration of which the variable declaration is a part. A **bind** declaration specifies that each of the identifiers in the bindList is to be bound to an already existing variable, rather than to a newly created one (see 7.4). If the variable binding condition is **readonly**, or omitted, then the newly declared identifier cannot be changed within the new scope. In particular, it cannot be assigned to, or passed as a variable parameter, and a procedure which imports it **var** cannot be called; the same restrictions apply to any variable that is part of it. Note that all the renamings of components of a single entire variable must be accomplished within a single bindList, since otherwise the no-overlap rule (see 7.3) would be violated.

The fixed address, if present, specifies the absolute address in memory where the variable is to be allocated. It is the compiler's responsibility to ensure that a variable allocated at a fixed address does not overlap any other variable. A *fixed-address component* is either a variable declared at a fixed address, or a module type containing a fixed-address component. Such a component must have a manifest type and may only appear as a component of a machine-dependent module type. Note that because of the no-overlap rule, only one variable of each type with a fixed-address component may be declared.

Note that a variable may not be declared to be of type Integer, but only of some subrange type. Constants may be of type Integer, however.

```
variableDeclaration   ::= var variableDeclarer |
                          bind variableBinding | bind "(" bindList ")"
bindList              ::= variableBinding { "," variableBinding }
variableBinding       ::= varBindingCondition identifier to variable
varBindingCondition   ::= readonly | var | empty
variableDeclarer      ::= identifierList [ fixedAddress ] ":" type [ initialization ]
fixedAddress          ::= "(" at manifestConstant ")"
initialization        ::= ":=" expression
```

Examples:

```
const iC, jC := -1 {iC and jC will be Integers and have the value -1}
const tc: Color := red
var k, l: -5 .. 5 := iC {both variables initially have the value of iC}
var sensitivity: array Device of Classification
bind var arrayEntry to a0(1) {a0(1) must be a valid reference. arrayEntry is simply another
  name for a0(1) over the scope of this declaration}
bind input to UserInput {input is UserInput for the scope of this declaration, but cannot be
  changed within the scope}
var a, b: SignedInt := iC {a and b initially have the value -1}
var cv: Color
const tenN := 10*n
const hue1 := Hue(red, blue)
const diskIdle: InterruptWord := (0, 1, InterruptWord.finishedOk, nullCommand)
var diskControl (at 104#8): InterruptWord := diskIdle
var dateTable: array 1 .. 10-iC of Date
const index: array -1 .. 9 of UnsignedInt := (3,1,4,1,5,9,2,6,5,3,6)
var str: String(10)
var shades: array Color of Hue
var jimH, butler, ralph, jimM, gerald: FamilyMember (male)
var Smiths, Joneses: Family
var i, j, x, y, z: SignedInt
var p, q: Boolean := false
var strP1, strP2: StringRef
var real2, real2: Real.Value
var p1, p2: ↑members
var country: (NotKnown, UnitedStates, Canada, GreatBritain, Other)
var operator: (plus, minus, times)
var col: Color
var anArray: array OneToOneHundred of SignedInt
```

Denotations of variables designate an entire variable, or a component of a variable, or a variable referenced by a pointer (see 6.2.6). Variables or constants occurring in examples below are assumed to be declared as indicated above.

Associated with every variable is a *main variable* which is entire; the variable is said to be *part* of its main variable. One variable is part of another if, roughly, an assignment to either can change the value of the other, and the space of possible values of the first variable is a (not necessarily proper) subset of the space of possible values of the second variable. The following sections define main variables and part precisely. "Part of" is a transitive relation: if x is part of y and y is part of z then x is part of z. It is also reflexive: x is part of x. Two variables are the *same* if and only if each is part of the other. Two variables *overlap* if and only if one is part of the other.

```
variable              ::= entireVariable | componentVariable
```

7.1. Entire variables

A variable identifier denotes an *entire variable*, which is its own main variable. An entire variable is never part of another entire variable known in the same scope (see 7.4). Hence, two entire variables which are simultaneously known never overlap.

entireVariable ::= variableIdentifier

7.2. Component variables

A component of a variable is denoted by the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

componentVariable ::= indexedVariable | fieldDesignator | referencedVariable
baseVariable ::= variable | functionDesignator

Corresponding to each kind of component variable described below except a referenced variable, there is a corresponding constant expression that differs from the component variable in only one way: a constant array, record, or module appears in place of the base variable.

7.2.1. Indexed variables

A component of an array variable is denoted by the variable followed by an index expression. The main variable of an indexed variable is the main variable of the array variable. The indexed variable is part of the array variable. An indexed variable i_1 is part of another indexed variable i_2 if and only if either they have the same array variable and the two indexes are equal, or the array variable of i_1 is part of i_2.

indexedVariable ::= arrayVariable "(" expression ")"
arrayVariable ::= baseVariable

The type of the index expression and the index type declared in the definition of the array type must be subranges of the same type. The value of the index expression must be a value of the index type for the program to be legal.

Examples:

index(−1)
dateTable((i **mod** 10 − *iC*) + 1)
shades(*green*)

7.2.2. Field designators

A component of a record or module variable, or a formal parameter of the type of any variable, is denoted by the variable followed by the field identifier of the component or parameter. The field identifier of a module component must be exported in the type definition. If the component is a constant or type, the record or module type may be used in place of a variable. A field designator is a variable only if the field identifier was declared as a variable; otherwise it is a constant. A variable field designator is readonly if the field identifier was exported as readonly. If a field designator is a variable, its main variable is the main variable of the containing variable, and the field designator is part of the containing variable. A field designator f_1 is part of another field designator f_2 if and only if either their containing variables are the same and their field identifiers are identical, or f_1's containing variable is part of f_2.

fieldDesignator ::= containingVariable "." fieldIdentifier
containingVariable ::= baseVariable

Examples:

str.length
jimM.sex
diskIdle.command
real1.mantissa

7.2.3. Referenced variables

If p is a pointer variable whose collection C has ObjectType T, p denotes that variable and its pointer value, whereas $p\uparrow$ is short for $C(p)$, which denotes the variable of type T referenced by p. In a referenced variable, the pointer is said to be *dereferenced.* The main variable of a referenced variable is the main variable of the collection to which the variable belongs. The referenced variable is part of the collection variable. A referenced variable r_1 is part of another referenced variable r_2 if and only if either they have the same collection and the two pointers are equal, or the collection of r_1 is part of r_2.

```
referencedVariable  ::= collectionVariable "(" pointer ")" | pointer "↑"
collectionVariable  ::= baseVariable
pointer             ::= variable | functionDesignator
```

Examples:

Smiths↑
Smiths↑.mother
strP1↑.text(1)

7.3 Scope rules and importing

A *scope* is a region of text in which an identifier (other than a field identifier) is known with a single meaning. A scope is either

a type, prototype, or routine declaration, beginning with the **type**, **procedure**, or **function** and ending at the end of the declaration, or

a region of the program between the end of a declaration and the next unmatched **end**, or

a record or module body, bracketed by **record**, or **module** and the imports and exports clauses, and the matching **end**, or

a routine body (see 10.)

A module or routine body is called a *closed* scope; other scopes are *open.* Note that a closed scope is nested within the open scope of the surrounding declaration.

An identifier is *accessible* in a scope S immediately nested in a scope V if it is

declared in S, or

declared pervasive in some enclosing scope (see 6.2.3 for syntax), or

explicitly imported into S (note that S must be closed), or

a formal parameter of V (which must be a prototype or routine declaration), or

accessible in V and S is open.

An identifier must be accessible in S to be used in S outside of an assertion, in any context except as a field identifier; the scope rules for field identifiers are given in 6.2.2 and 6.2.3. An identifier used in S and not declared in S is said to be *free* in S. An identifier may not be declared pervasive if it is a variable or if its definition imports anything explicitly. Note that a pervasive identifier is accessible in *every* scope contained in the scope of its declaration, no matter how deeply.

An identifier is *known* in S if it is

accessible in S, or

known in V if S is open, or

implicitly imported into S (see below).

An identifier must be known in S to be used in S within an assertion, in any context except as a field identifier.

Note that a pointer cannot be dereferenced within a given scope unless its collection is accessible in that scope, and cannot be dereferenced to a variable unless the collection is accessible as a variable in that scope; these rules are identical to the rules for indexed variables.

Note that there is only one circumstance in which a declaration in Euclid can have any side effects: a declaration of a module *variable* may have side effects from the execution of the module's initial action, if the module imports any variables.

7.3.1. Declaring new identifiers

New identifiers are declared

as record or module components,

as enumerated value identifiers,

in a declaration at the head of an executable scope,

as parameters of a for or discriminating case, or

as formal parameters of a routine or prototype declaration, or result identifier of a function declaration.

These new identifiers are accessible within the newly established scope. They are not accessible outside of this scope, except that:

Field identifiers of records, or of modules if exported, are accessible outside the scope as the field identifier in a suitable field designator, which is considered to be a continuation of that scope.

If an enumerated type is the definition in a declaration of type *T*, its value identifiers are declared in the same scope as *T*, rather than in the inner scope of the type declaration.

Note that the name declared by a routine declaration is *not* declared in the closed scope which is the definition, but in the enclosing scope; it must be imported explicitly into the definition if it is recursive. Even if it is declared pervasive, it must still be imported explicitly into the scope of its own definition; this requirement makes the presence of recursion obvious. On the other hand, any formal parameters of the declaration are accessible in the closed scope. Note also that importing of a module type into its definition is forbidden by the rule which restricts the use of an incompletely defined type identifier to the object type of a collection; a pointer type to a collection of such modules could be imported.

A new identifier may not be declared if it is similar (see 3.1) to any other identifier known in the scope. Of course, an identifier known in the enclosing scope of a closed scope, but not imported or pervasive, is not known, and hence may be redeclared. This is the *only* way in which an identifier can become unknown in an inner scope. This and the restriction on formal parameters of types (see 6.3) are the only restrictions on redeclaration.

7.3.2 Implicit importing

Identifiers can also be implicitly imported into a closed scope S. If *X* is a routine, module type, or prototype explicitly imported into S, then identifiers imported (explicitly or implicitly) into *X* or used in the type definitions of the formal parameter list of *X*, are implicitly imported into S. Furthermore, any identifier used in a formal parameter list is implicitly imported into the closed routine or module body which follows. The set of implicitly imported identifiers which are not explicitly imported is computed automatically, and printed by the compiler in its listing of the program, in the form of **thus** clauses appended to the explicit import lists (see 6.2.3.2 for syntax). Each import list has a **thus** clause which lists the identifiers implicitly imported as a result of the explicitly imported identifiers in that list, unless they have appeared in a previous **thus** clause, or are explicitly imported. Any **thus** clause in the source program is ignored.

