Mesa Language Manual
Version 5.0
April 1979
Mesa Language Manual

by James G. Mitchell
    William Maybury
    Richard Sweet

Version 5.0
April 1979
CSL-79-3

The Mesa language is one component of a programming system intended for developing and maintaining a wide range of systems and applications programs. Mesa supports the development of systems composed of separate modules with controlled sharing of information among them. The language includes facilities for user-defined data types, strong compile-time checking of both types and interfaces, procedure and coroutine control mechanisms, and control structures for dealing with concurrency and exceptional conditions.
# CONTENTS

## CHAPTER 1. INTRODUCTION
1.1. Syntax notation  

## CHAPTER 2. BASIC DATA TYPES AND EXPRESSIONS
2.1. A slice of Mesa code  
   2.1.1. Basic lexical structure  
2.2. Simple declarations  
2.3. The fundamental operations, +, =, and #  
2.4. Basic types  
   2.4.1. The numeric types INTEGER and CARDINAL  
   2.4.1.1. Numeric literals  
   2.4.2. Type BOOLEAN  
   2.4.3. Type CHARACTER  
   2.4.4. The numeric types LONG INTEGER and LONG CARDINAL  
   2.4.5. Type REAL  
   2.4.6. Relations among basic types  
2.5. Expressions  
   2.5.1. Numeric operators  
      2.5.1.1. Domains of the numeric operators  
      2.5.1.2. The operator LONG  
      2.5.1.3. CHARACTER operators  
   2.5.2. Relational operators  
   2.5.3. BOOLEAN operators  
   2.5.4. Assignment expressions  
   2.5.5. Operator precedence  
2.6. Initializing variables in declarations  
   2.6.1. Compile-time constants  
2.7. More general declarations  

## CHAPTER 3. COMMON CONSTRUCTED DATA TYPES
3.1. The element types  
   3.1.1. Enumerated types  
   3.1.2. Subrange types  
      3.1.2.1. Subranges of numeric types  
      3.1.2.2. Range assertions  
3.2. Arrays  
   3.2.1. Declaration of arrays  
   3.2.2. Array constructors  
3.3. Records  
   3.3.1. Field lists  
   3.3.2. Declaration of records  
   3.3.3. Qualified references  
   3.3.4. Record Constructors  
   3.3.5. Default field values  
   3.3.6. Extractors  
3.4. Pointers  
   3.4.1. Constructing pointer types  
   3.4.2. Pointer operations  
   3.4.3. Long Pointers  
   3.4.4. Automatic dereferencing
### CHAPTER 7. MODULES, PROGRAMS, AND CONFIGURATIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1. Interfaces</td>
<td>101</td>
</tr>
<tr>
<td>7.2. The fundamentals of Mesa modules</td>
<td>104</td>
</tr>
<tr>
<td>7.2.1. Including modules: the DIRECTORY clause</td>
<td>105</td>
</tr>
<tr>
<td>7.2.1.1. Enumerating items from an included module: the USING clause</td>
<td>106</td>
</tr>
<tr>
<td>7.2.2. Accessing items from an included module</td>
<td>106</td>
</tr>
<tr>
<td>7.2.2.1. Qualification</td>
<td>106</td>
</tr>
<tr>
<td>7.2.2.2. OPEN clauses</td>
<td>107</td>
</tr>
<tr>
<td>7.2.3. Scopes for identifiers in a module</td>
<td>108</td>
</tr>
<tr>
<td>7.2.4. Implications of recompiling included modules</td>
<td>109</td>
</tr>
<tr>
<td>7.3. DEFINITIONS modules</td>
<td>110</td>
</tr>
<tr>
<td>7.3.1. Interface variables</td>
<td>111</td>
</tr>
<tr>
<td>7.3.2. Default fields in interfaces</td>
<td>112</td>
</tr>
<tr>
<td>7.3.3. Inline procedures in interfaces</td>
<td>113</td>
</tr>
<tr>
<td>7.3.4. Usage hints for inline procedures in interfaces</td>
<td>114</td>
</tr>
<tr>
<td>7.4. PROGRAM modules: IMPORTS and EXPORTS</td>
<td>114</td>
</tr>
<tr>
<td>7.4.1. IMPORTS, interface types, and interface records</td>
<td>115</td>
</tr>
<tr>
<td>7.4.2. Importing program modules</td>
<td>116</td>
</tr>
<tr>
<td>7.4.3. Exporting interfaces and program modules</td>
<td>116</td>
</tr>
<tr>
<td>7.4.4. IMPORTS in DEFINITIONS modules</td>
<td>116</td>
</tr>
<tr>
<td>7.5. Controlling module interfaces: PUBLIC and PRIVATE</td>
<td>117</td>
</tr>
<tr>
<td>7.5.1. Access attributes in declarations</td>
<td>117</td>
</tr>
<tr>
<td>7.5.1.1. Declared names</td>
<td>118</td>
</tr>
<tr>
<td>7.5.1.2. Names specified in field lists</td>
<td>118</td>
</tr>
<tr>
<td>7.5.1.3. Names for variant parts and for tags in variant records</td>
<td>118</td>
</tr>
<tr>
<td>7.5.2. Access attributes in TYPE definitions</td>
<td>119</td>
</tr>
<tr>
<td>7.5.3. Default global access</td>
<td>120</td>
</tr>
<tr>
<td>7.5.4. Accessing the PRIVATE predefined symbols of other modules</td>
<td>120</td>
</tr>
<tr>
<td>7.6. The Mesa configuration language, an introductory example</td>
<td>120</td>
</tr>
<tr>
<td>7.6.1. Lexicon: a module implementing LexiconDefs</td>
<td>121</td>
</tr>
<tr>
<td>7.6.2. LexiconClient: a client module</td>
<td>123</td>
</tr>
<tr>
<td>7.6.3. Binding, loading, and running a configuration: an overview</td>
<td>123</td>
</tr>
<tr>
<td>7.6.4. A configuration description for running LexiconClient</td>
<td>124</td>
</tr>
<tr>
<td>7.7. C/Mesa: syntax and semantics</td>
<td>125</td>
</tr>
<tr>
<td>7.7.1. IMPORTS, EXPORTS, and DIRECTORY in C/Mesa</td>
<td>126</td>
</tr>
<tr>
<td>7.7.2. Explicit naming, IMPORTS, and EXPORTS</td>
<td>126</td>
</tr>
<tr>
<td>7.7.3. Default names for interfaces and instances</td>
<td>127</td>
</tr>
<tr>
<td>7.7.4. Multiple exported interfaces from a single component</td>
<td>128</td>
</tr>
<tr>
<td>7.7.5. Multiple components implementing a single interface</td>
<td>129</td>
</tr>
<tr>
<td>7.7.6. Nested (local) configurations</td>
<td>130</td>
</tr>
<tr>
<td>7.8. Loading modules and configurations: NEW and START</td>
<td>131</td>
</tr>
<tr>
<td>7.8.1. The NEW operation for making copies of modules</td>
<td>131</td>
</tr>
<tr>
<td>7.8.2. How the loader binds interfaces</td>
<td>132</td>
</tr>
<tr>
<td>7.8.3. STARTING, STOPping, and RESTARTing module instances</td>
<td>132</td>
</tr>
<tr>
<td>7.8.4. Loading and starting configurations</td>
<td>133</td>
</tr>
</tbody>
</table>

### CHAPTER 8. SIGNALLING AND SIGNAL DATA TYPES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1. Declaring and generating SIGNALS and ERRORS</td>
<td>134</td>
</tr>
<tr>
<td>8.1.1. ERROR in expressions</td>
<td>136</td>
</tr>
<tr>
<td>8.2. Control of generated signals</td>
<td>136</td>
</tr>
<tr>
<td>8.2.1. Preparing to catch signals: catch phrases</td>
<td>137</td>
</tr>
</tbody>
</table>
8.2.2. The scope of variables in catch phrases
8.2.3. Catching signals
8.2.4. RETRY and CONTINUE in catch phrases
8.2.5. Resuming from a catch phrase: RESUME

8.3. Signals within signals

CHAPTER 9. PORTS AND CONTROL STRUCTURES
9.1. Syntax and an example of PORTS
9.2. Creating and starting coroutines
  9.2.1. The CONNECT statement
  9.2.2. Low-level actions for a PORT call
  9.2.3. Control faults and linkage faults
  9.2.4. Saving arguments during faults
9.3. RESPONDING PORTS

CHAPTER 10. PROCESSES AND CONCURRENCY
10.1. Concurrent execution, FORK and JOIN
  10.1.1. A process example
  10.1.2. Process language constructs
10.2. Monitors
  10.2.1. An overview of monitors
  10.2.2. Monitor locks
  10.2.3. Declaring monitor modules, ENTRY and INTERNAL procedures
  10.2.4. Interfaces to monitors
  10.2.5. Interactions of processes and monitors
10.3. Condition variables
  10.3.1. Wait, notify, and broadcast
  10.3.2. Timeouts
10.4. More about monitors
  10.4.1. The LOCKS clause
  10.4.2. Monitored records
  10.4.3. Monitors and module instances
  10.4.4. Multi-module monitors
  10.4.5. Object monitors
  10.4.6. Explicit declaration of monitor locks
  10.4.7. Inline ENTRY procedures
10.5. Signals
  10.5.1. Signals and processes
  10.5.2. Signals and monitors
10.6. Initialization

APPENDICES
A. Pronouncing Mesa
B. Programming Conventions
  B.1. Names
  B.2. Layout
  B.3. Spaces
C. Alto/Mesa Machine Dependencies
  C.1. Numeric limits
  C.2. AltoDefs
  C.3. ASCII character set and ordering of character values
  C.4. Alto/Mesa STRING procedures
Preface

This document describes the Mesa programming language. Its approach is tutorial, and it is intended to be read somewhat as a textbook. It is neither a user's guide nor a reference manual.

The Elements of Mesa Style, by James Morris, is a recommended supplement to this manual. The style manual contains several examples of well-constructed Mesa programs, with explanations of their development and commentary on using the language properly. Its purpose is to provide assistance in using the features of Mesa to write programs that work reliably and are easily maintained.

Programmers should also read the Mesa Users Handbook, which provides an introduction to the use of the Mesa system and a guide to other documentation. The Mesa System Documentation and Mesa Debugger Documentation describe facilities available in the Alto implementation of the Mesa programming system. These include input and output, which are done procedurally and are not built into the language.

Suggestions, corrections and criticisms concerning the style and content of this manual are encouraged and should be sent to your support group.

Acknowledgements

The Mesa language was designed and first implemented by the Computer Science Laboratory of the Xerox Palo Alto Research Center. The principal participants were Butler Lampson, James Mitchell, Edwin Satterthwaite, Charles Geschke, and Richard Sweet. Subsequent development and maintenance of the language and compiler have been done by the Systems Development Department of Xerox. In addition to the original participants, John Wick and Richard Johnsson have made major contributions.

The original version of this manual was written by William Maybury and edited by James Mitchell. To reflect changes in the language, many sections have since been added or revised. James Mitchell and Richard Sweet have served as editors, with additional contributions by Edwin Satterthwaite, John Wick and Richard Johnsson. David Redell wrote most of Chapter 10.

Vicki Parish, Gail Pilkington, Janet Farness, and Ode Binkley helped greatly with manuscript preparation and formatting; Gail Pilkington, James Sandman, Barbara Koalkin, and Bruce Malasky assisted in proofreading and indexing.
CHAPTER 1.

INTRODUCTION

This manual concentrates on the Mesa programming language. Mesa is really a programming system of which the language is but one part. Other components of the system are documented separately, as are the details of preparing, compiling, debugging and running Mesa programs.

Each chapter of this manual discusses some aspect of the language, using examples as well as descriptions of semantics and syntax. The chapters emphasize different language features and provide different levels of detail. The complete treatment of some features requires more than one chapter. Generally, earlier chapters introduce topics, and later ones supply additional detail. Titles of chapters, sections and subsections indicate the language issues with which they deal.

In each major section, information is presented at three levels:

1. Ordinary usage (motivation, forms and semantics), frequently with examples.
2. Syntax equations (when appropriate).
3. Fine points (if applicable): restrictions, special cases, references to later material, precise semantics, etc.

Level (1) is intended to offer a basic understanding of Mesa. Reading only first level material should be adequate to begin programming in the language. Levels (2) and (3) supply more detail and provide information about the full power of Mesa.

As a rule, these levels of discourse occur separately and in the indicated order. A section with a heading followed by an asterisk (*) deals with specialized material that can be skimmed or skipped entirely on first reading. Occasionally, fine points or syntactic details are presented within first-level material. The reader will be able to distinguish between levels by their appearance. Fine points are written in a small font, like this. Syntax equations and syntactic categories appear in the following font: FontForSyntax.

Any italicized word or phrase is important. If a Mesa technical term is being introduced, it will be in italics; if a term is used before being defined, it will be italicized to warn the reader that it should not be taken lightly and that it has a particular meaning in Mesa. Occurrences of a technical term, once defined, are not distinguished. Lastly, names appearing in programs are italicized in both the program text itself and the explanations of that text.

Programming examples are indented relative to the surrounding text to distinguish them.
1.1. Syntax notation

Mesa's grammar is described by syntax equations written using a variation of Backus-Naur Form (or BNF). For those unfamiliar with BNF, an explanation follows. Reading and understanding that explanation is imperative for full use of this manual; in a first reading, details of the syntax equations can safely be skipped. Those familiar with BNF should scan this section to discover the particular variation being used.

An individual syntax equation defines a portion of the Mesa grammar. It specifies a rule for forming some class of phrases in the language. A phrase class has a name, e.g., Program, and is defined by one or more syntax equations. Phrase names are always printed in the syntax font when their use is meant to be technically accurate. For example, an OctalDigit, which can be any of 0, 1, 2, ..., 7, is defined by the equation:

\[ \text{OctalDigit} ::= 0|1|2|3|4|5|6|7 \]

Each equation consists of a phrase name on the left, followed by the operator ::= (which should be pronounced "is defined to be"), in turn followed by a formation rule for that phrase class. A formation rule consists of one or more alternatives, separated by the syntactic operator vertical bar, | (which should be pronounced "or"). The ordering of alternatives is not important. In the definition of OctalDigit, "3" is an alternative.

Each alternative is a sequence of symbols, where a symbol is either a phrase name (in the syntax font) or a syntactic literal. In a syntax equation, a literal symbol stands for itself. The reserved words of Mesa, such as BEGIN, appear as literals; they are always written using upper-case characters in the font shown. The digits 0, 1, 2, etc. and special characters, such as =, + and -, also are used to form literal symbols. Some composite symbols are formed from more than one special character, e.g., =>. Spaces in syntax equations are used only to separate the items in the rules and have no special significance.