Implicitly imported identifiers make it possible to write assertions (especially legality assertions) that refer to values not needed by the executable program. Note that identifiers that are only imported implicitly may only be used in assertions. Note that because of the implicit import rule, and the corresponding rule for exported routines and types (see 6.2.3), the type of any formal parameter or component of *X* can be expressed in S using the same sequence of basic symbols that appeared in its declaration; this allows legality assertions involving such types to be expressed in S. Note also that it is not possible for S to cause any variable to be referenced or modified unless the variable is either declared in S or imported into S.

7.3.3 Explicit importing

A non-pervasive identifier accessible outside a closed scope must be explicitly imported to be accessible within the scope. If an enumerated type is explicitly imported, all its enumerated value identifiers are automatically explicitly imported. An explicitly imported identifier has the same status as a newly declared one. The import clause can specify (in the binding condition) for each variable identifier whether it is imported as an ordinary variable, or readonly (the default). A readonly variable may not be changed explicitly; i.e., it may not be assigned to, or bound to a variable. A readonly variable is not a constant, however, since its value may change as a result of statements executed in a module in an enclosing scope where it is not readonly. A **readonly** variable identifier may not be imported as a **var**, and a variable identifier may not be imported as a constant. A constant identifier is always imported as a constant; its binding condition may be **const** or may be omitted.

Note that a closed scope has the property that all its possible interactions with enclosing scopes can be determined by examining its import list, identifiers declared pervasive in some enclosing scope, its parameters, and, in the case of a module, its export list. In the case of a routine no export list is needed, since nothing is left after the routine returns.

7.4 Binding

An identifier may be *bound* to a variable when it appears

> as a **var** or **readonly** formal parameter in a procedure declaration (functions cannot have such parameters);
>
> in a variable binding in a variable declaration.

A variable to which an identifier is bound is said to be *named.*

The scope of a binding is the scope of the declaration, and within this scope the identifier denotes the variable. That is, the initial value of the identifier is the value of the named variable at the time of binding, and the last value assigned to the identifier will be the value of the named variable after control finally leaves the scope. If this variable is part of an array variable, its index is evaluated when the scope is entered; if it is part of a referenced variable, the pointer is evaluated when the scope is entered.

The type of the identifier being bound must be the same as the type of the named variable to which it is bound. A **var** identifier may not be bound to a **readonly** variable. A component of a packed structure or a machine-dependent record must not appear as a named variable. Note that this does not prevent discrimination of a packed variant record (see 9.2.2.2), since in that case it is the entire record which is named, not a component.

Any variable bound (**readonly** or **var**) to an identifier known in a scope S is considered to be named in S.

To simplify the description of the rules for naming variables, we will assume for the rest of this section that a procedure does not import any variables; the initial and final actions of a module are considered to be parameterless procedures for this purpose. Any procedure which does import variables is to be rewritten with additional variable formal parameters, as described in 6.2.3. The rewritten program will behave exactly like the original one. In order to ensure that the rewritten program is a legal one, however, we impose the following requirement: any variable imported by a procedure must be known in every scope that contains a call of the procedure, if the necessary field identifiers are exported.

The language ensures that an entire variable can never overlap (see 7.1) *any* other variable known in the same scope that has a different main variable, or in other words that

> the value of an entire variable can change only
>
> > as the result of assignment to that variable or one of its parts, or
> >
> > as a result of a procedure call in which that variable was the main variable of an actual parameter corresponding to a variable formal parameter;
>
> an assignment to an entire variable can never change the value of any variable that is known in the scope containing the assignment, except one of its own parts.

To prevent binding from destroying this non-overlap property, the following restriction is imposed: no two variables which are named by a scope can overlap. If the compiler cannot determine whether or not two variables overlap (e.g., $a(i)$ and $a(j)$ overlap iff $i=j$), it will assume that they don't, and generate a legality assertion to that effect for the verifier to deal with. Note that variable identifiers that it is illegal to use in a scope because of this rule are still known, and hence are not eligible for redeclaration. An identifier is known *everywhere* in its scope, except inside nested closed scopes that do not import it (explicitly or implicitly).

Note that in general, identifiers that are declared as constants cannot cause any aliasing problems, since their values can always be copied. Of course the compiler is free to use a pointer rather than copy a value if it can determine that the meaning of the program is the same; this will certainly be true if the variable involved does not overlap any variable accessible in the same scope. In other cases the value must be copied.

8. Expressions

"Grant me some wild expressions, Heavens, or I shall burst."
Farquhar, *The Constant Couple*, V, iii

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands (i.e., variables and constants), operators, and functions.

The rules of composition specify operator *precedences* according to eight classes of operators. Unary minus has the highest precedence, then the multiplying operators, then the adding operators, then the relational operators, then **not**, then **and**, then **or** and finally, with the lowest precedence, $->$. Sequences of operators of the same precedence are associated from left to right.

Since functions cannot have side effects, the order of evaluation of operands in an expression need not be defined.

The rules of precedence are expressed by the following syntax:

```
factor       ::= variable | literalConstant | constantIdentifier | functionDesignator | set |
                 "(" expression ")" | "--" factor
set          ::= setTypeIdentifier "(" elementList ")"
elementList  ::= element { "," element } | empty
element      ::= expression | simpleType | all
term         ::= factor | term multiplyingOperator factor
sum          ::= term | sum addingOperator term
relation     ::= sum | sum relationalOperator sum | sum [ not ] in simpleType
negation     ::= relation | not relation
conjunction  ::= negation | conjunction and negation
disjunction  ::= conjunction | disjunction or conjunction
expression   ::= disjunction | disjunction "->" disjunction
```

Expressions in an element list for a set of type T = **set of** U must all be of type U. $T()$ denotes the empty set of type T, $T(U)$ denotes the set containing all the elements of U, and $T(x..y)$ denotes the set of all values in the interval $x..y$. If V is a subrange of U, $T(V)$ is an abbreviation for $T(V.\text{first}..V.\text{last})$.

Note that the operators on sets, summarized in 6.2.5, can be used to perform bitwise logical operations, and in fact these operators are intended to be implemented with the machine's logical operations on words.

Examples:

Factors:
```
x
15
(x+y+z)
Abs(x+y)
Hue(blue, col, green)
Hue(Color)
SymSet(1, 5, -4 .. -1, 2)
-x
```

Terms:
```
x*y
i div (1-i)
x mod (5*y)
```

Sums:
```
x+y
huel xor shades(red)
i*j+1
huel - Hue(blue)
```

Relations:	*x* = 15
	x **not**= 15
	p <= *q*
	(*i*<*j*) = (*j*<*k*)
	cv **in** *huel*
	cv **not in** *shades*(*orange*)
	i **in** *OneToOneHundred*
	i **not in** 25 .. (x*5)
Negations:	**not** (*p* **not**= *q*)
	not *q*
Conjunctions:	*x*<=*y* **and** *y*<*z*
	p **and not** *q*
Disjunctions:	*p* **or** (*x*>*y*)
Expressions:	false --> *p* **or** (*x* > *y*)
	*a***a* > *b***b* --> (Abs(*a*) > Abs(*b*))

8.1. Operators

The types of the operands must be the same as the types specified below, or subranges of those types. A consequence is that a value whose type is exported from a module cannot be an operand in an expression outside the module, except perhaps of the "=" operator. Where the type of the result is specified below to be Integer, it is never a subrange, regardless of the operand types. Thus first, last and size are not defined on arithmetic expressions.

It is the responsibility of the compiler to ensure that all legal arithmetic expressions evaluate to their mathematically correct values. The compiler may generate legality assertions restricting the values of operands in expressions, but it must support the evaluation of all *well-behaved* expressions, i.e., expressions whose Integer operations are all well-behaved. An operation is well-behaved if its operands and result are all in the range SignedInt. In addition, certain combinations of operators and *extended range* operands are considered well-behaved, as specified in the descriptions of individual operators.

An extended range operand is

a literal constant that is in UnsignedInt, but not in SignedInt,

a constant, variable or function designator whose type is a subrange of UnsignedInt but not of SignedInt,

a sum or difference with at least one extended range operand.

8.1.1. Multiplying operators

multiplyingOperator ::= "*" | **div** | **mod**

operator	operation	type of operands	type of result
*	multiplication set intersection	Integer the same set type *T*	Integer *T*
div	division with truncation	Integer	Integer
mod	modulus	Integer	Integer

The **div** operator truncates toward zero, so that --(*a* **div** *b*)= --*a* **div** *b*. Also, *a* **div** --*b*= --*a* **div** *b*. The **mod** operator is defined by *a* **mod** *b*=*a*--((*a* **div** *b*)**b*). The right operand of **div** or **mod** must be non-zero.

8.1.2. Adding operators

addingOperator ::= "+" | "−" | **xor**

operator	operation	type of operands	type of result
+	addition set union	Integer the same set type *T*	Integer *T*
−	subtraction set difference	Integer the same set type *T*	Integer *T*
xor	symmetric difference	the same set type *T*	*T*

When used as an operator with one Integer operand only, − denotes sign inversion.

If + or − has an extended range operand, the operation is well-behaved if and only if the result is in the range UnsignedInt.

8.1.3. Relational operators

relationalOperator ::= "=" | **not** "=" | "<" | "<=" | ">" | ">="

operator	type of operands	type of result
=, **not** =	most types	Boolean
<, >	any enumerated or subrange type	Boolean
<=, >=	any enumerated, subrange or set type	Boolean
in, **not in**	any enumerated or subrange type and a set type with a compatible base type, respectively	Boolean
in, **not in**	any enumerated or subrange type and an index type (not value), respectively.	Boolean

Both operands of the first six operators must be (subranges of) the same type.

The operators <= and >= stand for less than or equal, and greater than or equal respectively. They may also be used for comparing values of set type, and then denote set inclusion. If *p* and *q* are Boolean expressions, *p*=*q* denotes their equivalence. Note that all enumerated types define ordered sets of values.

If <, <=, >, or >= has an extended range operand, the operation is well-behaved if and only if the value of the other operand is non-negative.

8.1.4 Other operators

operator	operation	type of operands	type of result
not	logical negation	Boolean	Boolean
and	logical "and"	Boolean	Boolean
or	logical "or"	Boolean	Boolean
−>	logical implication	Boolean	Boolean

The right operand of **and** or −> need not be legal if the left operand is false; the right operand of **or** need not be legal if the left operand is true.