The phrase name empty is often used as one of the alternatives in a formation rule. It means that the rule permits an "empty" phrase as one of its alternatives (i.e., an actual phrase is optional; it may or may not occur in the result of applying the formation rule).

Comments embedded in syntax rules are preceded by a double dash, --, and appear to the right, e.g.,

\[ \text{Digit} ::= \text{OctalDigit}|8|9 \quad -- \text{a decimal digit is an OctalDigit or an 8 or a 9} \]

Often, only part of the total definition of a phrase class is given. To indicate that there are other ways of forming phrases of that class, an ellipsis (...) is used as an alternative within the rule. The definition of Statement is distributed throughout much of the manual in this way. When a certain statement form, such as the AssignmentStmt, is being discussed, the following partial rule appears:

\[ \text{Statement} ::= \text{AssignmentStmt}|... \quad -- \text{this is just an example.} \]

One can read this as, "A Statement is defined to be an AssignmentStmt, among other things."

Within a single alternative, the order of symbols is important. The alternative acts as a "template" for forming an actual phrase; literal names and literal characters are copied, while substitutions are made for the phrase names. Consider the following example:

\[ \text{ReturnStmt} ::= \text{RETURN} | \text{RETURN} \text{Constructor} \]

("A ReturnStmt is defined to be \text{RETURN} or \text{RETURN} followed by a Constructor.") The second alternative means that \text{RETURN} and some actual phrase defined by \text{Constructor} occur in exactly that order.
Syntax equations can indicate recursive substitution; for example:

\[ \text{IdList} :: = \text{identifier} | \text{identifier} , \text{IdList} \]

In a Mesa program, an \textit{identifier} is basically a name. This equation defines an \textit{IdList} to be a list of one or more names, with commas separating them if there is more than a single name in the list.

This result is explained as follows. The formation rule for \textit{IdList} consists of two alternative rules:

1. Rule 1: (First alternative) "An \textit{IdList} is defined to be an \textit{identifier}," i.e., any one \textit{name} can replace an \textit{IdList}.
2. Rule 2: (Second alternative) "An \textit{IdList} is defined to be an \textit{identifier} followed by a comma followed by another \textit{IdList}," i.e., \textit{name}, \textit{IdList} can replace an \textit{IdList}.

To derive a single name, use Rule 1 as shown below. (Note: The substitutions are emphasized by writing them in \textit{italics}.)

\[ \text{IdList} ::= \text{name} \quad (\text{by Rule 1}) \]

To derive two names separated by a comma:

\[ \text{IdList} ::= \text{name}, \text{IdList} \quad (\text{by Rule 2}) \]
\[ \text{name, name} \quad (\text{by Rule 1}) \]

To derive three names separated by commas:

\[ \text{IdList} ::= \text{name}, \text{IdList} \quad (\text{by Rule 2}) \]
\[ \text{name, name, IdList} \quad (\text{by Rule 2}) \]
\[ \text{name, name, name} \quad (\text{by Rule 1}) \]

To derive \( n \) names separated by commas, use Rule 2 \( n-1 \) times and then use Rule 1.

The following syntax equation also relies on recursion:

\[ \text{StmtSeries} :: = \text{empty} | \text{Statement} | \text{Statement} ; \text{StmtSeries} \]

The equation is read as, "A \textit{StmtSeries} is defined to be empty, or a single statement, or a series of statements separated by semicolons; the last statement may be followed by a semicolon."

A trailing semicolon is possible because:

1) A \textit{StmtSeries} may take the form specified by the third alternative, "\textit{Statement} ; \textit{StmtSeries}."
2) After some number of further substitutions using the third alternative, the recursive reference to \textit{StmtSeries} may take the "\textit{empty}" form, i.e., "... \textit{Statement} ; \textit{empty}.
3) \textit{empty} is replaced by nothing at all, i.e., "... \textit{Statement} ;".

Commas and semicolons are used as major separators for a variety of constructs in Mesa. To distinguish between such constructs, a convention is adopted that the suffix "List" on a phrase name implies a sequence separated by commas, while "Series" implies a sequence separated by semicolons. This convention is reflected by the phrase names \textit{IdList} and \textit{StmtSeries} above.
CHAPTER 2.

BASIC DATA TYPES AND EXPRESSIONS

This chapter presents some of the fundamentals of Mesa. It discusses how to declare, initialize and assign values to variables. It also describes the basic types for numeric, character and Boolean data, as well as the operators used to construct expressions having these types.

The Mesa language is strongly typed. The programmer is given a collection of predefined types and the ability to construct new ones; he is encouraged to choose or invent suitable types for each particular application. Every variable used in a Mesa program must be declared to have one of these types; every constant has a type; and every expression has a type derived from its components and context. All types can be deduced by static analysis of the program, and the language requires that each value be used in a way consistent with its type according to rules specified here and in chapter 3. The type of an object determines its representation and structure as well as the set of applicable operations. In addition, the type system can be used to partition the universe of objects and avoid confusion, even among classes of objects that are represented identically.

2.1. A slice of Mesa code

The example below is an excerpt from a Mesa program. It assigns to \(gcd\) the greatest common divisor (GCD) of a pair of integers, \(m\) and \(n\) (where \(m\), \(n\) and \(gcd\) are integer variables in the program from which this excerpt was taken; we assume their values need not be preserved). The example uses the Euclidean Algorithm for finding the GCD of two numbers and works as follows:

If both \(m\) and \(n\) are zero, the GCD is zero (by convention).

Otherwise, repeat the following until \(n\) is zero: find the remainder of dividing \(m\) by \(n\); set \(m\) to the value of \(n\); then set \(n\) to the remainder. The final value of \(m\) is the GCD of the original \(m\) and \(n\) except that it may be negative; taking its absolute value gives the GCD.

Example. Slice of Mesa Code Using the Euclidean Algorithm

\[
\begin{align*}
\text{IF } m=0 \text{ AND } n=0 & \text{ THEN } gcd \leftarrow 0 \quad \text{-- by convention} \quad (1) \\
\text{ELSE} & \quad (2) \\
\text{BEGIN} & \quad (3) \\
r: \text{ INTEGER}; & \quad (4) \\
\text{UNTIL } n=0 & \quad (5) \\
\text{DO} & \quad (6) \\
r \leftarrow m \text{ MOD } n; & \quad \text{-- } r \text{ gets remainder of } m/n \quad (7) \\
m \leftarrow n; n \leftarrow r; & \quad \text{-- update variables} \quad (8) \\
\text{ENDLOOP}; & \quad (9) \\
gcd \leftarrow & \quad \text{-- in case one of } m \text{ or } n \text{ was negative -- } \text{ABS}[m]; \quad (10) \\
\text{END;} & \quad (11) \\
\end{align*}
\]
The example contains twelve lines of source code, including comments. The numbers in parentheses at the right side are for reference only and are not part of the source code. Comments begin with the symbol "--" and terminate at line endings. They may also be completely embedded within lines, in which case they both begin and end with "--".

Line (2) begins an IF statement that uses the values of $m$ and $n$ to select between two alternatives. If both values are zero, the assignment statement following THEN is executed; it assigns the value 0 to $gcd$ (the character "=" is Mesa's assignment operator). If either is nonzero, the assignment is skipped and the compound statement following ELSE (lines (4) through (12) inclusive) is executed. (Distinguishing the two cases is actually unnecessary, but doing so illustrates more features of Mesa.)

The second alternative is a block, a series of declarations followed by a series of statements, all bracketed by "BEGIN" and "END". Line (5) declares a variable $r$ of type INTEGER for use within that block. A semicolon separates the declaration from the statements that follow it.

The iteration in the algorithm is performed by the loop (UNTIL $n=0$ DO...ENDIF), which contains three embedded assignment statements. The loop repeats until $n$ is equal to zero. If it is zero at the outset, the embedded statements are not executed at all. Statements are separated by semicolons. Asemicolon separates the declaration from the statements that follow it.

Within the loop, line (8) assigns to $r$ the value of the expression "$m \text{ MOD } n$", which gives the remainder of dividing $m$ by $n$. Line (9) updates $m$ to contain the previous value of $n$ and then updates $n$ for the next iteration, if any. Control transfers from the end of the loop, line (10), back to line (6), where the new value of $n$ is tested. If it is not zero, the loop is repeated; otherwise, execution continues with the first statement following the loop, line (11).

When control reaches the assignment statement in line (11), $m$ either has its original value (if $n$ was zero) or contains the value $n$ had just before it became zero. The expression "ABS[$m$]" has the form used for calling a function and passing it one or more arguments; square brackets enclose the argument list. Normal parentheses, "(" and ")", are used only for nested expressions, e.g., "$a*(b+c)/(d*e*f)$." The assignment places the absolute value of $m$ into $gcd$; this is the correct result. At this point, the reader is urged to trace through the example with initial values for $m$ and $n$ of 15 and 12, respectively; the result should be $gcd=3$.

### 2.1.1. Basic lexical structure

The names $gcd$, $m$, $n$, and $r$ in the example are called identifiers. The general form of an identifier is given by the following (informal) syntax:

An identifier is a sequence consisting of any mixture of upper-case letters, lower-case letters or digits, the first of which is a letter. Upper and lower case letters are different and do distinguish identifiers.

The following, valid identifiers are all distinct:

```
aBc  Abc  DiskCommandWord  displayVector  mach1  x32y40
```

Identifiers consisting entirely of capital letters are reserved for use by the Mesa language. Some, such as IF, are punctuation symbols; others name built-in types, such as INTEGER, or functions, such as ABS. All such words that have special meaning are not to be defined by the programmer are called reserved words. It is legal for the programmer to use fully capitalized identifiers, but he risks a clash with a reserved word (possibly a new one in some future version of the language). To avoid this, at least one digit or lower case letter should appear in any identifier. Appendix E lists the current set of reserved words.
Mesa uses the blank (or space) character to separate basic lexical units of the language (such as reserved words and identifiers). Blanks are significant separators of lexical units. They may not be embedded in identifiers, composite symbols (such as >=), or numeric literals (such as 1000). Blanks are meaningful in STRING constants (section 6.1.1), and there is a CHARACTER constant for space (section 2.4.3). As a separator, any sequence of contiguous blanks is equivalent to a single blank. A TAB character also behaves exactly as a blank when used as a separator.

A carriage-return character behaves as a blank for separating lexical units also, but it has one extra function: if the last part of a line is a comment, the carriage return acts as the terminator of that comment. Thus, multiline comments (those containing carriage returns) must begin with "--" on each new line. Line breaks have no significance as statement separators. For example, the single loop statement in the example extends over a number of lines, and a semicolon is used to separate two statements in a series.

Semicolons are used for separating declarations, for separating a series of declarations from following statements, and for separating statements in a series from one another. They cannot be used with abandon, however: care is necessary when writing if statements (sec. 4.2.1) or select statements (sec. 4.3.1). Multiple statements can be written on a single line, separated by semicolons.

2.2. Simple declarations

The example (Euclidean Algorithm) contains the following declaration:

```
r: INTEGER;
```

This declares r to be a variable of type INTEGER (sec. 2.4.1), one of Mesa's built-in types. More than one variable can be declared at the same time. For instance,

```
x, y, divisor: INTEGER;
```

declares identifiers x, y and divisor as variables of type INTEGER. These examples reflect the two primary purposes of every declaration:

- to designate one or more identifiers as variables, and
- to specify their type.

A declaration always begins with a single identifier or a list of identifiers. Conventionally, "list" is used to denote a single item as well as multiple items separated by commas. An identifier list (IdList) is defined as follows:

```
IdList ::= identifier | identifier, IdList
```

A declaration begins with an IdList followed by a colon. The colon is followed by a type specification (INTEGER, for instance, is a type specification).

2.3. The fundamental operations: assignment, equality and inequality

The example contains the following five assignment statements:

```
gcd ← 0
r ← m MOD n
m ← n
n ← r
gcd ← ABS[m]
```
An assignment statement has the following syntax:

\[
\text{AssignmentStmt} ::= \text{LeftSide} \gets \text{RightSide} | \ldots
\]

\[
\text{LeftSide} ::= \text{identifier} | \ldots \quad \text{plus forms for array indexing, etc.}
\]

\[
\text{RightSide} ::= \text{Expression}
\]

The RightSide may be any expression (section 2.5) provided that its type conforms to that of the LeftSide. "Conforms" is defined in section 2.4.6 and is discussed further in section 3.5; for now, it can be taken to mean: "is the same as." The LeftSide may be a simple variable or a component of an aggregate variable (such as an element of an array). In any event, a LeftSide denotes a variable, something capable of receiving values. A LeftSide cannot, for example, be a constant, while a RightSide can.

The assignment operation (\(\gets\)), the equality operation (\(=\)) and the inequality operation (\(#\)) are called the fundamental operations. They can be applied to values of most types (including, for instance, entire arrays). The rules governing which pairs of operands may be used in a fundamental operation are detailed in section 3.5.

2.4. Basic types

The types of variables in a Mesa program fall into two broad classifications, built-in types and user-defined types. Chapter 3 describes how a programmer can define new data types using type constructors. This section discusses the basic, built-in types. These include several numeric types (INTEGER, LONG INTEGER, CARDINAL, LONG CARDINAL and REAL), a type for logical values (BOOLEAN), and a type for individual character values (CHARACTER). The built-in type STRING (for sequences of characters) is described in chapter 6.

2.4.1. The numeric types INTEGER and CARDINAL

Mesa provides two standard numeric types, one with values ranging over the signed integers; the other, over the unsigned integers. Neither type completely mirrors the corresponding mathematical abstraction (the integers \(\mathbb{Z}\) or the natural numbers \(\mathbb{N}\), respectively) because a finite representation is used for values of each type. The range of the type INTEGER is (approximately) symmetric about zero, and values of type INTEGER are represented as signed numbers. The range of the type CARDINAL is some finite interval of the natural numbers that includes zero, and values of type CARDINAL are represented as unsigned numbers. "Signed" and "unsigned" are not types; rather, they describe the machine representation of a numeric value.

The programmer must choose an appropriate type for each numeric variable. CARDINALS offer a somewhat greater positive range than INTEGERS, and this is significant in a few applications, e.g., those that manipulate addresses that might be the same size as the word size. More importantly, declaring a variable to have type CARDINAL asserts that its value is always nonnegative; the compiler can use such assertions to perform more checking and to generate better code. Programmers are encouraged to declare as much information about each variable as possible; the ranges of numeric variables can be further constrained by using subrange types (section 3.1.2).

The types INTEGER and CARDINAL are distinct and not interchangable. They are, however, closely related. Mesa allows most combinations of these types to occur within assignments and arithmetic expressions (but not relational expressions). Care is necessary to avoid ambiguity and failures of representation when values with different representations are mixed. This is discussed further in sections 2.4.6 and 2.5.1.1.
2.4.1.1. **Numeric literals**

A *numeric literal* is an instance of the phrase class `number`, defined as follows:

A `number` is a sequence of digits. The digits may optionally be followed by the letter B or D, which in turn may optionally be followed by another sequence of digits denoting a scale factor. No spaces are allowed within numeric literals.

If D is specified explicitly, or if neither B nor D appears, the `number` is treated as decimal. The letter B means the `number` is octal (radix 8). A scale factor indicates the number of zeros to be appended to the first sequence of digits; the *scale factor itself is always a decimal number*. The literals below all denote the same value:

- 6400
- 6400D
- 64D2
- 14400B
- 144B2

A numeric literal always denotes a nonnegative number (i.e., 5 is considered to be an expression in which the unary negation operator is applied to the literal 5 to produce an `INTEGER` value). To be valid in a context requiring a `CARDINAL`, the value of the literal must be a valid `CARDINAL` number. Similarly, if an `INTEGER` is required by context, the value must be a valid (positive) `INTEGER`. (See section 2.4.4 for more details)

2.4.2. **Type BOOLEAN**

A `BOOLEAN` value can be either `TRUE` or `FALSE`, and these are the only literals of type `BOOLEAN`; i.e.,

```
BOOLEANLiteral ::= FALSE | TRUE
```

`BOOLEAN` expressions are used in conditional statements (following `IF`) and in certain loop constructs. For instance, the following skeletal form describes the flow of control in Example 1:

```
IF m = 0 AND n = 0 THEN . . .
ELSE
    . . .
    UNTIL n = 0
    DO
    . . .
ENDLOOP;
```

The expression "n = 0" is a `BOOLEAN` expression; its value is `TRUE` if the value of `n` is zero and `FALSE` otherwise. The expression "m = 0 AND n = 0" is also a `BOOLEAN` expression; its value is `TRUE` just if both relations are. The relational and logical operators discussed in sections 2.5.2 and 2.5.3 all yield `BOOLEAN` values.

Variables of type `BOOLEAN` can be assigned values and appear as operands (although not of arithmetic operators) just as any other Mesa variables. For example, the above program outline could validly be replaced by the following:

```
mIsZero, nIsZero: BOOLEAN;

mIsZero = (m = 0); nIsZero = n = 0; -- compute whether m and n are zero
IF mIsZero AND nIsZero THEN . . .
ELSE
    . . .
    UNTIL nIsZero = TRUE
    DO
    . . .
    nIsZero = n = 0;
ENDLOOP; -- equivalent to just nIsZero by itself
```

```
    . . .
    nIsZero = n = 0;
ENDLOOP; -- recompute whether n is zero just before testing
```

2.4.3. Type CHARACTER

A value of type CHARACTER represents a single character of text. CHARACTER values are ordered (according to the order specified in appendix C) and can be compared using the normal arithmetic relations. CHARACTER values are distinct from numbers, and they cannot be assigned to variables with numeric types. Limited arithmetic is, however, allowed on characters (section 2.5.1.2).

A characterLiteral is written as an apostrophe (') immediately followed by a single character (which can be a blank, carriage-return, semicolon, apostrophe, or any other character) or as an octal number followed by C. For example:

```plaintext
lowerCaseA = 'a;
mark = ';
endMarker = ';;
asciiCR = 15C;
```

---

2.4.4. The numeric types LONG INTEGER and LONG CARDINAL *

For some applications, the ranges of the numeric types introduced in section 2.4.1 are too limited. Mesa provides both a predefined type LONG INTEGER, with signed representation, and a predefined type LONG CARDINAL, with unsigned representation, for such applications. These types offer greater ranges, but their values occupy more storage and are generally more time-consuming to manipulate than those of the previously introduced numeric types.

In an implementation, values of types INTEGER and CARDINAL are expected to be represented by single machine words, while values of types LONG INTEGER and LONG CARDINAL are expected to occupy two words. For this reason, INTEGER and CARDINAL will be referred to as short numeric types; LONG INTEGER and LONG CARDINAL, as long numeric types. On a machine using two's complement arithmetic and a word length of \( N \) bits, the following table indicates the range spanned by each numeric type (".." replaces the mathematician's comma in this interval notation):

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>([2^N - 1 .. 2^N - 1])</td>
</tr>
<tr>
<td>CARDINAL</td>
<td>([0 .. 2^N])</td>
</tr>
<tr>
<td>LONG INTEGER</td>
<td>([2^{2N - 1} .. 2^{2N - 1}])</td>
</tr>
<tr>
<td>LONG CARDINAL</td>
<td>([0 .. 2^{2N}])</td>
</tr>
</tbody>
</table>

The actual ranges for these types are given in appendix C, the machine dependencies appendix.

Long numeric constants are denoted by numeric literals defined by the phrase class number (section 2.4.1.1). The allowable type of any decimal or octal literal is determined by its value, as summarized by the following table (using the conventions introduced in the preceding paragraph):

<table>
<thead>
<tr>
<th>Range</th>
<th>Allowable Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>([0 .. 2^N - 1])</td>
<td>INTEGER, CARDINAL, LONG INTEGER, LONG CARDINAL</td>
</tr>
<tr>
<td>([2^N - 1 .. 2^N])</td>
<td>CARDINAL, LONG INTEGER, LONG CARDINAL</td>
</tr>
<tr>
<td>([2^N .. 2^{2N - 1}])</td>
<td>LONG INTEGER, LONG CARDINAL</td>
</tr>
<tr>
<td>([2^{2N - 1} .. 2^{2N}])</td>
<td>LONG CARDINAL</td>
</tr>
</tbody>
</table>

As in the case of short numeric types, the types LONG INTEGER and LONG CARDINAL are distinct but closely related. Mesa allows most combinations of these types and the types INTEGER and CARDINAL to occur within assignments, arithmetic expressions and relational expressions, but care is necessary when this is done (see sections 2.4.6 and 2.5.1.1).
2.4.5. Type REAL (interim) *

The values of Mesa's type REAL are approximations of mathematical real numbers. These approximations are sometimes called floating-point numbers. For the current version of Mesa, a standard representation for floating-point values has not been chosen. The language nevertheless provides some help with floating-point computation. It allows declaration and assignment of REAL values, and REAL expressions constructed using the standard infix operators are converted to sequences of procedure applications by the compiler.

A REAL value is assumed to occupy the same amount of storage as a LONG INTEGER (i.e., two words). Beyond this, no assumptions are made about the representation of REALS. There are no literals with type REAL. Users of real arithmetic must provide an appropriate set of procedures for performing the arithmetic and relational operations.

Although Mesa provides no denotations of REAL literals, it does provide automatic conversion from INTEGER, LONG INTEGER, CARDINAL or LONG CARDINAL to REAL (section 2.4.6). Thus numbers (numeric literals) can appear in REAL expressions and provide denotations of certain REAL constants.

2.4.6. Relations among basic types *

If two types are completely interchangable, they are said to be equivalent. A value having a given type is acceptable in any context requiring a value of any other type equivalent to it; there is no operational difference between two equivalent types. None of the basic types discussed in section 2.4 is equivalent to another basic type.

One type is said to conform to another if any value of the first type can be assigned to a variable of the second type. A type trivially conforms to itself or to any type equivalent to itself. In more interesting cases, an automatic application of a conversion function may be required prior to the assignment. Conformance and its implications are discussed further in section 3.5.

There are nontrivial conformance relations involving the types INTEGER, LONG INTEGER, CARDINAL, LONG CARDINAL and REAL. These relations allow certain combinations of the numeric types to be mixed, not only in assignments but also in arithmetic and relational operations (section 2.5). They also permit these types to share denotations of constants (section 2.4.4). The conformance relations can be summarized as follows:

INTEGER and CARDINAL conform to INTEGER.
INTEGER and CARDINAL conform to CARDINAL.
INTEGER, LONG INTEGER, CARDINAL and LONG CARDINAL conform to LONG INTEGER.
INTEGER, LONG INTEGER, CARDINAL and LONG CARDINAL conform to LONG CARDINAL.
INTEGER, LONG INTEGER, CARDINAL, LONG CARDINAL and REAL conform to REAL.

Pairs of numeric types not on this list do not conform; e.g., it is not possible to assign a LONG INTEGER to an INTEGER or a REAL to a CARDINAL.

Particular care is required when numeric types with different representations are intermixed. Mathematically, \( \mathbb{Z} \supset \mathbb{N} \); however, it is not necessarily true that INTEGER \( \supset \) CARDINAL or that LONG INTEGER \( \supset \) LONG CARDINAL. For instance, with the assumptions above, the intersection of INTEGER and CARDINAL is \([0, 2^{N-1})\). Within this interval, the signed and unsigned representations agree, and the interpretation of a short numeric value is unambiguous. If a CARDINAL value lies in this range, it can validly be assigned to an INTEGER variable, and vice-versa; outside this range, the value represented by a given word depends upon whether it is viewed as a CARDINAL or as an INTEGER. Similar
considerations apply to LONG CARDINAL and LONG INTEGER.

Example:

With the assumptions above and $N=16$, the unsigned value $177777B$ and the signed value $1$ are encoded by the same bit pattern.

Assignment of an unsigned value to an INTEGER variable, or of a signed value to a CARDINAL variable, implicitly invokes a conversion function, which is just an assertion that the value to be assigned is an element of $\text{CARDINAL} \cap \text{INTEGER}$. It is the responsibility of the programmer to ensure that the conversion is valid. In many cases this is not too difficult, but programmers are urged to avoid mixing signed and unsigned representations when this is possible. It almost always is.

Mesa does guarantee that $\text{LONG} T \supseteq T$ for any type $T$ and that $\text{LONG INTEGER} \supset \text{CARDINAL}$; thus it is always valid to assign a short numeric value to a LONG INTEGER variable or a short unsigned value to a LONG CARDINAL variable. The properties of conversion to type REAL are not specified by the language.

Some fine points:

- A user supplied procedure $\text{FLOAT}$ is automatically applied to convert a value from type $\text{LONG INTEGER}$ to $\text{REAL}$. Short numeric values are converted first to $\text{LONG INTEGER}$ and then to $\text{REAL}$.
- Conversion from a short numeric value to a $\text{LONG INTEGER}$ (and thus to a $\text{REAL}$) is substantially more efficient when the value has an unsigned representation.
- The conversion of a constant to type $\text{REAL}$ occurs every time the containing expression is evaluated at run-time.

Neither $\text{BOOLEAN}$ nor $\text{CHARACTER}$ conforms to any other basic type.

Examples:

- $i$: INTEGER; $n$: CARDINAL; $ii$: LONG INTEGER; $x$: REAL;
  
  (valid) $i \leftarrow 0$;
  $ii \leftarrow 0$;
  $x \leftarrow n$;
  $x \leftarrow ii$;

- (invalid) $i \leftarrow x$;
  $n \leftarrow \text{TRUE}$;

2.5. Expressions

Expressions are constructs describing rules of computation for evaluating variables and for generating new values by the application of operators. The overall syntactic rule for an expression is given by

$$\text{Expression} ::= \text{Disjunction} | \text{AssignmentExpr} | \text{IfExpr} | \text{SelectExpr} \ldots$$

The Disjunction form includes all the numeric operations, relational operations, and BOOLEAN (logical) operations and is discussed in this section. An AssignmentExpr allows one to write multiple assignments in a single statement and is discussed in section 2.5.4. The IfExpr and SelectExpr forms are discussed in chapter 4.

The basic unit from which expressions are built is called a Primary. This syntactic class includes references to variables, literals, function calls (chapter 5), and any arbitrary expressions embedded in parentheses:
Primary ::= Variable | Literal | (Expression) | FunctionCall . . .
Variable ::= LeftSide
Literal ::= number | BooleanLiteral | characterLiteral
FunctionCall ::= BuiltinCall | Call -- defined in chapter 5

Recall that every expression has a well-defined type in Mesa. The general rules for determining the type of an expression from the types of its constituent parts are given in section 3.5. In this section, the types of the basic expression forms (as functions of the types of their operands) will be outlined. For example, the type of a Primary is the type of the Variable or Literal involved, or reduces to the type of the Expression within parentheses, or is the type of the value returned by the BuiltinCall (some of which are defined below) or the Call of a user-defined procedure (section 5.1).

A Primary can be of almost any type; this is not true of most of the expression forms built up using Mesa's operators. Some operators are numeric and some are BOOLEAN. The next sections discuss the numeric operations, the relational operations, and the operations applicable only to BOOLEAN values. Considered together, the operators form a single hierarchy with respect to their precedence, which is described with each operator class and summarized in section 2.5.5.

2.5.1. Numeric operators

The operations on numeric values are addition, subtraction, multiplication, division, modulus, and arithmetic negation. The syntax for this group of operations is

Factor ::= Primary | − Primary -- negation
Product ::= Factor | Product MultiplyingOperator Factor
MultiplyingOperator ::= × | ÷ | MOD
Sum ::= Product | Sum AddingOperator Product
AddingOperator ::= + | −

These operators have their usual mathematical meanings. The division operation on integers, /, always truncates toward zero; thus \((i/j) = \lceil i/j \rceil\). The MOD operator yields the remainder of dividing one number by another (MOD is not applicable to REAL operands). MOD is defined by the relation \((i/j)_\rceil = (i \mod j) = i\), and the sign of the result of MOD is always the sign of the dividend. (This is the reason that line 11 of Example 1 takes the absolute value of the computed \(gcd\); if \(m = 12\) and \(n = 8\) initially, the \(gcd\) would be 4 if its absolute value were not taken.)

The built-in function \(\text{MIN}\) computes the minimum value in a list of expressions; similarly, the \(\text{MAX}\) function, the maximum value. The built-in function \(\text{ABS}\) computes the absolute value of its argument. The syntax for calls on the built-in functions is

BuiltinCall ::= MIN [ExpressionList] | MAX [ExpressionList] | ABS [Expression] | . . . -- other built-in functions later
ExpressionList ::= Expression | ExpressionList , Expression

For the arithmetic operators and built-in functions, the order in which the operands are evaluated is undefined, but the syntax implies a precedence ordering that controls the association of operators with their operands. In that ordering, unary negation precedes the multiplying operators, which in turn precede the adding operators. \(\text{Sequences of operators of the same precedence associate from left to right}\) (with the exception of the embedded assignment operator, section 2.5.4). Thus, an
expression such as \( a + b \cdot c \) does not specify the order of evaluation of \( a, b \) and \( c \) but does require that the operations be performed in the following order: negate \( c \); then multiply the result by \( b \); finally, add that result to the value of \( a \).