8.2. Function designators

A function designator specifies the evaluation of a function. It consists of the identifier or field designator designating the function, and a list of actual parameters. The parameters are expressions, and their values are substituted for the corresponding formal parameters. The type of the function designator is the type of the function's result identifier (see 9.1.2, 10., and 11.).

functionDesignator :: = function ["(" expression { "," expression } ")"]
function :: = [containingVariable "."] functionIdentifier

Examples:

FindMax(*Index*)
Gcd(147, *k*)
Power(*Index*(*i*), *str.length*)
Real.Add(*real1*, *Real.Number*(314159, 1))

9. Statements

"The statements was interesting, but tough."
Huckleberry Finn, Ch. 17

Statements denote algorithmic actions, and are said to be *executable.*

statement ::= simpleStatement | structuredStatement

9.1. Simple statements

A simple statement is a statement of which no part constitutes another statement. The empty statement consists of no symbols and denotes no action.

simpleStatement ::= assignmentStatement | procedureStatement | escapeStatement | assertStatement | emptyStatement
emptyStatement ::= empty

9.1.1. Assignment statements

The assignment statement serves to replace the current value of a variable with a new value specified by an expression.

assignmentStatement ::= variable ":=" expression

The variable and the expression must be of the same type, with the exceptions permitted by 6.4.

Note that assignment is not allowed if the type *T* of the variable is not assignable (see 6.), or if the variable is readonly. Except when pointers to counted or checkable collections are involved (see 6.2.6), assignment need not require any more work than copying the bits of the representation.

Examples:

```
x := y+z
p := i in 1..99
p := (1<=i) and (i<100)
shades(blue) := Hue(blue, Color.Succ(c))
real1 := real2
```

9.1.2. Procedure statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of *actual parameters* which are assigned or bound to the corresponding *formal parameters* declared in the procedure declaration (cf. 10.). The correspondence is established by the positions of the parameters in the lists of actual and formal parameters, respectively. There are two kinds of parameters: *constant* parameters and *variable* parameters; routine and type parameters are not permitted.

In the case of a *constant parameter*, the actual parameter must be an expression (of which a variable is a special case). The corresponding formal parameter denotes a local constant of the called procedure, and the current value of the expression is the value of this constant. As in the case of a constant declaration, the type of the actual parameter must be assignable, and assignment-compatible with the type of the formal (see 6.4).

In the case of a *variable parameter*, the actual parameter must be a variable, and the corresponding formal parameter is bound to this actual variable (see 7.4) during the entire execution of the procedure. The types must be the same. A variable parameter must be used whenever the parameter represents a result of the procedure.

```
procedureStatement  ::= procedure [ "(" expression { "," expression } ")" ]
procedure           ::= [containingVariable "."] procedureIdentifier
```

Examples:

```
TreeSort(DA)
ZeroArray(DA)
Replace(str, i, 3, '***')
```

9.1.3 Escape statements

An escape statement serves to indicate that further processing should continue at the statement following the smallest enclosing repetitive statement (**exit**), or that control should return immediately from the routine currently being executed (**return**). An **exit** must be within the scope of a repetitive statement that in turn is within the smallest closed scope containing the **exit**; hence control cannot leave a routine body through the execution of an **exit**. A **return**, on the other hand, may cause control to leave any number of open scopes that contain the **return** and are contained in the smallest routine body containing the **return**. Note that any module variables that are destroyed as a result of an escape will have their final actions executed before the escape (see 6.2.3). The **when** clause, if present, makes execution of the escape conditional. Thus, the statement

```
S when B
```

is equivalent to

```
if B then S end if.
```

An expression must not appear in a **return** statement unless the statement is in a function body, and in that case the type of the expression must be assignment-compatible with the type of the function's result value.

```
escapeStatement  ::= escapeBody [ when expression ]
escapeBody       ::= exit | return | return "(" expression ")"
```

Example:

```
begin
var flag : (a, b, finished) := finished
for ... loop
    ...
    flag := a; exit
    ...
    flag := b; exit
    ...
    end loop
case flag of
    a => ... end a
    b => ... end b
    finished => ... end finished
    end case
end
```

9.1.4 Assert statements

An assert statement introduces an assertion (see 6.2.3) that is supposed to hold whenever control reaches that point in the program. The compiler treats it as a comment, as it does the assertions supplied by invariant, pre and post clauses, unless the assertion is a Boolean expression, and the checked option is enabled for the enclosing scope (9.2.1), in which case the Boolean expression is evaluated, and execution of the program is terminated if it is false.

assertStatement ::= **assert** assertion

Examples:

assert (x<y **and** y<z)
assert {z*(w**i) = x**y}

9.2. Structured statements

Structured statements are constructs composed of other statements, which are to be executed either in sequence (compound statement and block), conditionally (conditional statements), or repeatedly (repetitive statements).

structuredStatement ::= compoundStatement | **block** |
conditionalStatement | repetitiveStatement

9.2.1. Compound statements and blocks

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. Note that a compound statement is a statement, and has no brackets; hence a sequence of statements can be written wherever a single statement can be written.

compoundStatement ::= statement { ";" statement }

Example: $z := x$; $x := y$; $y := z$

A block is a compound statement within which new identifiers can be introduced. The symbols **begin** and **end** act as brackets to delimit the scope of the new identifiers. If a scope S starts with **checked**, checking is enabled for that scope; if S starts with **not checked**, checking is disabled for S; otherwise checking is inherited from the enclosing scope (see also 6.2.3). If checking is enabled, each legality assertion in S, and each assertion in the source text of S that is a Boolean expression, is compiled into a runtime check, which aborts execution of the program if the assertion is false.

block := **begin** executableScope **end**
executableScope ::= checkedClause [declaration ";"] statement

Example:

```
begin
const twoX := 2*x
var w: SignedInt
w := twoX*twoX-x
      begin
      bind y to w;
      y := twoX*twoX*twoX+y        {equivalent to w := twoX*twoX*twoX+w}
      end
end
```

9.2.2. Conditional statements

A conditional statement selects for execution a single one of its component statements.

conditionalStatement ::= ifStatement | caseStatement

9.2.2.1. If statements

The if statement specifies that a statement is to be executed only if a certain condition (Boolean expression) is true. If it is false, the statement following the symbol **else** is to be executed, if present.

The statement

if *a* **then** S **elseif** ... **end if**

is an abbreviation for

if *a* **then** S **else if** ... **end if end if.**

ifStatement	::=	**if** expression **then** executableScope elseifClause [**else** executableScope] **end if**
elseifClause	::=	{ **elseif** expression **then** executableScope }

The expression between the symbols **if** or **elseif** and **then** must be of type Boolean.

Examples:

if *x*<15 **then** *z* := *x*+*y*; *cv* := *blue* **else** *cv* := *red*; *z* := 0 **end if**

if *p*1 **not**= *members*.nil **then** *p1* := *p1*↑.*relations*; *p2* := *members*.nil **end if**

if *str*(1) = $$$ **then** *country* := *UnitedStates*
elseif *str*(1) = $# **then** *country* := *GreatBritain*
else *country* := *NotKnown*
end if

9.2.2.2. Case statements

The case statement consists of an expression (the *selector*) and a list of elements, each labelled by a set of manifest constants of the type of the selector. It specifies that the one element is to be executed whose label contains the current value of the selector. A special label **otherwise** can be used to label a statement that should be executed if none of the other labels contains the current value of the selector. If none of the labels contains the selector, and there is no **otherwise**, the program is illegal. Each element, except the **otherwise** element, must be terminated with **end** followed by one of the constants in its label.

If the selector is discriminating an object, the parameter bound to the object is automatically declared in each case list element, either as a constant whose value is the expression in the object, or as a variable bound to the variable in the object. The expression or variable in the object must be a variant record *r*, say of type *T*. The value of *r*.itsTag is used to select one of the case list elements; in this situation, each case label list of the discriminating case statement must correspond to exactly one variant of the record. Within the element selected by a particular value of the tag, say *red*, the parameter has the type *T*(*red*). Thus with the type declaration

```
type T(tag: Color) = record
    ...
    case tag of
        red =>...
        green =>...
        ...
        end case
    end T
```

the program

```
var anyx: T(any); ...;
```

```
case x := anyx of
    red =>...
    green =>...
    end case
```

is equivalent to

```
var anyx: T(any); ...;
case anyx.itsTag of
    red => const x: T(red) := anyx; ...
    green => const x: T(green) := anyx; ...
    ...
    end case
```

except that the constant declarations in the latter would not be legal, because it is illegal to assign a *T*(any) to a *T*(*red*).

```
caseStatement       ::= simpleCase | discriminatingCase
simpleCase          ::= case expression caseTail
discriminatingCase  ::= case object caseTail
caseTail            ::= of caseBody end case
caseBody            ::= caseListElement { ";" caseListElement } otherwiseElement ";"
caseListElement     ::= caseLabelList "=>" executableScope end caseLabelEnd | empty
otherwiseElement    ::= ";" otherwise "=>" executableScope | empty
object              ::= [ const ] parameter ":=" expression |
                        varBindingCondition parameter bound to variable
parameter           ::= identifier
```

Examples:

```
case operator of
    plus =>         x := x+y end plus
    minus =>        x := x-y end minus
    times =>        x := x*y end times
    end case

case i of
    1 => cv := red end 1
    2 => cv := blue end 2
    3,8 => cv := green end 3
    4..6,9,10 => cv := yellow end 4
    otherwise => cv := purple
    end case

case var s bound to anyStream↑ of {begin new line}
    display => s.height := s.height+1 end display
    tape, disk =>
        s.position := s.position+1
        s.buffer(s.position) := $$N
        end tape
    keyboard => end keyboard {don't send characters to input device}
    otherwise => {also null}
    end case
```

9.2.3. Repetitive statements

Repetitive statements specify that certain statements are to be executed repeatedly. If a bound on the number of repetitions is known before the repetitions are started, or if the repetitions are controlled by a generator, the for statement is the appropriate construct; otherwise the loop statement should be used.

A repetitive statement introduces a new scope. The statements of this scope are executed repeatedly. The declarations of the scope take effect before each execution of the statements starts, and remain in effect until the end of that execution.

repetitiveStatement ::= loopStatement | forStatement

9.2.3.1. Loop statements

The statements in the scope are executed repeatedly until control leaves the scope through an escape statement.

loopStatement ::= **loop** executableScope **end loop**

Examples:

loop; **exit when** *Color*.Ord(*tc*) = *x*; *tc* := *Color*.Succ(*tc*) **end loop**

```
loop
if Odd(i) then z := z*x end if
i := i div 2
exit when i=0
x := x*x
end loop
```

loop *k* := *i* **mod** *j*; *i* := *j*; *j* := *k*; **exit when** *j* = 0; **end loop**

9.2.3.2. For statements

The for statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a new constant identifier called the *parameter* or *controlled constant* of the for statement.

The parameter is declared as a constant in the scope. The type of the parameter is the type of the *value* component of the module type, the type of the elements of the index type, or the base type of the set.

```
forStatement    ::= for parameter generator ";" loop executableScope end loop
generator       ::= in [ containingVariable "." ] moduleTypeIdentifier |
                    [ decreasing ] in indexType | in setExpression
setExpression   ::= expression
```

A module type generator is a module type which has three components with special names: variables called *value* (of any assignable type) and *stop*: Boolean, and a procedure called *Next*. These identifiers must be exported. A for statement of the form

for *v* **in** ModuleTypeGenerator **loop** LoopBody **end loop**

is equivalent to the block

```
begin var crec: ModuleTypeGenerator
      loop exit when crec.stop
      begin const v:=crec.value; LoopBody end
      crec.Next
      end loop
end
```

The initial and final actions in the declaration of the generator module type can perform any initialization or cleanup that may be appropriate; note that the final action is executed whenever control finally leaves the for statement, whether normally or via an escape statement.