Examples:

\[ i, j, k: \text{INTEGER}; \ m, n: \text{CARDINAL}; \]

Factors:

\[
\begin{align*}
&n \\
&15 \\
&i+j+k \\
&15 \\
&\text{MIN}(i, j, k, 15)
\end{align*}
\]

Products:

\[
\begin{align*}
&m*n \\
&i/15 \\
&n \mod 8 \\
&m/n*10 & \text{-- same as } (m/n)*10 \text{ because of left-associativity} \\
&k*(i+1)/2 \mod 3 & \text{-- same as } (((k)*(i+1))/2) \mod 3
\end{align*}
\]

Sums:

\[
\begin{align*}
&i+1 \\
&i+j \\
&j \mod 8 & \text{-- same as } n(n \mod 8) \text{ because of precedence} \\
&m \cdot m/n*n & \text{-- same as } m \mod n
\end{align*}
\]

2.5.1.1. Domains of the numeric operators *

In principle, each arithmetic operator designates the corresponding mathematical function. Unfortunately, the hardware underlying any implementation of Mesa does not provide this function but only a set of related partial functions. For each operator, the compiler must choose as appropriately as possible from this set. The choice is made by considering the types of the operands.

Example:

With the usual assumptions, \( 177777B \) and \( 1 \) are represented by the same bit pattern. The value of \( 177777B > 0 \) is TRUE, but that of \( 1 > 0 \) is FALSE.

Mesa provides the operators \( +, -, *, /, \text{MIN}, \text{MAX} \) and \text{ABS} for all the numeric types. The operation \( \mod \) is defined for all numeric types except \text{REAL}; the operation of unary negation, for all but \text{CARDINAL} and \text{LONG CARDINAL}. For each of these operators, the type of the result is the same as the type of the operands. In addition, the result of the operation is considered to have signed representation if all the operands have signed representation, and to have unsigned representation if all the operands have unsigned representation. Thus, adding two \text{INTEGER} values yields an \text{INTEGER} result, and dividing one \text{CARDINAL} by another yields a \text{CARDINAL} result.

Some fine points:

Division and modulus operations on short numeric values are substantially more efficient if their operands are unsigned.

Addition, subtraction, and comparison of long numeric values are fast; multiplication and division are done by software and are relatively slow.

Operations upon \text{REAL} values are implemented as calls on user-supplied procedures. These procedures must be assignable to variables declared as follows (chapter 5):

\[
\begin{align*}
&\text{FADD, FSUB, FMUL, FDIV: \text{PROCEDURE [REAL, REAL] RETURNS [REAL];}} \\
&\text{FCOMP: \text{PROCEDURE [REAL, REAL] RETURNS [INTEGER];}} \\
&\text{FLOAT: \text{PROCEDURE [LONG INTEGER] RETURNS [REAL];}}
\end{align*}
\]

All other \text{REAL} arithmetic operations are fabricated from these primitives.
Although the mathematical integers ($\mathbb{Z}$) and real numbers are closed under all these operations (except division by zero), the subranges defining the types INTEGER, LONG INTEGER, CARDINAL and LONG CARDINAL generally are not. When the result of an operation falls outside the range of its assumed type, a representational failure called overflow or underflow occurs. In the current version of Mesa, it is the programmer’s responsibility to guard against overflow and underflow conditions.

The implications of Mesa’s conventions for subtraction are worth emphasizing. If both operands have valid signed representations, the result has a signed representation. If both have only unsigned representations, the result has an unsigned representation and is considered to overflow if the first operand is less than the second.

Example:

\[
i: \text{INTEGER}; \quad m, n: \text{CARDINAL};
\]
\[
i \leftarrow m\tilde{n};
\]
\[
i \leftarrow \text{if } m \geq n \text{ THEN } m\tilde{n} \text{ ELSE } (n\tilde{n});
\]

The arithmetic operations are defined for operands that all have the same type, but it is possible to mix numeric types (and thus representations) within an expression. In this case, operands are converted as necessary to the "smallest" type to which all the operands conform, the operation for that type is applied, and the result also has that type. The rule for expressions involving type REAL is easy to state:

If any operand has type REAL, the REAL operation is used.

The rules governing combination of numeric operands with differing representations involve some additional concepts and are stated in section 3.6. Again, the programmer should try to avoid such combinations when possible. (Recall that literals in INTEGER \cap CARDINAL have whatever representation is required by context.)

2.5.1.2. The operator LONG *

The built-in function LONG converts any value with a short numeric type to a long numeric type. A value with an unsigned representation is converted to LONG CARDINAL; one with a signed representation, to LONG INTEGER. The syntax is as follows:

\[
\text{BuiltInCall ::= \quad \ldots | \text{LONG [ Expression ]}}
\]

This operation is necessary when the standard conversion rules do not give the desired result. It can also be used to emphasize the conversion.

Example:

\[
\text{LONG}[m*n]
\]
-- "short" multiplication, overflow lost
\[
\text{LONG}[m]*\text{LONG}[n]
\]
-- "long" multiplication

Some fine points:

Lengthening a single-precision expression is substantially more efficient if that expression has an unsigned representation.

The Mesa implementation provides standard procedures (not part of the language) for performing certain multiplication and division operations in which the operands and results do not all have the same length. These procedures provide less expensive equivalents of, e.g., LONG[$m$]*LONG[$n$].
2.5.1.3. **CHARACTER operators**

Limited CHARACTER arithmetic is possible and is sometimes useful for manipulating the encodings of CHARACTER values. The following arithmetic operations are defined for operands of type CHARACTER:

A CHARACTER value plus or minus a short numeric value yields a CHARACTER value.

Subtracting two CHARACTER values yields an INTEGER value.

No other arithmetic operations on characters are allowed. Since the results of character arithmetic depend upon details of the character encoding, such arithmetic should be used with discretion.

Examples:

```plaintext
c: CHARACTER; digit: INTEGER;
digit = c - '0';
c + 'A' + (c'a)  -- assumes c is lower case
```

2.5.2. **Relational operators**

The relational operators include = and ≠, <, <= (less than or equal), > (greater than or equal), ≥, and their negatives (e.g., NOT<, ~<, ~>=. etc.). These operators always yield BOOLEAN results, depending on the truth or non-truth of the relation expressed. The operators = and ≠ apply to most types; the others, to any ordered type (i.e., to any type whose values are considered to be ordered). Ordered types include INTEGER, LONG INTEGER, CARDINAL, LONG CARDINAL, REAL, BOOLEAN, CHARACTER (with the ordering given in appendix C), enumerated types (section 3.1), and subranges of ordered types (section 3.1).

The relational operators also include the composite operator IN, which takes a numeric value as its left operand and an interval as its right operand. Its value is TRUE if the left value lies in the interval and FALSE otherwise. The syntax for relational operators is

```
Relation ::= Sum | Sum RelationTail
RelationTail ::= RelationalOperator Sum
               | Not RelationalOperator Sum
               | IN SubRange
               | Not IN SubRange
RelationalOperator ::= < | <= | = | ≠ | > | ≥
Not ::= ~ | NOT
SubRange ::= SubRangeTC | . . . -- explained in chapter 3
SubRangeTC ::= Interval | . . . -- explained in chapter 3
Interval ::= [ Expression .. Expression )
            ( Expression .. Expression )
            ( Expression .. Expression ]
            [ Expression .. Expression ]
```

The extra syntax for SubRange and SubRangeTC is placed here to be consistent with later uses of the class Interval in chapter 3. The syntax for intervals follows mathematical notation; a square bracket indicates the inclusion of the respective end point in the interval, while a parenthesis indicates its exclusion. For example, the following intervals all designate the range from 1 to 5 inclusive:

```
[1 .. 5]  [1 .. 6)  (2 .. 6)  (2 .. 5]
```
In the above examples, 1 is the lower bound of each interval; the upper bound is 5. The bounds of an interval are its end points, regardless of whether the interval is written as a closed, half-open or open one. The bounds are not required to be constants. An interval with an upper bound less than its lower is said to be empty; no values lie in such an interval. For example, the following are all empty intervals:

\[ [1 .. 2] \quad [1 .. 1) \quad (2 .. 1) \quad (2 .. 2] \]

Examples:

Relations:

\[
\begin{align*}
& n = 15 \\
& m \not= n \quad -- \ or \ m = n \\
& i \leq j \\
& (i < j) = (j < k) \quad -- \ with \ two \ BOOLEAN \ operands \\
& n \ in \ [1 .. 5) \quad -- \ n \geq 1 \ and \ n < 5 \\
& i \ not \ in \ [-1 .. 5] \quad -- \ only \ legal \ if \ i \ is \ signed \ (because \ -1 \ is)
\end{align*}
\]

A fine point:

The relational operators, like the arithmetic operators, denote families of hardware operations when they have numeric operands. Again, there is one operation for each numeric type. If there is a unique "smallest" type to which all the operands conform, they are converted to that type as necessary and then the comparison is performed. There is no unambiguous choice of such a type for numeric operands with different representations; an attempt to compare two such values is an error. The precise rules appear in section 3.5.

2.5.3. BOOLEAN operators

The operators NOT (logical negation), AND and OR apply only to BOOLEAN values. The syntax is

\[
\begin{align*}
\text{Negation} & \quad ::= \ Relation \ | \ Not \ Relation \\
\text{Conjunction} & \quad ::= \ Negation \ | \ Conjunction \ AND \ Negation \\
\text{Disjunction} & \quad ::= \ Conjunction \ | \ Disjunction \ OR \ Conjunction
\end{align*}
\]

NOT negates the logical value of a BOOLEAN expression. \( p \ AND \ q \) has the value TRUE if and only if both \( p \) and \( q \) are TRUE. \( p \ OR \ q \) is TRUE if at least one of \( p \) or \( q \) is TRUE.

When evaluating a Boolean expression, evaluation of primaries is guaranteed to take place from left to right. In the operation AND or OR, the second operand is evaluated only if the first operand's value does not determine the value of the expression.

A fine point:

"\( x \ AND \ y \)" is equivalent to the IfExpr "IF \( x \) THEN \( y \) ELSE FALSE"; i.e., when \( x \) is FALSE, \( y \) is not evaluated.

"\( x \ OR \ y \)" is equivalent to the IfExpr "IF \( x \) THEN TRUE ELSE \( y \)"; i.e., when \( x \) is TRUE, \( y \) is not evaluated.

It is therefore safe to have expressions of the form "\( x \ AND \ y \)" where \( y \) is defined only when \( x \) is TRUE, e.g., "\( x \# 0 \ AND \ c/x > 2 \)" or "\( p=\text{NIL} \ OR \ p/f=0 \)".

Examples:

Negations:

\[
\begin{align*}
& \text{NOT} \ i = 15 \quad -- \ same \ as \ \text{NOT}(i = 15) \\
& \sim q \quad -- \ q \ must \ be \ of \ type \ BOOLEAN \\
& \sim(p \ AND \ q)
\end{align*}
\]

Conjunctions:

\[
\begin{align*}
& i \leq j \ AND \ j < k \\
& p \ AND \ \sim q \\
& i = 5 \ AND \ j \ NOT \ IN \ [-1..1]
\end{align*}
\]

Disjunctions:

\[
\begin{align*}
& m \# n \ OR \ m = 15 \\
& \sim p \ OR \ \sim q
\end{align*}
\]
2.5.4. Assignment expressions

The assignment operation can be embedded in other expression forms. When it is, the result of the operation has the type of the LeftSide and the value received by the LeftSide in the assignment. The "+-" operator has the lowest precedence of any operator. Its syntax is the same as that of the AssignmentStmt:

\[
\text{AssignmentExpr} ::= \text{LeftSide} +\text{-}\text{RightSide}
\]

If this form is used to perform multiple assignments, it is important to note that "+-" is right-associative. Thus, the assignment expression \(a+-b+b+1\) first assigns the value of \(b+1\) to \(b\) and then assigns \(b\)'s new value to \(a\).

Examples:

Assignment Expressions:

\[
\begin{align*}
m+15 \\
m+n+15 \\
m+n+n+1 & \quad \text{-- same as } m+(n+(n+1)) \\
i+(j+(j+1) \mod n)*2 & \quad \text{-- all these parentheses are necessary}
\end{align*}
\]

Rules governing assignments of numeric values when the types are not identical are summarized in section 2.4.6.

Fine point:

Because the order of evaluation of the primaries is not defined, expressions such as \("(i+j) + (j+k)\)" have unpredictable values and should not be used.

2.5.5. Operator precedence

The following table summarizes the precedences of the unary and binary operators introduced in this section. The order is from highest precedence (tightest binding of operands) to lowest; operators on the same line have the same precedence.

\[
\begin{align*}
*, \div, \mod & \quad \text{-- unary negation} \\
+,- & \quad \text{-- subtraction} \\
=, \# , <, <\!, >, >\!\!, \in & \\
\neg, \text{ NOT} & \\
\text{AND} & \\
\text{OR} & \\
+ & \\
\end{align*}
\]

Parentheses can be used to explicitly control the association of operands with operators.

2.6. Initializing variables in declarations

A variable may be given an initial value in a declaration. For example, the Boolean variable delimited could be set initially FALSE by using the declaration:

\[
\text{delimited: BOOLEAN} \leftarrow \text{FALSE;}
\]
Variables (of the same type) can be initialized collectively:

\[ n, n0 : \text{INTEGER} \leftarrow 7; \]

This declares two separate integer variables \( n \) and \( n0 \) and initializes each to 7.

Any expression that could be used as the RightSide of an assignment can be used to initialize a variable:

\[ i : \text{INTEGER} \leftarrow \text{ABS}[n]; \quad \text{-- this will set } i \text{ to 7} \]
\[ i\text{Squared} : \text{INTEGER} \leftarrow i^2; \quad \text{-- } i\text{Squared} \text{ is initialized to 49} \]
\[ j : \text{INTEGER} \leftarrow i\text{Squared}i + 1; \quad \text{-- } j \text{ is initialized to } 491 + 1 = 43 \]

All initializations shown so far have taken "assignment" (or "\( \leftarrow \)"") form. There is another form, the "fixed" (or "\( = \)"") initialization. For example,

\[ \text{octalRadix} : \text{INTEGER} \leftarrow 8; \]

This means that \( \text{octalRadix} \) is to have a fixed value. It is never valid as the LeftSide of an assignment. We call \( \text{octalRadix} \) a constant because its value can never change after it is initialized (recall that the number 8 is called a literal). Normally, the term "constant" will include the term "literal"; if the distinction is important, then "literal" will be used.