Note that if the generator module type imports no variables, the loop body and the generator are independent, and interact only through the parameter values that are passed from the generator to the body. Thus termination can be proved solely as a property of the generator.

A for statement with an index type generator, of the form

for *v* **in** *AnIndexType* **loop** LoopBody **end loop**

is equivalent to the block

```
begin var vv:AnIndexType := AnIndexType.first
      if vv <= AnIndexType.last then
            loop
            const v: = vv
            LoopBody
            exit when vv = AnIndexType.last
            vv: = AnIndexType.Succ(vv)
            end loop
      end if
end
```

If **decreasing** is present, interchange first and last, and replace Succ by Pred and <= by >= . Note that the bounds of an index type must be constants.

A for statement with a set expression generator, of the form

for *v* **in** *SetExp* **loop** LoopBody **end loop**

is equivalent to the statement

```
begin const se: = SetExp
for v in se.BaseType
      loop
      if v in se then LoopBody end if
      end loop
end
```

Examples:

```
for lm in OneToOneHundred
      loop
      if anArray(lm)>max then max := anArray(lm) end if
      end loop

for ci decreasing in Color loop Q(ci) end loop
```

9.2.4 Other uses of binding

If a record variable is to be used a number of times in field designators, it is often convenient to bind a short identifier to it (see 7.4). Note that the binding is fixed on entry to the scope.

Example:

```
begin bind d to dateTable(i+5);
if d.month = 12 then d.month := 1; d.year := d.year+1
else d.month := d.month+1
end if
end
```

is equivalent to

```
if dateTable(i+5).month = 12 then
    dateTable(i+5).month := 1
    dateTable(i+5).year := dateTable(i+5).year+1
else dateTable(i+5).month := dateTable(i+5).month+1
end if
```

and also to

```
begin
bind d to dateTable (i+5)
bind (m to d.month, y to d.year)
if m = 12 then m := 1; y := y+1 else m := m+1 end if
end
```

10. Procedure declarations

"But a name for an effect."
Cowper

A procedure declaration serves to define a part of a program, and to associate an identifier with it so that it can be activated by procedure statements; a function declaration (see 11.) plays a similar role. Collectively, procedures and functions are called *routines.*

If a routine is declared with **forward**, everything except the routine body must appear in the forward declaration, and only the identifier is repeated in the true declaration. The import list of a forward declaration may itself contain identifiers preceded by **forward**, indicating that their own forward declaration is yet to come. The thus list of an identifier declared in a forward declaration is produced by the compiler at the point where the actual declaration is encountered; it is necessary that declarations (possibly forward) of all imported identifiers (including implicitly imported identifiers) have been encountered by this point.

A machine-code routine is exactly like an ordinary routine, except that its body is a sequence of machine instructions, represented as manifest Integer constants according to an implementation-dependent convention. An implementation may define a more elaborate syntax for code bodies. Machine code routines may only appear in machine-dependent modules.

```
procedureDeclaration ::= procedureHeading "=" routineDefinition
routineDefinition    ::= importClause preAssertion postAssertion routineBody
routineBody          ::= block identifier | codeBlock identifier | forward
codeBlock            ::= code manifestConstant { ";" manifestConstant } end
```

The *procedure heading* specifies the identifier naming the procedure, and the formal parameter identifiers (if any). The parameters are either constant or variable parameters (see also 9.1.2).

The standard representation of a routine must be defined by the implementation, so that a routine can be the argument of a type converter. A linkage between a machine code routine and a Euclid routine *R* can then be made by a declaration of the form

```
converter MakeRoutineLink (procedure) returns RoutineLink
var Rlink: RoutineLink (at 100) := MakeRoutineLink(R)
```

with an appropriate jump in the machine code body to the routine address stored at 100. The type *RoutineLink* would of course have to be properly declared in the program.

If the heading is prefixed by **inline**, this is a hint to the compiler that the procedure body should be copied at each call. Such copying tends to result in faster execution, at the expense of a larger object program. The meaning of the program is not changed by the **inline** prefix. However, an **inline** routine may not have a **forward** body or import its own name (i.e., may not be recursive).

```
procedureHeading    ::= [ inline ] procedure procedureIdentifier formalParameterList
formalParameterList ::= "(" formalSection { "," formalSection } ")" | empty
formalSection       ::= pervasive bindingCondition identifier { "," identifier } ":" type
preAssertion        ::= pre assertion ";" | empty
postAssertion       ::= post assertion ";" | empty
```

A formal section without **const**, **var**, or **readonly** implies that its constituents are constants.

A type specification for a formal parameter may have actual parameters that are preceding formal parameters; thus

```
procedure f(n: 0..1000, a: array 1..n of SignedInt) ...
```

is a legal declaration. This procedure might be called as follows:

begin var *aa*: array 1..200 **of** SignedInt; ... *f*(200, *aa*); ... **end**

Furthermore, in order to reduce the proliferation of parameters that would otherwise be required, we allow the following shorthand: the type of a formal parameter may be a prototype application with some or all of the actual parameters of the type replaced by the symbol **parameter**. Each actual parameter of the prototype for which **parameter** appears is treated as though it appeared as an additional formal parameter of the procedure, and the appropriate actual parameter is supplied in every call of the procedure. Thus

type *Ta*(*n*: UnsignedInt)=**array** 1..*n* **of** SignedInt; **procedure** *f*(*a*: *Ta*(**parameter**)) ...

is also legal and is equivalent to the previous declaration of *f*, except that all the calls on *f* will be modified appropriately. The previous call would be written

... *f*(*aa*) ...

and would be modified to become

... *f*(200, *aa*) ...

The use of a procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure. Note that the identifier must be explicitly imported, even if it is pervasive.

Examples of procedure declarations:

```
type AnIndex=1..256

type DataArray(n: AnIndex) = array 1..n of SignedInt

procedure TreeSort(var a: DataArray(parameter)) =
      {This procedure is a version of Floyd's TreeSort algorithm in CACM, 7 (1964), p. 701. TreeSort
        sorts the array a in ascending order}
post {(a in Perm(a') and j in 1 .. a.n-1) --> a(j)<=a(j+1)}
      begin type Index = a.IndexType

      inline procedure Swap(i1, i2: Index) =
      imports (var a)
      post {a in Perm(a') and a(i1)=a'(i2) and a(i2)=a'(i1)}
            begin
            const t:=a(i1)
            a(i1):=a(i2); a(i2):=t
            end Swap

      procedure SiftUp(low, high: Index) =
      imports(var a)
      pre {j in 2*(low+1) .. high --> a(j)<=a(j div 2)}
      post {j in 2*low .. high --> (a(j)<=a(j div 2) and a in Perm(a'))}
        begin var son: Index:=low
            loop const father:=son
            son:=2*father
            return when son>high
            if son<high and a(son)<a(son+1) then son:=son+1 end if
            return when a(son)<=a(father)
            Swap(son, father)
            assert {j in 2*low .. son --> a(j)<=a(j div 2)}
            end loop
      end SiftUp

      for i decreasing in 1 .. (Index.last div 2)
            loop
            SiftUp(i, Index.last)
```

```
        assert {Sifted(2*i, Index.last)}
        end loop
    for i decreasing in 1 .. Index.last-1
        loop
        Swap(1, i+1)
        SiftUp(1, i)
        end loop
    end TreeSort

type DataArraySegment(m, n: AnIndex) = array m..n of SignedInt

procedure ZeroArray(var a: DataArraySegment(parameter, parameter)) =
post {i in a.m .. a.n -> a(i)=0}
    begin
    for i in a.IndexType loop a(i) := 0 end loop
    end ZeroArray

procedure Replace(var target: String(parameter),
    first, len: StringIndex, source: String(parameter)) =
pre (target.length+source.length-len <= target.length)
post {(i in 1..first-1 -> target(i) = target'(i)) and
  (i in first .. first+source.length-1 ->
    target(i)=source(i--first+1)) and
  (i in first+source.length .. target.length+source.length-len ->
    target(i) = target'(i+len-source.length)) and
  (target.length = target'.length+source.length-len)}
    begin
    const offset := source.length-len
    const tl := target.length
    if offset > 0 then
        for i decreasing in first+len .. tl loop target(i+offset) := target(i) end loop
    elseif offset < 0 then
        for i in first+len .. tl loop target(i+offset) := target(i) end loop
    end if
    target.length := tl+offset
    for i in 1 .. source.length loop tgt(first+i-1) := source(i) end loop
    end Replace
```

Mutually recursive procedures require the use of **forward** in an imports clause:

```
procedure A = imports (forward B) forward
procedure B = imports (A) thus (B) begin . . . end
procedure A = imports ( ) thus (A) begin . . . end
```

A more symmetric version declares all the mutually recursive procedures with **forward**. In the following example *A* calls *B* calls *C* calls *A*:

```
procedure A = imports (forward B) forward
procedure B = imports (forward C) forward
procedure C = imports (A) forward

procedure A = imports ( ) thus (C, A) begin . . . end
procedure B = imports ( ) thus (A, B) begin . . . end
procedure C = imports ( ) thus (B, C) begin . . . end
```

11. Function declarations

"The Form remains, the Function never dies."
Wordsworth

Function declarations serve to define parts of the program that compute values. A function is activated by the evaluation of a function designator (see 8.2), which is a constituent of an expression.

functionDeclaration ::= functionHeading "=" routineDefinition

The function heading specifies the identifier naming the function, the formal parameters of the function, and the type of the function.

functionHeading ::= [**inline**] **function** functionIdentifier formalParameterList
[**returns** resultIdentifier ":" type]

Functions may return values of any assignable type (see 9.1.1). The value of the result identifier when the function returns (i.e., after any finalization) determines the value of the function. In all other respects the result identifier is exactly like a local variable identifier. A return statement with an expression assigns the expression to the result identifier before returning. A return statement without any value is supplied automatically just before the end of the body. A machine-code function returns its value by an implementation-defined convention.

Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function. Note that the identifier must be explicitly imported, even it if is pervasive.

A function may not have variable parameters, or import anything **var** (although importing a variable **readonly** is legal); hence, a function cannot have side effects. Furthermore, a function will return the same value whenever it is called with the same actual parameters, if the values of all imported variables are unchanged.