Initial values for fixed initialization can be arbitrary expressions. Paraphrasing the earlier example:

\[ i0 : \text{INTEGER} \leftarrow \text{ABS}[\text{octalRadix}]; \quad i0\text{Squared} : \text{INTEGER} \leftarrow i0^2; \]
\[ j0 : \text{INTEGER} \leftarrow i0\text{Squared}i0 + 1; \]

The initializing expression can use values that are not known at compile time. In this example, if \( \text{octalRadix} \) did not have fixed initialization, the values of \( i0, i0\text{Squared}, \) and \( j0 \) would be computed and assigned at run-time. Variables are initialized in the order of appearance in a declaration, and later declarations can use variables initialized earlier, as shown by the example.

\subsection*{2.6.1. Compile-time constants}

Wherever possible, the Mesa compiler evaluates expressions containing only constants. If a variable is initialized using the fixed form and the expression can be evaluated at compile time, then that variable has a known value. Since it can never appear as the LeftSide of an assignment operator, it too becomes a compile-time constant (the variables \( i0, i0\text{Squared}, \) and \( j0 \) in the previous section are all compile-time constants).

Example:

\[ \beta : \text{INTEGER} = 3; \]
\[ \alpha : \text{INTEGER} = \beta1; \]

In this case, \( \alpha \) is a compile-time constant (with the value 2), since the expression \( \beta1 \) involves only compile-time constants. Compile-time constants need not occupy memory at run-time; the compiler can replace references to compile-time constants, such as \( \alpha \) and \( \beta \), by their known values.

Some fine points:

Knowledge of compile-time constant values can also be exploited when analyzing expressions, processing other declarations, or generating object code.

One side effect of this propagation of constants is that the representation of a numeric constant is known at compile-time. For instance, \( \alpha \) above is declared to be an \text{INTEGER}, but because its value is 2, it may also be used as a \text{CARDINAL}. However, declaring the type of \( \alpha \) determines what kind of arithmetic (signed or unsigned) will be used to compute its value, whether at compile-time or run-time (section 2.5.1).
In certain contexts, an expression is required to yield a compile-time constant value. A (sub)expression denotes such a constant if all the operands are compile-time constants and the operation is not one of those listed below (current restrictions):

- Conversion of a numeric value to type REAL.
- Any arithmetic or relational operation with operands of type LONG, INTEGER, LONG CARDINAL or REAL.
- Application of any function (chapter 5) other than a built-in function.
- The @ operation (section 3.4).
- The SELECT operation (section 4.3.3).

2.7. More general declarations

Preceding sections have introduced all the syntactic components of a declaration. The general form is defined as follows:

\[
\text{Declaration} \quad ::= \quad \text{IdList} : \text{TypeSpecification} \text{ Initialization} ;
\]

For the moment, TypeSpecification is defined as one of the built-in types; chapter 3 describes other forms of TypeSpecification.

\[
\text{TypeSpecification} \quad ::= \quad \text{PredefinedType} | \ldots
\]

\[
\text{PredefinedType} \quad ::= \quad \text{INTEGER} | \text{CARDINAL} | \text{BOOLEAN} | \text{CHARACTER} | \text{LONG INTEGER} | \text{LONG CARDINAL} | \text{REAL} | \text{STRING} | \text{WORD} | \text{UNSPECIFIED}
\]

An Initialization is formally defined as follows:

\[
\text{Initialization} \quad ::= \quad \text{empty} | \quad + \text{Expression} | \quad = \text{Expression} | \quad \ldots \quad \text{-- other forms are given later}
\]

Fine points:

The predefined type WORD is provided to describe values on which bit-by-bit logical operations are to be performed. Currently, it is a synonym for CARDINAL.

The predefined type UNSPECIFIED is a device for bypassing most type checking. An UNSPECIFIED value is a single machine word, and it matches the type of any object that occupies at most a single machine word, including INTEGER, CARDINAL, CHARACTER, BOOLEAN, UNSPECIFIED, STRING, and any user-defined type (chapter 3) that fits in a single machine word.

For numeric operations, its representation is similarly fluid. If a CARDINAL and an UNSPECIFIED value are the operands of some arithmetic operation, then the UNSPECIFIED value is considered to be unsigned. If an UNSPECIFIED is combined with a signed value, it is treated as if it were signed too. If an UNSPECIFIED is combined with an UNSPECIFIED, they are both treated as signed.

Less type checking is sacrificed by using LOOPHOLE (section 3.5.1) than by declaring variables with type UNSPECIFIED.
CHAPTER 3.

COMMON CONSTRUCTED DATA TYPES

Mesa encourages the programmer to augment the collection of predefined types by constructing new types. Types can be defined to describe objects that are structured collections of related values (e.g., a vector of Booleans, a table, or a complex number consisting of real and imaginary components). Mesa's type system has other, perhaps less obvious applications. These include expressing some of the programmer's knowledge about a class of variables (e.g., that all take on values restricted to some known interval), separating variables with different semantics into different classes so that they cannot be confused (e.g., to avoid "comparing apples and oranges"), and hiding implementation details of abstractions (e.g., to prevent the user of a table-lookup package from depending upon the internal organization of the table).

Programmer-created types have the same status as Mesa's built-in types. They can be used to declare variables and to construct further new types. In addition, values of most constructed types can be operands of the fundamental operations (+, =, #).

A new type identifier is declared using the following syntax:

\[
\text{TypeDeclaration} ::= \text{idList} : \text{TYPE} = \text{TypeSpecification} ;
\]

Each identifier in the idList is thereby declared to name the type denoted by the TypeSpecification. If this declaration form is compared to a normal declaration, i.e.,

\[
\text{Declaration} ::= \text{IdList} : \text{TypeSpecification} \text{Initialization} ;
\]

it can be seen that "TYPE" fills the role of a TypeSpecification, and " = TypeSpecification" plays the role of Initialization. In fact, the newly declared identifier has type "TYPE" and a value (which must be constant, hence the " = ") that is a TypeSpecification.

Any predefined Mesa type (section 2.7) is a valid TypeSpecification; thus the following are valid type declarations:

- \text{SignedNumber}: \text{TYPE} = \text{INTEGER};
- \text{UnsignedNumber}: \text{TYPE} = \text{CARDINAL};
- \text{TruthValue}: \text{TYPE} = \text{BOOLEAN};
- \text{Char}: \text{TYPE} = \text{CHARACTER};

These type identifiers are now valid type specifications and can be used to declare variables:

- \( i, j \): SignedNumber;
- \( n \): UnsignedNumber;
- \( b \): TruthValue;
- \( c \): Char;

After this series of declarations, \( i \) and \( j \) have type SignedNumber, which is equivalent to integer; \( n \) has type UnsignedNumber, which is equivalent to Cardinal; etc. This is a trivial way of defining
new types. A more interesting way uses a type constructor as the TypeSpecification and generates a truly new type, not just an additional name for an existing one. A TypeSpecification can be defined as

\[
\text{TypeSpecification} ::= \text{PredefinedType} \mid \text{Typeldentifier} \mid \text{TypeConstructor}
\]

(TYPE itself is not a TypeSpecification; it can be used only to declare types.)

There is an important point worth emphasizing here. A TypeSpecification that is a PredefinedType or a Typeldentifier denotes an existing type and yields the same type every time it is used. A declaration such as the one of SignedNumber introduces a synonym for the name of an existing type. Synonyms can be more descriptive and thus improve readability, but they do not partition the set of values. The types SignedNumber and INTEGER are fully equivalent, and values with these types can be used interchangeably. On the other hand, a TypeConstructor constructs a new type. The rules for equivalence and conformance of constructed types depend upon the forms of their constructors and are discussed as the constructors are introduced. In some cases, each appearance of a constructor generates a unique type, i.e., writing the same sequence of symbols twice generates two distinct, incompatible types. For this reason, programmers usually should name such a type, using a TypeDeclaration, and thereafter use the type's identifier. Of course, introducing an identifier for a constructed type can make a program easier to read and modify in any case.

The predefined types are described in chapter 2 (except for STRING in chapter 6 and process related types in Chapter 10). The simplest form of a Typeldentifier is given by

\[
\text{Typeldentifier} ::= \text{identifier} \mid \ldots
\]

-- which is a declared type

-- other forms given in chapters 6 and 7

The rest of this chapter discusses the attributes and uses of some common constructed types: enumerations, subranges, arrays, records, and pointers. The syntax for TypeConstructor is

\[
\text{TypeConstructor} ::= \text{EnumerationTC} \mid \text{SubrangeTC} \mid \text{ArrayTC} \mid \text{RecordTC} \mid \text{PointerTC} \mid \text{LongTC} \mid \text{ProcedureTC} \mid \text{ArrayDescriptorTC} \mid \text{RelativeTC} \mid \text{SignalTC} \mid \text{PortTC} \mid \text{ProcessTC}
\]

-- for enumerations

-- for subranges

-- for arrays

-- for records

-- for pointers

-- for long pointers, etc

-- see chapter 5

-- see chapter 6

-- see chapter 6

-- see chapter 8

-- see chapter 9

-- see chapter 10

(The suffix "TC" is to be understood as an abbreviation for "TypeConstructor").

Enumerations define a set of values by giving a list of identifiers. These identifiers can be viewed as members of an ordered set.

Subranges define types with values drawn from those of a larger, encompassing type but restricted to lie in a specified interval. The subrange takes on the characteristics of the enclosing type; for example, a subrange of INTEGER can be used to declare variables that behave as INTEGERS but are constrained to take values within some interval.
Arrays are sequences of components that are homogeneous with respect to type and are accessed by computed indices ("subscripting"). Records are sequences of components that have potentially different types and are accessed using fixed component names ("selection"). Records and arrays are Mesa's aggregate data types.

Pointers are scalar values used to access data objects indirectly. A pointer value is represented by an address. Pointers can be used to build linked lists, tree structures, etc. Long pointers are pointers capable of spanning a larger address space than ordinary pointers.

Chapter 3 concludes with a discussion of type determination, the process by which Mesa decides whether an expression has an acceptable type for a given operation. This is closely related to questions of the equivalence and conformance of types.

3.1. The element types

This section describes a class of types called element types. Their common properties are the following:

1. They are ordered types; values of an element type can be operands of all the relational operators (section 2.5.2).
2. They are scalar types; a value with an element type does not have any visible or directly accessible internal structure insofar as the language is concerned.
3. They can be used to declare subrange types (section 3.1.2).
4. They are the only types allowed as index types of arrays (section 3.2).

The element types are INTEGER, CARDINAL, CHARACTER, BOOLEAN, the types generated by EnumerationTC, and the types generated by SubrangeTC. Because of (3) above, this definition is recursive; subranges of subranges are allowed. The definition of the class ElementType is

```
ElementType :: = INTEGER | CARDINAL | CHARACTER | BOOLEAN |
                        EnumerationTC | SubrangeTC
```

A fine point:

Note that LONG INTEGER and LONG CARDINAL, although ordered scalar types, are not element types. It is not possible to declare subranges of these types or to use long numeric values as array indices.

3.1.1 Enumerated types

Consider the following declarations and a typical assignment:

```
channelState: INTEGER;
disconnected: INTEGER = 0;
busy: INTEGER = 1;
available: INTEGER = 2;
...
channelState ← busy;
```

Suppose channelState is a variable that is intended to range over a set of three "states" named disconnected, busy, and available, which are represented by values 0, 1, and 2. These values have no real significance; 5, 6, and 7 would serve equally well. Enumerated types are well suited to such an application (where the underlying values are unimportant). The above declarations could be replaced by a single declaration of a variable with an enumerated range:
channelState: \{disconnected, busy, available\};

... channelState \rightarrow busy;

The effect is the same as before; channelState is a variable with values ranging over the same "states", and similar assignment statements can be used.

The enumeration has some advantages over the original declarations:

It is more convenient; the programmer does not have to provide values for disconnected, busy, and available.

It allows more type checking. In the INTEGER case, one could assign any short numeric value to channelState.

It helps documentation; an enumeration shows all of its possible values.

An enumerated type is constructed by specifying a list of identifiers between braces, "\{...\}". These identifiers are not variables, but constants of that enumeration called identifier constants. They represent nothing more than their own names.

The type constructor EnumerationTC is defined as follows:

\[
\text{EnumerationTC} \quad ::= \quad \{ \text{IdList} \}
\]

The IdList supplies all the identifier constants for the enumeration, and duplication of identifiers is illegal. Separately specified enumerations are distinct. Every appearance of an EnumerationTC generates a new type that is not equivalent to, and does not conform to, any other enumeration. Thus the declarations

foreground: \{red, orange, yellow, green, blue, violet\};  
background: \{red, orange, yellow, green, blue, violet\};

specify two different enumerations. It is illegal to assign background to foreground, despite the fact that the same identifier list appears in each declaration. Occasionally, the inability to declare any further variables with the same type can be used to advantage by the programmer. Otherwise, the best way to avoid such problems is first to declare a type and then to declare variables using the identifier of that type; for example:

\[
\text{Color: TYPE} = \{\text{red, orange, yellow, green, blue, violet}\};
\]

foreground: Color;

background: Color;

This allows the assignment of background to foreground as well as the declaration of further variables with the same type (perhaps initialized differently).

The identifier constants in two different enumerated types have no association whatsoever and do not need to be distinct from one another. To identify unambiguously the enumeration from which a constant is taken, one can, and sometimes must, qualify the identifier constant by the name of the enumerated type. For example, given the additional declaration

\[
\text{Fruit: TYPE} = \{\text{orange, lemon}\};
\]

Color[orange] denotes a color and Fruit[orange] denotes a fruit. More generally, the syntax used for this form of qualification is

\[
\text{Primary} \quad ::= \quad \ldots | \text{TypoIdentifier} [ \text{identifier} ]
\]

(This adds a new case to the syntactic definition of Primary, which already allows an identifier constant.)
Often qualification is not necessary; for instance, the following is permitted:

```
hue: Color;
hue = orange;  -- the type of hue implies Color[orange]
```

In the following situations, an identifier constant need not be qualified, because the intended enumerated type is established by the context:

- as the `RightSide` of an assignment
- as an initializing `Expression`
- as a component in an array or record constructor (sections 3.2.2 and 3.3.4)
- as an argument of a procedure (chapter 5)
- as an array index (section 3.2)
- as the right operand of a `Relation`, including that part of a `Relation` used to label an arm in a discrimination (section 4.3)
- as the bounds in a `SubrangeTC` (section 3.1.2)

The values of an enumeration are ordered. The ordering is given by the order of appearance in the `IdList` used to construct the enumerated type. The leftmost identifier has the smallest value, and values increase from left to right. The following relations all have the value `TRUE`:

```
Color[red] < Color[orange]
Color[red] < violet
hue in [red .. yellow]  -- assuming hue = orange
```

There are two additional built-in functions that are applicable to enumerations: `FIRST[TypeSpecification]` yields the smallest value of the specified enumeration; e.g., `FIRST[Color] = red`. Similarly, `LAST[TypeSpecification]` produces the greatest value in an enumeration; e.g., `LAST[Color] = violet`. It is also possible to iterate over all values of an enumeration (section 4.5).

The predefined type `BOOLEAN` is really an enumerated type, and its definition is:

```
BOOLEAN: TYPE = {FALSE, TRUE};
```

Thus, `FALSE < TRUE`, `FIRST[BOOLEAN] = FALSE`, and `LAST[BOOLEAN] = TRUE`. Note, however, that the `BOOLEAN` constants `TRUE` and `FALSE` may always be used without qualification.

### 3.1.2. Subrange types

In many cases, the values of a variable are inherently range-limited. For instance, a value for `day` (of the month) lies in the range `[1..31]`. In other cases, the range is limited by design. For instance, a value for `year` might be limited to the range `[1900 .. 1999]`. Mesa permits the user to declare such variables in the following way:

```
day: CARDINAL [1 .. 31];
year: CARDINAL [1900 .. 1999];
```

Since these intervals cover a subrange of `CARDINAL`, the variables `day` and `year` are called `subrange variables`. It is useful to think of `day` and `year` as having type `CARDINAL` with the additional constraint that values are restricted to the specified intervals.
Subrange types have a number of advantages and uses. Subrange declarations unambiguously
document the range of values intended for a variable and thus aid software maintenance. The
compiler is able to optimize storage allocation when dealing with range-restricted variables (for
example, in arranging the fields of a record, section 3.3) and can take advantage of subrange
declarations to generate more efficient object code.

The general form of a SubrangeTC is

\[
\text{SubrangeTC} ::= \text{Typeldentifier} \text{ Interval} \\
\]

The Typeldentifier must evaluate to an ElementType. Thus, one can declare types that are
subranges of INTEGER, CARDINAL, CHARACTER, BOOLEAN, enumerated types, and other subrange types.
For example,

\[
\begin{align*}
\text{SymmetricRange} & : \text{TYPE} = \text{INTEGER}[1..1]; \\
\text{PositiveInteger} & : \text{TYPE} = \text{CARDINAL}[1..\text{LAST[INTEGER]}]; \\
\text{UpperCaseLetter} & : \text{TYPE} = \text{CHARACTER}['A..Z']; \\
\text{DegenerateType} & : \text{TYPE} = \text{BOOLEAN}[\text{TRUE..TRUE}]; \\
\text{CoolColor} & : \text{TYPE} = \text{Color}[\text{yellow..}\text{LAST[Color]}]; \quad \text{-- excludes red, orange, yellow} \\
\text{AthroughM} & : \text{TYPE} = \text{UpperCaseLetter}['A..M']; \quad \text{-- subrange of a a subrange}
\end{align*}
\]

The base type for a subrange is that type of which it is a subrange and which is not itself a
subrange; e.g., the base type for both UpperCaseLetter and AthroughM is CHARACTER.

The Expressions that define the end points of an interval must have types that conform to the
type denoted by the Typeldentifier (or yield short numeric values if the identifier is omitted).
Also, for the purpose of defining a subrange type, the end points must be compile-time constants.

A fine point:

It is permissible for the interval defining a subrange type to be empty. It is not legal to use a variable of such
a type, but an empty subrange is sometimes useful for specifying the bounds of an array in a record declaration
(section 3.2).

A subrange type conforms to its base type, and a base type conforms to any of its subrange types.
By extension, any two subrange types with the same base types are mutually conforming (even if they
do not overlap in any way). A more revealing point of view is that the value of a subrange
variable has the base type as its type, and an assignment of a value to a subrange variable makes an
associated assertion that the value is in the appropriate interval. A violation of such an assertion is
called a range error. It is the programmer's responsibility to guard against range errors. As implied
by this viewpoint, appropriate literals of the base type serve as literals of the subrange type, and any
operations defined on the base type automatically extend to the subrange type (but usually without
closure).

Examples:

\[
\begin{align*}
\text{n: CARDINAL [0..10]; m: INTEGER [5..5];} \\
\text{m \leftarrow 0; n \leftarrow 0;} \quad \text{-- inherited literals} \\
\text{n \leftarrow n + 1;} \quad \text{-- not valid if } n = 10 \\
\text{n \leftarrow m;} \quad \text{-- only valid if } m \text{ IN [0..5]}
\end{align*}
\]

The preceding discussion implies that subrange restrictions can be ignored in answering many type-
related questions; in this sense, subrange types are "weak." Two subrange types are equivalent if
their base types are equivalent and if the corresponding bounds are equal. For these types,
equivalence is much stronger than conformance. Equivalence becomes important when subrange
types are used in the construction of other types.
FIRST and LAST are applicable to all subrange types and yield the corresponding bound. For example, FIRST[CoolColor]=green and LAST[A through M]=’M. It is also possible to iterate over all values in a subrange (section 4.5).

A fine point:
The operators FIRST and LAST are applicable to all element types, including INTEGER, CARDINAL and CHARACTER, as well as LONG INTEGER and LONG CARDINAL. When applied to the numeric types, they supply information about the range of values supported by a particular implementation.

3.1.2.1. Subranges of numeric types *

The description above applies to subranges of both enumerated and numeric types. Numeric subranges introduce one further complication, which is the question of representation. Omission of the initial TypeIdentifier in a SubrangeTC is permissible if and only if each bound in the Interval specifies a short numeric value. In that case, INTEGER or CARDINAL is the base type, and the choice depends upon the representations of the bounds.

A numeric subrange type has a signed representation if both bounds are elements of INTEGER and at least one is not an element of INTEGER \( \cap \) CARDINAL. Similarly, it has an unsigned representation if both bounds are elements of CARDINAL and at least one is not an element of INTEGER \( \cap \) CARDINAL. If both bounds are elements of INTEGER \( \cap \) CARDINAL, values of that subrange type are considered to have both representations. Any other combination of bounds is illegal.

Examples:

\[
\begin{align*}
sl: [10..10]; & \quad \text{-- signed representation} \\
sl: [100..33000]; & \quad \text{-- unsigned representation (if 33000 > LAST [INTEGER])} \\
sl: [0..10]; & \quad \text{-- both representations}
\end{align*}
\]

With respect to the choice of signed or unsigned versions of arithmetic and relational operators, a quantity with both representations is treated flexibly. When combined with an unsigned value, it is considered to be unsigned; the unsigned operation and result are chosen. When it is combined with a signed value, the operation and result are signed. The rules governing combinations of values with both representations depend upon the context in which the result is used; the default is to choose signed representation and INTEGER operations. The precise rules are discussed in Section 3.6.

Examples:

\[
\begin{align*}
: \text{INTEGER}; \ n: \text{CARDINAL}; & \quad \text{-- plus the declarations above} \\
(\text{signed}) & \quad s1 + 1 \\
 & \quad s1 + s3 \\
 & \quad s3 \ ? \\
(\text{unsigned}) & \quad s2 + 1 \\
 & \quad s2 + s3 \\
 & \quad s3 \ ? n
\end{align*}
\]

A fine point:
The representation assumed for a literal also depends upon context. In fact, any short numeric constant \( c \) is treated as if its type were \([c..c]\).
3.1.2.2. Range assertions

Assignment to a subrange variable implies an assertion about the range of the expression being assigned. The programmer may make such an assertion explicitly, for any expression, by using a range assertion. If \( S \) is an identifier of a subrange type and \( e \) is an expression with a type \( T \) conforming to \( S \), the \textbf{Primary} \( S[e] \) has the same value as \( e \) and is additionally an assertion that \( e \) in \([\text{FIRST}[S \cap T], \text{LAST}[S \cap T]) \) is true. In addition to user defined types, the basic types \texttt{INTEGER} and \texttt{CARDINAL} may be used in range assertions.

A program that violates one of its range assertions is in error. In addition to providing documentation and (optional) run-time checking, a subrange assertion affects the attributes attached to an expression. For example, an assertion of an \texttt{INTEGER} range (or a signed subrange) forces the result to be treated as a value with signed representation. This is useful for controlling the choice of an operation when the intended one cannot correctly be inferred from the operands (section 3.6).

Examples:

\[
i: \text{INTEGER}; n: \text{CARDINAL}; S: \text{TYPE} = [0..10];
\]
\[
\text{CARDINAL}[i] \quad \text{-- i is asserted to be nonnegative}
\]
\[
S[n] \quad \text{-- asserts n in [0..10]}
\]

3.2. Arrays

Arrays are indexable collections of homogeneous components. In other words, the components of a given array all have the same type, and each corresponds to one index value in a range of indices associated with that array. The range of indices (which is actually a type called the \textit{index type}) and the component type determine the array type. For example:

\[
\text{earningsPerQuarter: ARRAY}[1..4] \text{OF INTEGER};
\]

declares a variable with a constructed array type having an index type of \([1..4]\) and a component type of \texttt{INTEGER}. Thus, \texttt{earningsPerQuarter} is an array of four integer elements: \texttt{earningsPerQuarter}[1], \texttt{earningsPerQuarter}[2], ..., \texttt{earningsPerQuarter}[4]. \texttt{earningsPerQuarter} by itself refers to the entire array variable. (Aggregate variables and components of aggregates are generally called "variables". If a distinction is needed, the term \textit{component} is used and always means an item contained within an aggregate.)

An index type must be an element type (other than \texttt{INTEGER} or \texttt{CARDINAL}). A one-to-one correspondence exists between the components of an array and the values of the index type. This allows array elements to be accessed via "indexed references". An \textit{indexed reference} selects and accesses the component corresponding to a particular index value. In its simplest form, it consists of the name of an array followed by a bracketed \textit{Expression} with a type conforming to the array's index type.

An index type can be specified using a type identifier:

\[
\text{Quarter: TYPE} = [1..4];
\]
\[
\text{profit, loss, earnings: ARRAY Quarter OF INTEGER;}
\]
\[
\text{thisQuarter: Quarter;}
\]
\[
\text{earnings[thisQuarter] \text{+ profit[thisQuarter]} \text{ loss[thisQuarter]}};
\]

The arrays \texttt{profit}, \texttt{loss}, and \texttt{earnings} have \texttt{Quarter} as their index types, and \texttt{thisQuarter} is a subrange variable with type \texttt{Quarter}. 
Index types may also be enumerations or subranges thereof. For example,

\[\text{CallType: } \text{TYPE} = \{\text{longDistance, tieLine, toll, local, inPlant}\}\]

\[\text{nearbyCalls: ARRAY CallType[toll..inPlant] OF CARDINAL;}\]

\[\ldots\]

\[\text{nearbyCalls[local]} \leftarrow \text{nearbyCalls[local]} + 1;\]

Components may be of any desired type. In particular, the component type may itself be an array type. This allows an approximation of multidimensional arrays, which are otherwise absent in Mesa. For example, a two-dimensional data structure can be declared and used as follows:

\[\text{Matrix3by4: } \text{TYPE} = \text{ARRAY[1..3] OF ARRAY[1..4] OF INTEGER;}\]

\[\text{mxy: Matrix3by4;}\]

\[\ldots\]

\[\text{mxy[3][4]} \leftarrow 0; \quad -- \text{clear last component.}\]

In the assignment statement, \(\text{mxy}\) is an expression of array type (with index type \([1..3]\) and component type \(\text{ARRAY[1..4] OF INTEGER}\)). \(\text{mxy[3]}\) is an indexed reference to the third component of \(\text{mxy}\). This in turn yields an expression of array type (with index type \([1..4]\) and component type \(\text{INTEGER}\)). Thus, \(\text{mxy[3][4]}\) is an indexed reference to the fourth component of that subarray. Because of left-associativity, \(\text{mxy[3][4]}\) is the same as \((\text{mxy[3]})[4]\).

An array constructor consists of an optional type identifier followed by a list of values (syntactically, expressions) enclosed in brackets. The list specifies values for components of an array in index order. The declaration below uses an array constructor to initialize an array that can be used as a translation table; i.e., \(\text{octalChar[n]}\) holds the character denoting octal digit \(n\):

\[\text{octalChar: ARRAY[0..7] OF CHARACTER = \{'0', '1', '2', '3', '4', '5', '6', '7'\};}\]

Note that the number of values in the list (eight) matches the number of indices in the index type. This is required for array constructors. A special form using the replicator \(\text{ALL}\) is available for abbreviating array constructors in which all components have the same value. For example, the following two declarations are equivalent:

\[\text{dashes: ARRAY[0..7] OF CHARACTER + \{'-', '-', '-', '-', '-', '-', '-'\};}\]

\[\text{dashes: ARRAY[0..7] OF CHARACTER + ALL \{'-'\};}\]

Array variables may also be initialized using other array values. Consider the following example:

\[\text{freshVector: ARRAY[0..3] OF CARDINAL = ALL[0];}\]

\[\text{currentVector: ARRAY[0..3] OF CARDINAL + freshVector;}\]

In this case, \(\text{currentVector}\) is initialized with \(\text{freshVector}\)'s value, i.e., all three of \(\text{currentVector}\)'s elements are initially set to zero.

When the operands of any of the fundamental operations \((+, -, \#)\) are arrays, the operation is applied on a component-by-component basis. The initialization of \(\text{currentVector}\) above uses assignment in this way. Similarly, the expression "\(\text{currentVector = freshVector}\)" yields the result \(\text{TRUE}\) if and only if all three components of each array are equal (as they are in the above example). Because the declaration of \(\text{freshVector}\) uses fixed initialization, assignment either to the entire array or to one of its elements is illegal.
3.2.1. Declaration of arrays

Arrays are declared using the array type constructor, ArrayTC:

\[
\begin{align*}
\text{ArrayTC} &::= \text{PackingOption} \ \text{ARRAY} \ \text{IndexType} \ \text{OF} \ \text{ComponentType} \\
\text{PackingOption} &::= \text{empty} \ | \ \text{PACKED} \\
\text{IndexType} &::= \text{ElementType} \ |
\text{Typeldentifier} \\
\text{ComponentType} &::= \text{TypeSpecification}
\end{align*}
\]

Two array types are equivalent if both their index types and their component types are equivalent and if they are both packed or both unpacked (see below). An array type conforms to another if the two types are equivalent. Thus it is possible to assign or compare array variables with separately constructed types if those types are structurally identical (see the assignment to currentVector above).

A fine point:

In addition, one array type freely conforms to another if the component type of the first freely conforms to that of the second, the index types are equivalent, and they are both packed or both unpacked (see section 3.5).

Declarations of initialized array variables take the form

\[
\text{IdList} : \text{ArrayTC Initialization}
\]

The initializing expression must have an array type conforming to the one being declared.