Examples:

```
function FindMax(a: DataArraySegment(parameter, parameter))
    returns index: SignedInt =
      post {k in a.m .. a.n --> a(index) >= a(k)}
      begin
      index := a.m
      for i in a.m+1 .. a.n
            loop
            assert {k in a.m .. i-1 --> a(index) >= a(k)}
            if a(i) > a(index) then index := i end if
            end loop
      end FindMax

function Gcd(m, n: SignedInt) returns r: SignedInt =
      imports(Gcd)
      begin if n=0 then return (m) else return (Gcd(n, m mod n)) end if end Gcd

function Power(x: SignedInt, y: UnsignedInt) returns z: SignedInt =
      begin var w: SignedInt; var i: UnsignedInt
      w := x; i := y; z := 1
            loop assert {z*(w**i) = x**y}
            exit when i = 0
            if Odd(i) then z := z*w end if
            i := i div 2
            w := w*w
            end loop
```

```
        assert {z = x**y}
        end Power

    function Substr(s: String(parameter), first: StringIndex, len: StringLength)
      returns r: String(len) =
        pre (first+len <= s.length+1)
        post {(i in 1..len -> r(i) = s(i+first-1)) and r.length = len}
        begin
        r.length := len
        for i in 1..len loop r(i) := s(i+first-1) end loop
        end Substr

    function Catenate(s1: String(parameter), s2: String(parameter), size: StringLength)
      returns r: String(size) =
        pre ((s1.length + s2.length <= size) and
          (r.length = s1.length + s2.length))
        post ((i in 1..s1.length -> r(i) = s1(i)) and
          (i in 1..s2.length -> r(i+s1.length) = s2(i)))}
        begin
        for i in 1 .. s1.length loop r(i) := s1(i) end loop
        for i in 1 .. s2.length loop r(i+s1.length) := s2(i) end loop
        end Catenate
```

12. Programs

"All are but parts of one stupendous whole."
Pope, An Essay on Man

A Euclid program consists of a sequence of module type declarations, possibly prefixed by an include clause that causes additional text to be inserted into the program. The include clause is a list of items, each of which names a file containing the text of a Euclid program; the file is named by a literal string, according to an implementation-defined convention. If **from** is present, only the named module types are included; otherwise all the declarations in the file are included. If the same type identifier from the same file is included more than once, duplicates are suppressed. If different files contain types with the same name, however, an error results because of the normal Euclid rule that forbids redeclaration of names.

An implementation may use some method other than the textual substitution described above to provide this facility. In particular, it may take advantage of the fact that an included file has already been compiled. Thus the structure of compilation units is intended to facilitate separate compilation (although not to require it).

This report does not specify how the module types declared in programs are instantiated to start a program.

```
program          ::= compilationUnit
compilationUnit  ::= [ includeClause ";" ] typeDeclaration { ";" typeDeclaration }
includeClause    ::= include includeItem { ";" includeItem }
includeItem      ::= [ typeIdentifier { "," typeIdentifier } from ] fileName
fileName         ::= literalString
```

Examples:

```
include Scanner, SymbolTable from "ParserUtilities"
type Parser = module
...
end Parser

type NumberTable = module exports (Search, Delete, Insert)
    {This module implements a table of numbers, e.g., currently open accounts, as an associa-
      tive memory}

    pervasive const tableSize := 763

    pervasive type TableIndex = 1 .. tableSize

    pervasive type CyclicScan(item: SignedInt) = {a generator for a for loop}
        module exports (Next, value, stop)
        const start := (item mod tableSize)+1
        var value: TableIndex := start
        var stop: Boolean := false

        procedure Next =
        imports(var value, start, var stop)
            begin
            if value = tableSize then value := 1
            else value := value+1 end if
            stop := (value not= start)
            end Next
        end CyclicScan

    type State = (fresh, full, deleted)

    type TableEntry(flag: State) =
```

```
        record
        case flag of
                full => var key: SignedInt end full
                otherwise =>{ nothing }
                end case
        end TableEntry

var table: array TableIndex of TableEntry(any)

function Search(key: SignedInt) returns r: Boolean =
imports (table)
        begin
        for i in CyclicScan(key)
                loop
                case entry := table(i) of
                        fresh => return (false) end fresh
                        full => return (true) when entry.key = key; end full;
                        otherwise =>
                        end case;
                end loop;
        return (false);
        end Search;

procedure Delete(key: SignedInt) =
imports (var table)
        begin
        const deletedEntry: TableEntry(deleted) := ();
        for i in CyclicScan(key)
                loop
                case entry := table(i) of
                        full => if entry.key = key then
                                        table(i) := deletedEntry;
                                        return
                                        end if
                                        end full
                        fresh => return end fresh
                        otherwise =>
                        end case
                end loop
        end Delete

procedure Insert(key: SignedInt) =
imports (var table, Search)
        begin
        return when Search(key) {if already there};
        for i in CyclicScan(key)
                loop
                case table(i).flag of
                        fresh, deleted =>
                                        var t: TableEntry(full)
                                        t.key := key
                                        table(i) := t
                                        return
                                        end fresh
                        otherwise =>
                        end case
                end loop
        assert (false) {table will never be full}
        end Insert
```

```
const freshEntry: TableEntry(fresh) := ()
initially
        begin
        for i in table.IndexType loop table(i) := freshEntry end loop
        end
end NumberTable
```

13. Implementation standards

> *"'That's not a regular rule: You invented it just now.'*
> *'It's the oldest rule in the book,' said the King.*
> *'Then it ought to be Number One,' said Alice."*
>
> *Alice in Wonderland,* Ch. 12

One motivation for the development of Euclid was the need for a powerful and flexible language that could be reasonably efficiently implemented on most computers. Its features are defined without reference to any particular machine in order to facilitate the interchange of programs. To establish a reasonable minimum standard for Euclid implementations, the following requirements are imposed on every implementation.

1. Word symbols, such as **begin**, **end**, etc., may be written as a sequence of letters (without surrounding escape characters). They may not be used as identifiers. An implementation may also allow such symbols to be written in other ways (e.g., in boldface), provided there is a straightforward transformation into the representation as a sequence of letters.
2. Blanks, ends of lines, and comments are defined as *separators.* An arbitrary number of separators may occur between any two consecutive Euclid symbols, with the following restriction: no separators may occur within identifiers, numbers, and word symbols.
3. At least one separator must occur between any pair of consecutive identifiers, numbers, or word symbols.
4. The implementation may set limits on the size and complexity of the source program. However, these limits must be chosen from the following list, and must not be more restrictive than indicated below. An implementation should not reject a program for exceeding some limit not on this list; it may accept programs which exceed any of these limits.
    a) The range of UnsignedInt (must include $0..2^{16}-1$). The range of SignedInt (must include $-2^{15}+1..2^{15}-1$). It is recommended, but not required, that larger subranges of Integer than these be permitted, say up to $0..2^{32}-1$ and $-2^{31}+1..2^{31}-1$. Integer literal constants with values in any permitted subrange of Integer must be accepted.
    b) The maximum number of elements in the base type of a set (at least 16).
    c) Depth of nesting of **end**s (at least 20).
    d) Depth of nesting of parentheses in an expression (at least 7). Number of basic symbols in an expression (at least 50).
    e) The total number of identifiers known in a scope (at least 200). The total number of identifiers in a program (at least 1000).
    f) The number of non-compound statements and declaration parts in the source program (at least 2000).
    g) The maximum number of characters in an identifier (at least 50).
    h) The value of stringMaxLength (at least 255).
    i) The maximum value of alignment (may be 1).

13.1 Representation of special symbols

The preferred representations of special symbols that are not words, in the IBM PL/1 60-character set, and in the Model 33 Teletype set, are as follows:

Special symbol	*PL/1*	*Teletype*
{	(*	(*
}	*)	*)
↑	@	↑
break	_	\

Programs can be converted from one representation to another by a finite-state algorithm that recognizes each special symbol and identifier in the source representation, and outputs the corresponding symbol in the target representation. During this conversion, break characters can be supplied by any uniform algorithm. The recommended strategy for break characters is as follows:

> If neither representation has lower case, or both do, break characters should be preserved.
>
> If only the source has lower case, a break character should be inserted between a lower case letter and a following upper case letter in an identifier.
>
> If only the target has lower case, all letters should be converted to lower case, except that when a letter follows a break character, the break character should be dropped and the letter left in upper case.

Unfortunately, this algorithm removes initial capitals; analysis of the declarations is required to determine which identifiers denote types or routines and hence should be capitalized.

13.2 Standard format for programs

It is strongly recommended that an implementation include an option to produce a version of the source program in a standard format. The recommended standard is:

> One level of indentation for each unmatched **begin**, **record**, **module**, **loop**, **if**, or **case**. The bracket and its corresponding **end** should also be indented, except in the case of **if .. elseif .. else**, and case. The **for** clause should not be indented. Indentation should be omitted if the entire compound statement or declaration will fit on one line. Thus

```
a := 3; a1 := 31; a2 := 32
    begin
    b := 4
    end

c := 5

if b=4 then
    a := 6
else
    a := 7
end if

if b=4 then a := 6 else a := 7 end if

    loop
    ...
    end loop

for i in 0..5
    loop
    ...
    end loop
```

> A second level of indentation for the scope in each case element. Thus

```
case a of
 3 =>
```

```
        OutputLine(a, b, 100)
        InputLine(a, b, 100)
        end 3
    4 =>
        OutputLine(a, b, 200)
        InputLine(a, b, 200)
        end 4
    end case
```

If a statement is too long for one line, it should be continued on subsequent lines with a small amount of indentation (one or two spaces).

Several short statements may be put on the same line.

Semicolons should be omitted at the ends of lines (see 3.1).

13.3 Annotation

It is strongly recommended that an implementation include an option to produce an annotated listing of the source program, in which all identifiers automatically imported into a closed scope and the formal parameter declarations corresponding to uses of **parameter** are noted.

It is recommended that options exist to add to the annotated listing declarations for all explicitly imported identifiers, and warnings of identifiers imported but not used, or imported **var** but not set.

14. Implementation notes

"'The horror of that moment,' The King went on, 'I shall never, never forget!'
'You will, though,' the Queen said, 'if you don't make a memorandum of it.'"
Through the Looking-Glass, Ch. 1

This section discusses implementation techniques for parts of the Euclid language that are relatively new or tricky. Of course, no implementation is required to use these techniques.

14.1 Identifiers

Identifiers may vary in capitalization, and in the presence or absence of break characters. The Euclid rule is that each time an identifier is used, it must be written the same way it was declared (see 3.). This rule can be efficiently enforced by normally looking up the identifier exactly as it is written, and making the more expensive comparison that ignores break characters and capitalization only when adding an identifier to the symbol table. If a hash table is used, the hashing algorithm should probably be chosen to map equivalent identifiers into the same hash code.

An alternative implementation is to store the identifier in a standard case, with break characters removed, and to append to it the additional information needed to keep track of the case of each letter, and the presence of break characters.

14.2 Parsing

Euclid has been designed to be amenable to deterministic parsing [Aho and Johnson 74]. The syntax presented in the body of the report is not directly suitable for this purpose, since it was chosen primarily to aid the reader and facilitate the exposition.