The previous section describes indexed references to array components. A formal definition follows:

\[
\begin{align*}
\text{IndexedReference} &::= \text{Variable}[\text{Expression}] \\
& \quad \text{(Expression)}[\text{Expression}] \\
\text{LeftSide} &::= \ldots | \text{IndexedReference}
\end{align*}
\]

The Variable or the parenthesized Expression must be of some array type, and the bracketed Expression must conform to the index type for that array type. An IndexedReference is itself part of the definition of a Leftside (and therefore of a Variable, section 2.5).

Some fine points:

Unless an array is packed, each component is "aligned", i.e., begins on a word boundary. Currently, a byte is the smallest unit into which the elements are packed. Thus a packed array of CHARACTER wastes no space, but a packed array of BOOLEAN has considerable overhead.

Since packed array elements are not necessarily word aligned, one cannot use the @ operator (section 3.4) to generate the address of an element.

The length of an array is the number of its elements. For variables with an array type, the length is fixed and known at compile-time. (Dynamic arrays are possible in Mesa through the use of array descriptors, discussed in section 6.2.1.)

The IndexType of an array may legally be an empty interval. In this case, no storage is allocated for the array. This is useful when the array appears as the last component of a MACHINE DEPENDENT RECORD (section 3.3) and the user will be obtaining storage for each record plus some number of array elements from a free storage manager. Note that \([0..0)\) is not equivalent to \([1..1),\) since the intervals specify different initial indices for the array.

Three function-like operators are relevant to arrays (and more relevant to array descriptors): LENGTH, BASE, and DESCRIPTOR. These are discussed in section 6.2, but a brief summary is provided below. For this summary, \(\text{arg}\) denotes an expression with some array type.

\[
\begin{align*}
\text{LENGTH} [\text{arg}] &\quad \text{-- yields the number of array elements.} \\
\text{BASE} [\text{arg}] &\quad \text{-- yields a pointer value for locating the first array element.} \\
\text{DESCRIPTOR} [\text{arg}] &\quad \text{-- yields arg's array descriptor value (consisting of base and length).}
\end{align*}
\]
3.2.2. Array constructors

In the preceding examples, array constructors are used only for initialization. Actually, constructors for arrays may be used in any **RightSide** context. An array constructor is defined as follows:

```
Primary ::=
  Constructor [ ... ]
Constructor ::= OptionalTypeld [ ComponentList ] | ALL [ Component ]
OptionalTypeld ::= Typeldentifier | empty
ComponentList ::= PositionalComponentList [ ... ]
  -- other forms for record constructors
PositionalComponentList ::= Component [ PositionalComponentList , Component ]
Component ::= empty | -- elided component
  Expression | NULL
```

The **empty** components in a constructor are said to be **elided**, and **NULL** components are said to be **voided**. The values of both elided and voided components are undefined. In the first form of array constructor, the number of **Expressions** plus elided or voided components must match the length implied by the array type. The type of each **Expression** must conform to the array's component type. The expressions (and elided or voided components) are taken in order to form a sequence that is the constructed array value.

Consider the following example:

```
Triple: TYPE = ARRAY [1..3] OF CARDINAL;
  triplet: Triple <+ [11, 12, 13];
```


When the array type is implied by context, the **Typeldentifier** may be omitted (see the discussion of record constructors, section 3.3.4). Thus the declaration above could be written as

```
triplet: Triple <+ [11, 12, 13];
```

Taken out of context, the constructor `[11, 12, 13]` is ambiguous; it could be assigned to any array of three numeric elements; for example:

```
trio: ARRAY {Patty, Laverne, Maxine} OF LONG INTEGER <+ [11, 12, 13];
```

The second form of constructor, using **ALL**, is only valid when the array type is implied by context. The type of the **Expression** must conform to the array's component type. The value of the constructor is an array in which the specified value is replicated a number of times equal to the length of the array. The expression is evaluated just once. In the case of an array of arrays, the structure must be mirrored by nesting in the constructor, as shown by the following example:

```
allOnes: Matrix3by4 <+ ALL [ALL [1]];   
```

Some fine points:

The value of an elided or voided component of an array constructor is not defined, but it will have some value. In particular, if the statement

```
triplet <+ [1, . 3];
```

is executed after the previous assignment to `triplet`, the value of `triplet[2]` is undefined.

Any array constructor in which all components are compile-time constants is a compile-time constant. Also, selection from an array that is a compile-time constant using a constant index yields a compile-time constant.
3.3. Records

A record is an aggregate that allows a group of related data items of different types to be packaged together. In the definition of a record type, the type of each individual component must be supplied, as in the following example:

```plaintext
MilitaryTime: TYPE = RECORD [hrs: [0..24], mins: [0..60]];
oldTime, newTime: MilitaryTime;
```

Here, MilitaryTime is a newly defined type, and oldTime and newTime are record variables of that type. MilitaryTime is a two-component record type, where the first record component is named hrs and the second mins. Every MilitaryTime record contains both components, but different record objects have their own values for these components.

A constructor of a record type contains a field list after the word RECORD. Each element in the list specifies one (or more) components of the record. For MilitaryTime, the field list is [hrs: [0..24], mins: [0..60]]. The component names, hrs and mins, are called field names. They are used to refer to components in any MilitaryTime record. For instance, the first component of oldTime may be selected using the qualified reference, "oldTime.hrs".

One can construct an entire record value using a record constructor. For instance, the constructors below yield MilitaryTime values with hrs components that have the value 13 and mins components that have the value of the expression "y+1":

```plaintext
MilitaryTime[13, y+1]
MilitaryTime[hrs: 13, mins: y+1]
```

The second constructor is an example of a keyword constructor, since it specifies the name of the component (e.g., as "hrs") with which a value is to be associated.

A default value can be specified for any field in the definition of a record type. The default is used in constructing records of that type when no value is specified in the constructor. Defaults are useful for suppressing detail and ensuring initialization of fields. In the following example, the two constructors have the same value:

```plaintext
Datum: TYPE = RECORD
    [value: INTEGER,
     nReads: CARDINAL + 0,
     nWrites: CARDINAL + 1
    ];

Datum[x]
Datum[value: x, nReads: 0, nWrites: 1]
```

The basic operations on (non-variant) record values include the fundamental operations (=, #, +), as well as qualification and extraction for accessing the record's components.

3.3.1. Field lists

There are two kinds of field lists, depending on whether the fields are "named" or "unnamed". (Field lists used to construct multi-component record types are almost always named).

Syntax equations:

```plaintext
FieldList ::= [UnnamedFieldList] | [NamedFieldList]
UnnamedFieldList ::= TypeSpecification |
    TypeSpecification , UnnamedFieldList
```
NamedFieldList ::= IdList : FieldDescription DefaultOption |
NamedFieldList, IdList : FieldDescription DefaultOption
FieldDescription ::= TypeSpecification
DefaultOption ::= empty | ~ DefaultSpecification -- section 3.3.5

Examples:
[i: INTEGER, b: BOOLEAN, c: CHARACTER] -- a named field list
[INTEGER, BOOLEAN, CHARACTER] -- a similar, but unnamed field list
[i1: CHARACTER, i2, i3: INTEGER] -- components listed and declared together
[i1: CHARACTER, i2: INTEGER, i3: INTEGER] -- equivalent to the above

Note that if one field is named, all must be named. Also, field names must be unique within a given field list. (The same identifiers may be used as field names in other field lists, however, or as names of declared variables.)

Field descriptions in a named field list contain a type specification, indicating the type of the field. Any type may be specified, including an array type or (some other) record type.

Some fine points:
A field's type specification must not imply an infinite nesting of records. For instance, the following is illegal:
A: TYPE = RECORD [k: B];
B: TYPE = RECORD [a: A];

Field lists occur in constructors of types other than records, such as PROCEDURES (chapter 5), SIGNALs (chapter 8), and in variant record specifications (chapter 6).

Unnamed field lists are normally used when component names would be ignored if they were present. This is common for functions that return single-component results. Unnamed field lists are sometimes used in specifying the input parameters for procedure variables that are to be set to one of several actual procedures. (However, an unnamed field list does not allow Calls using such a procedure variable to refer to the parameters by name.)

3.3.2. Declaration of records

The type constructor RecordTC is defined as follows:
RecordTC ::= RECORD FieldList |  
-- plus variant records (chapter 6)

where FieldList is defined in the previous section. Separately declared record types are unique, even if they look the same. Every appearance of a record constructor creates a new type that is not equivalent to, and does not conform to, any other record type. In the example:
RecType1: TYPE = RECORD [a,b: INTEGER];
rec1: RecType1;
RecType2: TYPE = RECORD [a,b: INTEGER];
rec2: RecType2;
rec3: RECORD [a,b: INTEGER];
rec4: RECORD [a,b: INTEGER];

the record variables rec1, rec2, rec3, and rec4 all have different, non-conforming types. None of these can be assigned to any of the others (despite the similarity of their components). It is, of course, legal to assign to a component any value with a conforming type. For example:
rec1.a + rec2.b + rec3.a + 5;
rec4.a + rec1.a; rec4.b + rec1.b;
Any single-component record type conforms to the type of its single component, but not vice versa. The automatic conversion in this case requires no computation.

Example:

```
Bundle: TYPE = RECORD [value: INTEGER];
recVar: Bundle;
intVar: INTEGER;
...
intVar ← recVar;  -- means intVar ← recVar.value
intVar ← recVar+1;  -- operand conversion
recVar ← Bundle[intVar];  -- a constructor
recVar.value ← intVar;
```

This conversion simplifies dealing with functions that return single-component records (chapter 5). It also provides a way of partitioning a set of variables that can be checked by the type system. In the example above, a direct assignment of intVar to recVar is invalid. Furthermore, no other single-component record type, such as

```
Prime: TYPE = RECORD [value: INTEGER];
```

can be confused with Bundle; assignment of a Bundle value to a Prime, or a Prime to a Bundle, is illegal. Either a Bundle or a Prime can, however, appear as a numeric operand. Defining Bundle and Prime as synonyms for INTEGER would not provide this additional checking.

Because of the uniqueness of constructed record types, record variables are typically declared in two steps: first the record type, then the record variables. The general form is:

```
identifier : TYPE = RecordTC;  -- define record type.
IdList : identifier Initialization;  -- same identifier as just defined
```

Record variables can also be declared directly:

```
IdList : RecordTC Initialization ;
```

This form is not very useful because the (unnamable) record type is not available for purposes such as declaring other records of the same type or writing constructors.

The Initialization shown in these general forms applies to the entire record variable, not to individual components. Any Initialization must have the proper record type. Initialization of record variables is shown in the next example.

```
noon: MilitaryTime = [hrs:12, mins:0];
midnight: MilitaryTime = [hrs:0, mins:0];
time: MilitaryTime = midnight;  -- start time at midnight.
```

Some fine points:

The Mesa compiler packs record components into machine words. The components may be arranged in an order that differs from the left-to-right order of the fields in the type constructor. All records of the same type have the same component arrangement.

Normally, the user is unconcerned with the actual arrangement of record components. When component arrangement is important, the user may specify "MACHINE DEPENDENT" records. An example is:

```
InterruptWord: TYPE = MACHINE DEPENDENT RECORD
[
   device: DeviceNumber,
   channel: [0..7],
   stopCode: [finishedOk, errorStop, powerOff],
   command: ChannelCommand
];
```
In this case, the user takes full responsibility for component arrangement. Components are positioned exactly as given, from left to right in machine words. In general, "fill" components are needed to ensure that no field crosses a word boundary (unless it starts on one). Components (such as ChannelCommand) may themselves be aggregates occupying more than one word.

It is also the user's responsibility to "fill out" the record to a full word if the record crosses a word boundary. InterruptWord might be correct for a 16-bit machine, but not for a machine having a larger word length.

Except in MACHINE DEPENDENT records, components are packed for storage efficiency. Some fields may be aligned (to the beginning of a word boundary) and some may not. Components occupying a full word or more are always aligned: arrays, INTEGERS and pointers, for example. Subrecords may or may not be aligned, depending on their size. Packed arrays are always aligned, even if there would have been space in the preceding word for a byte-sized element.

The function-like operator SIZE is often used to find the number of machine words occupied by a record of some type. The general form is: SIZE [TypeSpecification]. The result is a CARDINAL value, the number of words required by an object with the type specified by the argument. SIZE may be used to find the number of words required for any type of object.

3.3.3. Qualified references

Qualification is used to refer unambiguously to a named component of some record. The general form (which extends the definition of a LeftSide) is

```
QualifiedReference ::= Variable . identifier | (Expression) . identifier

LeftSide ::= ... | QualifiedReference
```

The field name is said to be "qualified by" the record value (the Variable or Expression) to the left of the dot. The operator associates from left-to-right in the case of multiple qualification. For example:

```
Latitude: TYPE = RECORD [degs: [0..360], mins, secs: [0..60]];
Longitude: TYPE = RECORD [degs: [90..90], mins, secs: [0..60]];
Position: TYPE = RECORD [latitude: Latitude, longitude: Longitude];
somePosition: Position;
```

Some of the possible qualified references to components of somePosition are listed below:

<table>
<thead>
<tr>
<th>Qualified Reference</th>
<th>Refers To</th>
</tr>
</thead>
<tbody>
<tr>
<td>somePosition.latitude</td>
<td>1st sub-record</td>
</tr>
<tr>
<td>somePosition.longitude</td>
<td>2nd sub-record</td>
</tr>
<tr>
<td>somePosition.latitude.degs</td>
<td>1st component of 1st sub-record</td>
</tr>
<tr>
<td>somePosition.longitude.secs</td>
<td>3rd component of 2nd sub-record</td>
</tr>
</tbody>
</table>

The association order for qualification means that names must occur in the proper sequence; e.g., somePosition.mins.longitude is incorrect. Also, a qualified reference must be complete, i.e., names may not be skipped (as in somePosition.secs, which would be ambiguous in any event).

Qualified references and indexed references have the same precedence (the highest possible) and may be intermixed. For example:

```
recordOfArrays: RECORD [a,b: ARRAY [0..100] OF CARDINAL];
arrayOfRecords: ARRAY [1..5] OF RECORD [i1,i2,i3: CARDINAL];
arrayOfRecords[5].i3 < recordOfArrays.a[0]; -- ("last" gets "first")
```

A fine point:

Qualification briefly opens up a given "name scope". For instance, in the record qualification, rec.x, the qualified name, x, must name a field of rec and selects that field. Scope is treated more fully in chapter 7.
3.3.4. Record constructors

A record constructor assembles a record value from a set of component values. In the following example, a constructor is used as a RightSide of an assignment.

```
MonthName: TYPE = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
Date: TYPE = RECORD
    [day: [1 .. 31],
     month: MonthName,
     year: [1900 .. 2000]
    ];

birthDay: Date;
dd: [1 .. 31]; mm: MonthName; yy: [1900 .. 2000]; now: [1900 .. 2000] + 1976;
birthDay + Date[25, Apr, now33];
```

This constructor yields a record value with type Date. The record assigned to birthDay contains the following component values:

<table>
<thead>
<tr>
<th>Component</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
<td>25</td>
</tr>
<tr>
<td>month</td>
<td>Apr</td>
</tr>
<tr>
<td>year</td>
<td>now33</td>
</tr>
</tbody>
</table>

(which is 1943)

A Constructor is a Primary and may not be used as the LeftSide of an assignment.