14.3 One-pass translation

Euclid has been designed to permit one-pass translation. To this end, identifiers must be declared before they are used. Recursive routines and types may break this rule by using **forward** for the definition, but all the type information must still be present before use.

14.4 Routine parameters

Constant parameters can be passed either by copying the value, or by reference, i.e., by passing the address of a variable containing the value, unless the variable overlaps some variable accessible in the routine, in which case the parameter must be passed by copying. The same test that is required to detect the overlap of two variables can be used to detect this overlap; it depends on the definition of overlap given in section 7. Note that (explicitly and implicitly) imported variables must be treated exactly like variable parameters for this test.

Variable parameters can be passed either by passing the address of the variable, or by copying the value on entry to the routine, and copying it back on exit; the latter copying is unnecessary if it is readonly. The absence of overlap means that this double copying will always work. If a variable is passed by double copying, then a constant parameter whose value is an overlapping variable can safely be passed by reference; this might be desirable if the constant is much larger than the variable.

14.5 Routines in modules

If a routine *R* is declared in a module *M*, and *R* imports a non-manifest component *c* of *M*, then *R* must obtain access to *c* when it is called. The call must take the form *m.R*(...), where *m* is a variable or constant of type *M*. This may be done in two ways:

> By passing *m* (presumably by reference), and treating *m* as a record within *R*; *c* would then be accessed by its known position relative to the address of *m*.
>
> By passing *c* explicitly. This might be preferable if it is the only such component.

If *R* imports only manifest constants, everything can be done at compile-time. If it imports any non-manifest component *c*, however, *c* must in general be passed as a parameter, since it could be different for different module variables. There is one exception: if *R* imports only constants that depend only on constants declared in module types for which only one variable is ever created, then the references to these constants can be compiled into *R*, and they need not be passed as parameters.

14.6 Constant components of records and modules

The same observations apply in general to constant components. Except under the conditions described above, a constant component or parameter must be stored in each variable, since it may be different from one variable to another. Of course, if the component or parameter is never referenced except during initialization, then it need not be stored.

14.7 Finalization

If a scope declares a module variable that includes a finalization statement, then code must be executed whenever the scope is exited which performs the finalization. This might be done inline, or by calling a routine. The same is true whenever a Free procedure is executed to free such a variable.

Since this machinery must be present anyway, it can be used to allow variables declared in the scope, whose size is not manifest, to be allocated someplace other than in the frame of the routine containing the scope. The finalization code for the scope would then be expanded to include code for freeing the storage used by such variables. Whether this technique is worthwhile depends on the allocation strategy used for activation records.

14.8 Inline code

In general, it is highly desirable for an implementation to consider the use of inline code for all short routine bodies, even if the program has not explicitly declared them inline. It is quite common for such bodies to be shorter than their calling sequences, especially since they can be subjected to normal optimization once they have been inserted inline.

14.9 Reference counts

There is an important special case in which it is possible to avoid incrementing and decrementing reference counts. Suppose that the program has a declaration

type *C* = **counted collection of** ...

We say that a scope S is *C*-conservative if it contains no assignments to variables of type ↑*C* that are not local to S, it contains no uses of *v*.refCount for variables in *C*, and all procedures that it calls are also *C*-conservative. Within S it is not necessary to update reference counts for variables in *C*, since no variable in *C* can be freed in S, and every such variable will have the same reference count on exit from S that it had on entry to S. This idea can be extended to routines that make assignments to variable parameters of type ↑*C*.

14.10 Representation of pointers

It is possible to take advantage of the fact that pointers are strictly segregated by collection, to make the representation of a pointer depend on the collection it is in. For instance, a pointer could be relative to some base address. Of course, such a representation cannot be used in a sensitive context.

14.11 Prototypes

Suppose *T* is a prototype. It is not generally necessary to store its actual parameters with a variable whose type is an application of *T*. The two exceptions (see 6.3) are:

T has an actual parameter **any**, or

the variable is in a collection whose object type contains **unknown**.

In all other cases, the values of the parameters are known from the declaration. If the declaration contains **parameter**, it must be in a formal parameter list, and the value can be passed as an additional formal (see 10.). If the type is exported from a module, it may import components of the module, in which case the remarks of 14.5 are applicable.

These considerations are especially relevant for variant records and arrays. A variant record is normally used in one of three ways:

a) To express the uniformity of several different record structures, even though the particular structure in use is always manifest from the declarations.

b) When the variant is expected to change during execution. A variable of this kind must be declared with **any**, and the tag must be stored in the record. Furthermore, enough space must be allocated for the largest of the possible variants.

c) When the variable is dynamic, and the variant is fixed at the time the variable is created. A collection of such variables must be declared with **unknown**, and the tag must be stored with each variable, or with each pointer.

The third case, involving **unknown**, is also appropriate for arrays. For example, there might be a collection of strings of widely varying length, all of which should be treated uniformly.

When one of the bounds of an array is a parameter, the size of the resulting type is not known at compile-time; such a type is called *length-unresolved*. If more than one length-unresolved variable is declared in a record or routine, it is not possible to determine the relative position of every such variable at compile-time. This situation can be dealt with by constructing pointers to all the length-unresolved components except the first one at the time the record variable or routine instance is created, and referring to them indirectly through these pointers.

14.12 Checkable collections

One possible implementation that meets the requirements of 6.2.6 is to allocate a *Finger* record in the system zone for each variable in a checkable collection *C*:

type *Finger* = **record var** refCount: UnsignedInt; **const** *addr*: AddressType
end *Finger*

The representation of a ↑*C* is the address of the finger, and an extra level of indirection is required on every reference.

References

[1] Aho, A.V. and Johnson, S.C. LR parsing. *Computing Surveys* **6**, 2 (June 1974).

[2] Ambler, A. et al. Gypsy: A language for specification and implementation of verifiable programs. *SIGPLAN Notices* **12**, 3, pp 1-10 (March 1977).

[3] Clark, B.L. and Ham, F.J.B. *The Project SUE System Language Reference Manual.* University of Toronto, Computer Systems Research Group Technical Report CSRG-42 (Sept. 1974).

[4] Clark, B.L. and Horning, J.J. Reflections on a language designed to write an operating system. *SIGPLAN Notices* **8**, 9 (Sept. 1973).

[5] Geschke, C.M. and Mitchell, J.G. On the problem of uniform references to data structures. *IEEE Trans.* SE-1, 2, pp 207-219 (June 1975).

[6] Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica* **1**, pp 271-281 (1972).

[7] Hoare, C.A.R. Hints on programming language design. Stanford University, Computer Science Department, Technical Report STAN-CS-73-403 (Dec. 1973).

[8] Holt, R.C. et al. The Euclid language: A progress report. *Proc. ACM National Conf.*, (Dec. 1978).

[9] Holt, R. C. and Wortman, D. B. A model for implementing Euclid modules and type templates. *SIGPLAN Notices* **14**, 8, pp 8-12 (Aug. 1979).

[10] Holt, R. C. et. al. The Toronto Euclid Compiler Project Workbook. I. P. Sharp Associates, Ltd., Toronto (March 1980).

[11] Ichbiah, J.D. et al. *The System Implementation Language LIS.* CII, 68 route de Versailles, 78430 Louveciennes, France (Dec. 1974).

[12] Jensen, K. and Wirth, N. *Pascal User Manual and Report*, 2nd ed. Springer-Verlag, 1975.

[13] Liskov, B. and Zilles, S. An introduction to CLU. *SIGPLAN Notices* **9**, 4 (April 1974).

[14] Liskov, B. Abstraction mechanisms in CLU. *Comm. ACM* **20**, 8, pp 564-576 (Aug. 1977).

[15] London, R.L. et al. Proof rules for the programming language Euclid. *Acta Informatica* **10**, pp 1-26 (1978).

[16] Popek, G.J. et al. Notes on the design of Euclid. *SIGPLAN Notices* **12**, 3, pp 11-18 (March 1977).

[17] Richards, M. BCPL: A tool for compiler writing and structured programming. *Proc. AFIPS Conf.* **34**, pp 557-566 (1969 SJCC).

[18] Thompson, D.H. Base + Builder language definition. Technical Note 4, Computer Systems Research Group, University of Toronto (March 1976).

[19] Wirth, N. The programming language Pascal. *Acta Informatica* **1**, pp 35-63 (1971).

[20] Wirth, N. Modula: A language for modular multiprogramming. *Software—Practice and Experience* 7, 1, pp 3-35 (Jan. 1977).

[21] Wortman, D.B. On legality assertions in Euclid. *IEEE Trans.* SE-5, 4 (July 1979).

[22] Wortman, D.B. and Cordy, J. Early Experiences with Euclid. *Proc. 5th Int. Conf. Software Eng.* (March 1981).

[23] Wulf, W., London, R.L. and Shaw, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans.* SE-2, 8, pp 253-265 (Dec. 1976).

Appendix A. Collected syntax

The syntax of Euclid, as presented in this report, is collected below for convenient reference. The numbers in the left margin are the numbers of the sections in which the following text appears.

3.1

letter ::= "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"

octalDigit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"

digit ::= octalDigit | "8" | "9"

hexDigit ::= digit | "A" | "B" | "C" | "D" | "E" | "F"

breakChar ::= <some implementation-dependent character not a letter or digit>

specialSymbol ::= "+" | "-" | "*" | "=" | "<" | ">" | "<=" | ">=" | "->" | "(" | ")" | ".." | "{" | "}" | ":=" | "." | "," | ";" | ":" | "'" | "↑" | "=>" | "$" | "#" | wordSymbol

wordSymbol ::= **abstraction | aligned | and | any | array | assert | at | begin | bind | bits | bound | case | checkable | checked | code | collection | const | converter | counted | decreasing | default | dependent | div | else | elseif | end | exit | exports | finally | for | forward | from | function | if | imports | in | include | initially | inline | invariant | loop | machine | mod | module | not | of | or | otherwise | packed | parameter | pervasive | post | pre | procedure | readonly | record | return | returns | set | then | thus | to | type | unknown | var | when | with | xor**

4.

identifier ::= letter { letterOrDigit }

letterOrDigit ::= letter | digit | breakChar

unsignedNumber ::= digit { digit } |
octalDigit { octalDigit } "#8" |
digit { hexDigit } "#16"

literalString ::= "'" { extendedCharacter } "'"

extendedCharacter ::= character | "$" extension

extension ::= digit digit digit | "S" | "T" | "N" | "$" | "'"

literalChar ::= "$" extendedCharacter

5.

literalConstant ::= unsignedNumber | literalString | literalChar | enumeratedValueIdentifier

manifestConstant ::= literalConstant | manifestExpression

manifestExpression ::= expression

6.

type ::= simpleType | structuredType

typeDeclaration ::= **type** typeIdentifier = preAssertion typeDefinition | prototypeDeclaration

typeDefinition ::= type | **forward**

6.1.

simpleType ::= enumeratedType | standardSimpleType | subrangeType | derivedSimpleType | simpleTypeAppl

derivedSimpleType ::= [containingVariable "."] simpleTypeIdentifier

6.1.1.

enumeratedType ::= "(" enumeratedValueIdentifier { "," enumeratedValueIdentifier } ")"

6.1.2.

6.1.3.

subrangeType ::= constantSum ".." constantSum
constantSum ::= sum

6.2.

structuredType ::= [**packed**] unpackedStructuredType | derivedStructuredType | structuredTypeAppl
unpackedStructuredType ::= arrayType | recordType | moduleType | mdRecordType | setType | collectionType | pointerType
derivedStructuredType ::= [containingVariable "."] structuredTypeIdentifier

6.2.1.

arrayType ::= **array** indexType **of** componentType
indexType ::= simpleType
componentType ::= type

6.2.2.

recordType ::= **record** fieldList endRecord
endRecord ::= **end record** | **end** identifier
fieldList ::= [recordDeclaration ";"] [variantPart] ";"
recordDeclaration ::= recordDeclarationPart { ";" recordDeclarationPart }
recordDeclarationPart ::= constantDeclaration | **var** variableDeclarer

variantPart ::= **case** tag [**default** manifestConstant] **of** variant { ";" variant } [otherwiseVariant] ";" **end case**
variant ::= caseLabelList "=>" recordDeclaration ";" **end** caseLabelEnd | empty
caseLabelList ::= caseLabel { "," caseLabel }
caseLabelEnd ::= literal | manifestConstantIdentifier | "(" caseLabel ")"
caseLabel ::= manifestConstant | subrangeType
tag ::= identifier
otherwiseVariant ::= **otherwise** "=>" recordDeclaration

6.2.3

moduleType ::= [**machine dependent**] **module** [identifier] importClause exportClause moduleBody endModule
endModule ::= **end module** | **end** identifier
moduleBody ::= checkedClause declaration ";" initialAction invariant finalAction
checkedClause ::= **checked** | **not checked** | empty
declaration ::= empty | pervasive declarationPart { ";" pervasive declarationPart }
declarationPart ::= constantDeclaration | variableDeclaration | typeDeclaration | procedureDeclaration | functionDeclaration | converterDeclaration | **assert** assertion
pervasive ::= **pervasive** | empty
invariant ::= [**abstraction** functionDeclaration ";"] **invariant** assertion ";" | empty
assertion ::= "(" expression ")" | empty

6.2.3.1

```
exportClause        ::= exports exportList ";" | empty
exportList          ::= "(" exportItem { "," exportItem} ")"
exportItem          ::= bindingCondition identifier [ with exportList ] | ":=" | "=" | "↑" |
                        subrangeType
bindingCondition    ::= const | readonly | var | empty
```

6.2.3.2

```
importClause        ::= singleImportClause { ";" singleImportClause } | empty
singleImportClause  ::= imports importList [thus importList] ";"
importList          ::= "(" importItem { "," importItem } ")" | "(" ")"
importItem          ::= [ forward ] pervasive bindingCondition identifier
```

6.2.3.4

```
initialAction       ::= initially routineDefinition ";" | empty
finalAction         ::= finally routineDefinition ";" | empty
```

6.2.4

```
mdRecordType        ::= machine dependent record [ alignmentClause ]
                        [ mdDeclarationPart { ";" mdDeclarationPart } ] ";" endRecord
mdDeclarationPart   ::= constantDeclaration |
                        var identifier "(" at manifestConstant [ bits subrangeType ] ")"
                        ":" typeDefinition [ initialization ]
alignmentClause     ::= aligned mod manifestConstant
```

6.2.5.

```
setType             ::= set of baseType
baseType            ::= simpleType
```

6.2.6.

```
collectionType      ::= countControl collection of objectType [ in zoneIdentifier ]
countControl        ::= counted [ manifestConstant ] | checkable | empty
objectType          ::= type
pointerType         ::= "↑" collectionVariable
```

6.3

```
prototypeDeclaration ::= type typeIdentifier typeFormalList = preAssertion typeDefinition
typeFormalList      ::= "(" typeFormalSection { "," typeFormalSection } ")"
typeFormalSection   ::= identifier { "," identifier } ":" indexType
simpleTypeAppl      ::= derivedSimpleType typeActualList
structuredTypeAppl  ::= derivedStructuredType typeActualList
typeActualList      ::= "(" typeActualParameter { "," typeActualParameter } ")"
typeActualParameter ::= expression | any | unknown | parameter
```

6.5

```
converterDeclaration ::= converter functionIdentifier "(" sourceType ")" returns targetType
targetType          ::= typeIdentifier
sourceType          ::= typeIdentifier | procedure | function
```

7.

```
constant            ::= expression
```

```
constantDeclaration  ::= const identifierList [ ":"  type ] ":=" expression |
                         const identifierList ":" type ":=" structuredConstant
identifierList       ::= identifier { "," identifier }
structuredConstant   ::= "(" [ constantItem { "," constantItem } ] ")"
constantItem         ::= manifestConstant | structuredConstant

variableDeclaration  ::= var variableDeclarer |
                         bind variableBinding | bind "(" bindList ")"
bindList             ::= variableBinding { "," variableBinding }
variableBinding      ::= varBindingCondition identifier to variable
varBindingCondition  ::= readonly | var | empty
variableDeclarer     ::= identifierList [ fixedAddress ] ":" type [ initialization ]
fixedAddress         ::= "(" at manifestConstant ")"
initialization       ::= ":="  expression

variable             ::= entireVariable | componentVariable
```

7.1.

```
entireVariable       ::= variableIdentifier
```

7.2.

```
componentVariable    ::= indexedVariable | fieldDesignator | referencedVariable
baseVariable         ::= variable | functionDesignator
```

7.2.1.

```
indexedVariable      ::= arrayVariable "(" expression ")"
arrayVariable        ::= baseVariable
```

7.2.2.

```
fieldDesignator      ::= containingVariable "." fieldIdentifier
containingVariable   ::= baseVariable
```

7.2.3.

```
referencedVariable   ::= collectionVariable "(" pointer ")" | pointer "↑"
collectionVariable   ::= baseVariable
pointer              ::= variable | functionDesignator
```

8.

```
factor               ::= variable | literalConstant | constantIdentifier | functionDesignator | set |
                         "(" expression ")" | "--" factor
set                  ::= setTypeIdentifier "(" elementList ")"
elementList          ::= element { "," element } | empty
element              ::= expression | simpleType | all
term                 ::= factor | term multiplyingOperator factor
sum                  ::= term | sum addingOperator term
relation             ::= sum | sum relationalOperator sum | sum  [ not ] in simpleType
negation             ::= relation | not relation
conjunction          ::= negation | conjunction and negation
disjunction          ::= conjunction | disjunction or conjunction
expression           ::= disjunction | disjunction "->" disjunction
```

8.1.1.

```
multiplyingOperator ::= "*" | div | mod
```

8.1.2.

addingOperator ::= "+" | "–" | **xor**

8.1.3.

relationalOperator ::= "=" | **not** "=" | "<" | "<=" | ">" | ">="

8.2.

functionDesignator ::= function ["(" expression { "," expression } ")"]
function ::= [containingVariable "."] functionIdentifier

9.

statement ::= simpleStatement | structuredStatement

9.1.

simpleStatement ::= assignmentStatement | procedureStatement | escapeStatement | assertStatement | emptyStatement
emptyStatement ::= **empty**

9.1.1.

assignmentStatement ::= variable ":=" expression

9.1.2.

procedureStatement ::= procedure ["(" expression { "," expression } ")"]
procedure ::= [containingVariable "."] procedureIdentifier

9.1.3

escapeStatement ::= escapeBody [**when** expression]
escapeBody ::= **exit** | **return** | **return** "(" expression ")"

9.1.4

assertStatement ::= **assert** assertion

9.2.

structuredStatement ::= compoundStatement | block | conditionalStatement | repetitiveStatement

9.2.1.

compoundStatement ::= statement { ";" statement }

block := **begin** executableScope **end**
executableScope ::= checkedClause [declaration ";"] statement

9.2.2.

conditionalStatement ::= ifStatement | caseStatement

9.2.2.1.

ifStatement ::= **if** expression **then** executableScope elseifClause [**else** executableScope] **end if**
elseifClause ::= { **elseif** expression **then** executableScope }

9.2.2.2.

caseStatement ::= simpleCase | discriminatingCase
simpleCase ::= **case** expression caseTail
discriminatingCase ::= **case** object caseTail
caseTail ::= **of** caseBody **end case**
caseBody ::= caseListElement { ";" caseListElement } otherwiseElement ";"
caseListElement ::= caseLabelList "=>" executableScope **end** caseLabelEnd | **empty**
otherwiseElement ::= ";" **otherwise** "=>" executableScope | empty
object ::= [**const**] parameter ":=" expression |
varBindingCondition parameter **bound to** variable
parameter ::= identifier

9.2.3.

repetitiveStatement ::= loopStatement | forStatement

loopStatement ::= **loop** executableScope **end loop**

forStatement ::= **for** parameter generator ";" **loop** executableScope **end loop**
generator ::= **in** [containingVariable "."] moduleTypeIdentifier |
[**decreasing**] **in** indexType | **in** setExpression
setExpression ::= expression

10.

procedureDeclaration ::= procedureHeading "=" routineDefinition
routineDefinition ::= importClause preAssertion postAssertion routineBody
routineBody ::= block identifier | codeBlock identifier | **forward**
codeBlock ::= **code** manifestConstant { ";" manifestConstant } **end**

procedureHeading ::= [**inline**] **procedure** procedureIdentifier formalParameterList
formalParameterList ::= "(" formalSection { "," formalSection } ")" | empty
formalSection ::= pervasive bindingCondition identifier { "," identifier } ":" type
preAssertion ::= **pre** assertion ";" | empty
postAssertion ::= **post** assertion ";" | empty

11.

functionDeclaration ::= functionHeading "=" routineDefinition

functionHeading ::= [**inline**] **function** functionIdentifier formalParameterList
[**returns** resultIdentifier ":" type]

12.

program ::= compilationUnit
compilationUnit ::= [includeClause ";"] typeDeclaration { ";" typeDeclaration }
includeClause ::= **include** includeItem { ";" includeItem }
includeItem ::= [typeIdentifier { "," typeIdentifier } **from**] fileName
fileName ::= literalString

Appendix B: Toronto Euclid

This section defines the Toronto Euclid subset of the Euclid language. The Euclid language is defined in the body of this report. Toronto Euclid is the name given to the subset supported by the Toronto Euclid compiler released in fall 1979. Note that any language restrictions imposed by the Euclid Report apply to Toronto Euclid even though not repeated here.

This appendix consists of three parts. The first part gives a list of the notable features of Euclid which are included in Toronto Euclid. The second part lists the ways in which Toronto Euclid extends or modifies the Euclid language. The third part gives a summary of the features of full Euclid which are not included in Toronto Euclid.

Note that the Toronto Euclid compiler is designed to eventually support full Euclid and hence does not necessarily enforce the Toronto Euclid subset. Features not included in Toronto Euclid may be correctly implemented, flagged as an error, or cause the compiler or object program to abort.

B.1 Notable features of Toronto Euclid

The following is a list of notable features of the full Euclid language which are included in Toronto Euclid.

Abstraction functions: are treated as comments (as required by the Euclid Report).

assert *statements* are supported both in statement lists and in declaration lists. Assert conditions are evaluated and checked at run time in checked scopes. Manifest assert conditions are evaluated at compile time.

bind *statements* are supported. Names may be bound to arbitrarily complex subscript and/or field references. Both **var** and **readonly** binds are acccepted and enforced in Toronto Euclid.

code *blocks:* Machine code blocks as the body of a routine are supported. In Toronto Euclid, code blocks are written in Unix assembly language and must begin and end with a line consisting only of the character "?" in column 1. Toronto Euclid code blocks can access only the parameters to the routine.

Constant folding: Manifest expressions (expressions consisting of values known at compile time) are evaluated at compile time and folded to the result value. Set expressions and expressions containing the **mod** operator are not folded at compile time.

Dynamic storage allocation (collections and pointers) is supported in the default zone (*SystemZone*) only. **counted** and **checkable** collections are not allowed.

exit when is supported. A loop may be exited either unconditionally (via **exit**) or conditionally (via **exit when**).

Export lists: The **exports** clause of Euclid modules is enforced in Toronto Euclid. Only fields and components which are exported from a module may be referenced outside the module (via the . operator). Only those fields which are exported **var** may be assigned to outside the module. In Toronto Euclid, every exported field and component is implicitly exported with all of its field and components. In Toronto Euclid, every exported symbol is implicitly exported with :=, = and ↑ (if it applies). The Toronto Euclid compiler accepts but does not enforce the **with** clause of full Euclid. It fully implements **var**, **const** and **readonly** in export lists.

for *loops* are supported, including subrange, named type and set generators. Module generators are not allowed. The **decreasing** attribute of a for loop is supported.

forward *types and routines:* The body of a type or routine may be declared **forward** and the actual body given in a later declaration.

Import lists: The **imports** clauses of Euclid modules and routines are fully enforced in Toronto Euclid. That is, every symbol defined externally to a module or routine which is used in the module or routine must be imported into the scope. Variables must be imported **var** if they are to be assigned to or passed to a **var** formal. Symbols which are declared or imported **pervasive** in an enclosing scope are automatically imported and need not be imported explicitly. The Toronto Euclid compiler fully implements **var, const, readonly** and **pervasive** in import lists.

initially *routines* in single-use modules are supported.

invariant *conditions* in modules are supported. The invariant condition is evaluated and checked at run time on exit from the initially routine, on entry to and again on exit from every exported procedure, and on entry to (but not exit from) every exported function, providing the module is declared in a checked scope. Manifest invariant conditions are evaluated at compile time.

machine dependent *records* are supported. The **aligned mod** clause causes an alignment of the record to be done by the compiler. Fields of the record are allocated at specific offsets within the record given by **at** clauses in the field declarations. The **bits** clause of full Euclid is not supported.

Manifest **if** *conditions:* No code will be generated for the **else** part of an **if** statement whose condition expression is folded to **false** at compile time nor for the **then** part of an **if** statement whose condition expression is folded to **false**. Case statements are not similarly optimized.

module *variables and types:* Single-use modules (module variables) are fully supported except that **finally** routines are not allowed. Multiple-use modules (module types) are supported but must not have an **initially** routine nor any internal initialization. (A module is multiple-use if it is declared as a type, is the component type of an array, or is the type of more than one variable. Modules nested within a multiple-use module are themselves considered multiple-use.)

Named types: Declaration and use of named types is supported. Prototypes are not supported.

Nonscalar assignment and comparison: Assignment and comparison of whole arrays, records and modules is supported. Euclid restricts non-scalar comparisons to "=" and **not** "=". Comparison of records and modules with holes (embedded unused storage) gives unpredictable results.

packed *types:* Array, record, module and set types may be declared **packed**. This has the effect of packing fields or components whose type is enumerated or subrange into bytes if possible. The Toronto Euclid compiler will pack enumerated types of 256 elements or less and subrange types in the range 0..255 into a byte on the PDP-11. Set types of 8 elements or less are always packed into a byte on the PDP-11.

pervasive *constants, types and routines:* Constants, types and routines which are declared or imported using **pervasive** are automatically imported into every subscope of the scope in which they become pervasive. (They may also be imported explicitly into a subscope if desired.)

readonly, const *and* **var** binding conditions are accepted and enforced in import lists, export lists, routine formals lists and bind statements.

Register variables: The Toronto Euclid compiler allocates scalar and set local variables of a routine to registers when possible. Four registers are available for this purpose on the PDP-11.

Routine pre- and post-conditions are evaluated and checked at run time in checked scopes. Manifest pre- and post-assertions are evaluated at compile time.

Semicolons: All semicolons are optional in Toronto Euclid.

set *variables and types* are supported. The set operators "+", "--", **xor**, "*", **in**, **not in**, "<=" and ">=" are supported. For loops with set generators are supported. Sets may have a maximum of 16 elements.

Standard components: The following standard components of full Euclid are supported in Toronto Euclid: *address, alignment, BaseType, ComponentType, first, Free, IndexType, ItsType, last, New, nil, ObjectType, Ord, Pred, size, sizeInBits, Succ.* (Note: standard components apply only to the types for which they are legal as specified in the Euclid Report.)

Structured constants: Array and record structured constants are supported. As in full Euclid, the elements of a structured constant must be manifest. The scalar values in a structured constant are not range checked.

thus *clauses* are accepted and treated as comments (as required in full Euclid). In Toronto Euclid, it is necessary to import into a closed scope only those symbols which are accessed directly. The list of symbols accessed indirectly through an imported symbol (the *thus list*) will be calculated and checked for overlap errors automatically by the compiler.

Type converters are supported for conversions from type to type. Routine type converters are not allowed.

Type pre-conditions: The **pre** condition in a type declaration is evaluated and checked at run time in checked scopes. Manifest preassertions are evaluated at compile time.

Variables at absolute locations: A variable may be declared using the **at** clause, which gives its absolute machine address.

B2. Extensions and modifications of Euclid in Toronto Euclid

Toronto Euclid extends or modifies Euclid in the following ways:

Character literals: The character literals $$E (end of file) and $$F (form feed) are provided. The corresponding $E and $F may appear in string literals.

checked *scopes:* In Toronto Euclid, the **checked** keyword causes run-time checking of all subscripts and user assertions including **pre**, **post** and **invariant** conditions. Scopes are **checked** by default. Case statement tag expressions are always run-time range checked, even in unchecked scopes. The Toronto Euclid compiler does not emit run-time checking code for Euclid legality assertions. It does not implement run-time range checking on assignment.

code *blocks:* The bodies of machine code routines appear as Unix assembly language code. The assembly language code must begin and end with a line consisting only of the character "?" in column 1. Toronto Euclid code blocks can access only the parameters to the routine.

Comments: The character "{" cannot appear in the body of a comment, so that unclosed comments can be detected by the compiler.

include *clauses* may be placed anywhere in a Toronto Euclid source program. The included file may contain any valid Euclid source text. Included files must not themselves contain include clauses.

Machine dependent features such as variables declared at an absolute location, machine dependent records, machine dependent characters and string literals containing machine dependent characters are not restricted to machine dependent modules in Toronto Euclid.

Non-scalar **const** *parameters* (except sets) are passed by reference. Scalar and set **const** parameters are passed by value. **var** and **readonly** parameters are passed by reference.

set *constructors:* In Toronto Euclid, subranges used as elements in a set constructor must be manifest. Type names are not allowed as elements in a set constructor. In Toronto Euclid, the result of a set constructor is never manifest.

Spelling rules: The Toronto Euclid compiler does not enforce Euclid capitalization rules in identifiers. Upper and lower case letters are considered equivalent in Toronto Euclid identifiers.

String literals: String literals are of type **packed array** 1..*length* **of char** in Toronto Euclid.

B3. Features not included in Toronto Euclid

The following features of full Euclid are not included in the Toronto Euclid subset:

bits *clauses (in machine dependent records):* The **bits** clause in the declaration of a field of a machine dependent record (e.g., **var** *x* (**at** 2 **bits** 1..5)) is not allowed.

checkable *collections* are not allowed.

counted *collections* are not allowed.

Export of ":=", "=" and "↑" is accepted but not enforced. In Toronto Euclid, every exported symbol is automatically exported with ":=", "=" and "↑" (if it applies). Every module automatically exports ":=" and "=" for itself. Note that export of symbols (including readonly export) is fully enforced.

Export of a subrange of symbols (e.g., **exports** (*red..blue*)) is not allowed.

finally *routines* are not allowed.

Initial values in records and multiple-use modules: Initial values for variables declared as fields of records, machine dependent records or multiple-use modules are not allowed. A multiple-use module must not have an **initially** routine. (A module is multiple-use if it is declared as a type, is the component type of an array, or is the type of more than one variable. Modules nested within a multiple-use module are themselves considered multiple-use.)

inline *routines:* The **inline** keyword is accepted but ignored.

Library includes: Include clauses of the form: **include** 'file' **from** 'library' are not allowed. Note that the simpler form, **include** 'file' is allowed.

Non-manifest array bounds: The bounds of an array's index type must be manifest. Exception: routine formal parameters which are arrays may have non-manifest bounds.

Non-scalar functions: The type returned by a function must be a scalar or set type.

Prototypes are not allowed. The features implied by **parameter**, **any** and **unknown** are not allowed.

Predefined (built-in) identifiers and standard components: The following predefined identifiers and standard components of Euclid are not supported: *Abs, Index, itsTag, Max, Min, Odd, refCount, String, StringIndex, stringMaxLength, SystemZone, zone.*

Range checking: The Toronto Euclid compiler does not implement run-time range checking on assignment. Subscripts are range checked in checked scopes. Case statement tag expressions are always range checked. The ranges of scalars in structured constants are not checked.

return when: The **when** clause is not allowed in a **return** statement. The **when** clause is allowed in **exit** statements, however.

Routine type converters: **procedure** and **function** are not accepted as the source type for a type converter.

Separate compilation of modules and routines is not supported.

Type String(n): The predefined parameterized type *String* is not allowed. String literals are allowed and have the type **packed array** 1..*length* **of** Char.

User zones (e.g., **collection of** *R* **in** *UserZone*) are not allowed.

Variant records and discriminating case statements are not allowed.

Index