Record constructors are of two kinds: keyword constructors and positional constructors. Within both kinds, the component value for a particular field may either be supplied or be omitted. If it is omitted, the value of the field is determined by the DefaultOption appearing in the declaration of the field (section 3.3.5).

Syntax equations:

```
Primary ::= ... | Constructor
Constructor ::= OptionalTypeld [ ComponentList ]
OptionalTypeld ::= Typeldentifier | empty
ComponentList ::= KeywordComponentList | PositionalComponentList
KeywordComponentList ::= KeywordComponent | KeywordComponentList, KeywordComponent
PositionalComponentList ::= Component | PositionalComponentList, Component
KeywordComponent ::= identifier : Component
Component ::= empty | -- elided component
             Expression | -- explicit component
             NULL | -- voided component
```

The initial Typeldentifier, if present, must name the type of the record being constructed.

In keyword constructors, the correspondence between constructor components and record components is strictly "by name". Keyword names may not be repeated in a constructor, but the order is irrelevant. For example, the following keyword constructors are equivalent:

```
Date[day: 25, month: Apr, year: now33]
Date[month: Apr, day: 25, year: now33]
```
Chapter 3: Common Constructed Data Types

All of these keyword constructors specify values for all the components. In the following example, the first keyword constructor elides the *month* component (the place for the component value is specified, but no value is given); the second voids the *month* component (by specifying `NULL` instead of a value):

```
Date[day: 25, month: , year: now33]   -- month is elided
Date[day: 25, month: NULL, year: now33]   -- month is voided
```

The distinction between an elided and a voided a field arises in the treatment of defaults (section 3.3.5). Since the declaration of `Date` specifies no default value for *month*, both of these examples construct records with a second component having an undefined value.

In a *positional constructor*, the correspondence between constructor components and record components is strictly "by position". The first constructor component corresponds to the first record component, the second value to the second component, etc. Positional constructors may be used for both records and arrays (section 3.2.2). It does not matter whether or not fields are named in the definition of the record type. The following three constructors are equivalent:

```
Date[day: 25, month: , year: now33]   -- value of *month* is undefined
Date[25, , now33]                     -- value of 2nd component is undefined (elided)
Date[25, NULL, now33]                 -- value of 2nd component is undefined (voided)
```

Positional constructors may elide or void components as shown above, but components not supplied at the end of the list must be elided by supplying a sufficient number of trailing commas.

*Keyword and positional notations may not be mixed in a single constructor.* The order of evaluation of components is not specified for either kind of constructor.

The initial *TypeIdentifier* in a constructor may be omitted when the constructor is used as:
- the RightSide of an assignment (unless the LeftSide is an extractor, section 3.3.5)
- an expression in an *Initialization*
- a component of an enclosing record or array constructor
- an argument of a procedure
- the right operand of a *Relation.*

In other cases, an initial *TypeIdentifier* must appear. It is never incorrect to supply the identifier, and sometimes doing so improves readability.

A *fine point:*

*Any record constructor in which all components are compile-time constants is a compile-time constant.* Also, a field selected from a record that is a compile-time constant is itself a compile-time constant.

### 3.3.5. Default field values

The definition of a record type may specify a default value for each field. These default specifications are optional; if present, they are used in constructing records of that type when no values for the corresponding fields are specified in constructors. An elided field, as discussed in the preceding section, supplies no value. Specifying some default treatment of a field also allows complete *omission* of that field in a constructor. In a keyword constructor, a field is omitted by omitting the keyword entirely; in a positional constructor, trailing fields (only) can be omitted by omitting the final commas. The positional constructor "[ ]" is considered to omit, not elide, its first component (if any).
In the following example, all constructors have the same value.

```
Interval: TYPE = RECORD
  [range: INTEGER, origin: INTEGER + 0, direction: {up, down} + up];
```

```
Interval[range: 10, origin: 0, direction: up] -- all fields specified
Interval[range: 10, origin: , direction: ] -- origin, direction elided
Interval[range: 10] -- origin, direction omitted
Interval[10] -- origin, direction omitted (positional form)
```

The syntax for specifying defaults in a NamedFieldList follows:

```
DefaultOption :: = empty | + DefaultSpecification
DefaultSpecification :: = empty |
Expression | NULL |
Expression | NULL
```

Note: In the final line, the vertical bar denotes itself and is embedded within an alternative.

Only named fields may have default values. In a DefaultSpecification, the Expression must have a type that conforms to the type of the corresponding field.

Suppose that $R$ is a record type with a field $v$ of type $T$. The above syntax allows five forms for the DefaultOption in the declaration of $v$. No matter which form is used, a constructor of an $R$ may explicitly specify a value for the field $v$. The various options control whether the existence of the field must be made evident in the constructor, whether an explicit value must be supplied and, if not, what action is taken. The options are interpreted as follows:

1. $v: T$
   In a constructor, the value of $v$ can be left undefined, but that must be indicated explicitly, by eliding or voiding the field. This rule also applies to an unnamed field.

2. $v: T +$
   Every constructor must supply an explicit value (not NULL) for $v$.

3. $v: T + e$
   If a constructor elides or omits $v$, the value of the expression $e$ is used; voiding the field is not permitted.

4. $v: T + NULL$
   As in (1) above, except that the constructor may omit $v$ entirely. If the field is omitted, elided or voided, its value is undefined.

5. $v: T + e | NULL$
   As in (4) above, except that a constructor may explicitly void $v$. If the field is omitted or elided, the value of $e$ is used; if it is voided, its value is undefined.

If the first or second form is used, the field cannot be omitted from a constructor; these forms are useful when such omission is likely to indicate a programming error. Omission is permitted by the other forms, which differ in the default action for an omitted or elided field. These forms are appropriate when a field has some common and meaningful default value (the third and fifth cases) or, alternatively, is relevant only in unusual circumstances (the fourth case). The last three forms are particularly suitable for extending the definition of a record type; constructors in existing programs need not be modified.
Fine points:

The second form of field declaration guarantees that the field \( v \) has a well-defined value in any record created by a constructor (but not otherwise).

If the Expression form of a default specification is used, that expression is evaluated at the time of construction but in the context of the declaration of the record, i.e., the expression is treated as a parameterless procedure invoked by evaluation of the constructor (see chapter 5).

The default value of a field cannot be specified in terms of other fields in the same record. Default values for fields of record types defined in DEFINITIONS modules (section 7.3.2) must be compile-time constants.

Examples:

\[ R: \text{TYPE} = \text{RECORD} \]
\[ \{ \]
\[ v1: \text{CARDINAL}, \]
\[ v2: \text{CARDINAL} \leftrightarrow, \]
\[ v3: \text{CARDINAL} \leftrightarrow 3, \]
\[ v4: \text{CARDINAL} \leftrightarrow \text{NULL}, \]
\[ v5: \text{CARDINAL} \leftrightarrow 5 \text{ [NULL] } \]
\[ \} ; \]

-- the following are valid
\[ R[v1: 1, v2: 2] \]
\[ R[v1: \cdot, v2: 2, v5: \cdot] \]
\[ R[v1: 1, v2: 2, v5: \text{NULL}] \]

-- the following are not valid
\[ R[\cdot] \]
\[ R[v1: 1, v2: \text{NULL}, v5: \text{NULL}] \]

3.3.6. Extractors

Extractors are used to "explode" record objects and assign their components to individual variables in a single statement. For example, the extractor below assigns the components of \( \text{birthDay} \) (defined in the previous section) to the variables \( dd, mm, \) and \( yy \), in that order:

\[ [dd, mm, yy] \leftrightarrow \text{birthDay}; \]

This has the same effect as the following three separate assignments, except that \( \text{birthDay} \) is evaluated only once:

\[ dd \leftrightarrow \text{birthDay} . \text{day}; \quad mm \leftrightarrow \text{birthDay} . \text{month}; \quad yy \leftrightarrow \text{birthDay} . \text{year}; \]

An extractor resembles a constructor in form, but there are some important differences:

An extractor may only be used as a LeftSide, never as an Expression.

The "components" of an extractor specify LeftSides, not Expressions.

Extractors always begin with a left bracket, never with a Typeldentifier.

The type of the record value assigned to an extractor must be known to the compiler. This means that the following (rather useless) statement is invalid because the constructor's type cannot be determined:

\[ [dd, mm, yy] \leftrightarrow [25, \text{Apr.} 1943]; \quad \text{-- invalid} \]

The statement should specify the type of the constructed value:

\[ [dd, mm, yy] \leftrightarrow \text{Date}[25, \text{Apr.} 1943]; \quad \text{-- valid} \]
Extractors, like constructors, may use keywords. This allows an extractor to be written without regard to the record's component order. For instance, the following statements are equivalent to the first one in this section:

\[ \text{[day: dd, month: mm, year: yy]} \mapsto \text{birthDay}; \]
\[ \text{[month: mm, day: dd, year: yy]} \mapsto \text{birthDay}; \]

Extractors may elide or omit any item, in which case the corresponding record component is not assigned. The extractors shown below are equivalent:

\[ \text{[day: dd, month: , year: yy]} \mapsto \text{birthDay}; \quad \text{-- month elided} \]
\[ \text{[day: dd, year: yy]} \mapsto \text{birthDay}; \quad \text{-- month omitted} \]
\[ \text{[dd, , yy]} \mapsto \text{birthDay}; \quad \text{-- 2nd component elided} \]

A positional extractor may omit trailing components without supplying trailing commas. The \textit{year} component of \textit{birthDay} is omitted below.

\[ \text{[dd, mm]} \mapsto \text{birthDay}; \]

An extraction operation (unlike an ordinary assignment) yields no value. \textit{This means that an extractor may not be embedded within an expression.} For example, the first statement following is illegal; the second is a valid alternative:

\[ r \mapsto \text{[x, y, z]} \mapsto s; \quad \text{-- invalid} \]
\[ \text{[x, y, z]} \mapsto r \mapsto s; \quad \text{-- valid} \]

Syntax equations:

\[
\text{AssignmentStmt} ::= \ldots \mid \text{Extractor} \mapsto \text{RightSide} \\
\text{Extractor} ::= \[ \text{KeywordExtractList} \] \\
\text{KeywordExtractList} ::= \text{KeywordExtract} \mid \text{KeywordExtract}, \text{KeywordExtractList} \\
\text{KeywordExtract} ::= \text{identifier: ExtractItem} \\
\text{PositionalExtractList} ::= \text{ExtractItem} \mid \text{ExtractItem}, \text{PositionalExtractList} \\
\text{ExtractItem} ::= \text{empty} \mid \text{component is ignored} \mid \text{LeftSide} \quad \text{-- component is assigned to LeftSide}
\]

The identifiers in a \texttt{KeywordExtractList} must be field names for the record type. Note that an extraction list can be empty, in which case the effect is to discard a record value. Extractors cannot be nested.

3.4. Pointers

Pointers allow efficient indirect access to objects. A pointer may refer to only one specific type of item. For instance, the following pointer provides access only to objects of type \texttt{INTEGER}:

\[ \text{intPtr: POINTER TO INTEGER;} \]

Another pointer might be specified to point only to \texttt{BOOLEAN} objects:

\[ \text{boolPtr: POINTER TO BOOLEAN;} \]

These are different types of pointers since they have different \textit{reference types}, \texttt{INTEGER} and \texttt{BOOLEAN}. Furthermore, since \texttt{INTEGER} and \texttt{BOOLEAN} are incompatible types, these pointer types are also incompatible; i.e., assignment of \texttt{boolPtr} to \texttt{intPtr}, or vice versa, is disallowed.
A pointer value is represented by the address of some data object. This object is called the pointer's referent. The postfix operator \( * \) may be applied to a pointer value of any type to yield that value's referent. The process of "following" a pointer to its referent is called dereferencing.

A dereferenced pointer designates a variable. When the pointer is declared as above, the variable can be used as a LeftSide or as a Primary. Thus \( \text{intPtr} \) and \( \text{boolPtr} \) are variables of type \text{INTEGER} and \text{BOOLEAN} respectively. The statement

\[
\text{boolPtr} \leftarrow (\text{intPtr} = 0);
\]

is executed by following \( \text{intPtr} \) to obtain a \text{INTEGER} value, testing that value, and assigning the result to the \text{BOOLEAN} variable referenced by \( \text{boolPtr} \).

Sometimes a pointer is created simply to identify an object or to allow indirect access to a value that is not to be modified. Mesa provides readonly pointers for such applications. A value with a readonly pointer type cannot be used to update its referent. For example, the declaration

\[
\text{ROintPtr}. \text{POINTER TO READONLY INTEGER};
\]

declares a readonly pointer. \( \text{ROintPtr} \) is a Primary with type \text{INTEGER} but not a valid LeftSide.

Any type specification is permitted as the reference type of a pointer type. The pointers declared below reference a named record type.

\[
\text{Person}: \text{TYPE} = \text{RECORD}
\]
\[
[\text{age}: [0..200],
\text{sex}: \{\text{male, female}\},
\text{party}: \{\text{Democratic, Republican}\}];
\]
\[
\text{candidate1}, \text{candidate2}: \text{Person};
\text{winner}, \text{loser}. \text{POINTER TO} \text{Person};
\]

Pointers to record objects may be used to qualify field names. If record \( \text{candidate1} \) is the referent of \( \text{winner} \), then qualifications such as

\[
\text{winner.age} \quad \text{winner.sex} \quad \text{winner.party}
\]

select the corresponding components of \( \text{candidate1} \). However, if \( \text{candidate2} \) were the referent, these same qualifications would select components of \( \text{candidate2} \). When applied to a pointer, the operation of selection implies dereferencing. For example, \( \text{winner.age} \) specifies dereferencing \( \text{winner} \) to obtain a record variable of type \text{Person} and then performing normal field selection on that record. Thus \( \text{winner.age} \) is an abbreviation of \( \text{winner.age} \).

It is common to define a record type containing components that are pointers referencing objects with the same record type. For example, the type declared as follows:

\[
\text{FamilyMember}: \text{TYPE} = \text{RECORD}
\]
\[
[\text{someone}: \text{Person},
\text{mother}, \text{father}: \text{POINTER TO} \text{FamilyMember}]
\]

might be used to create a tree of related persons in which the relations are expressed directly by pointer linkages.
The fundamental operations (=, #, +) applied to pointer values deal with the pointers themselves, not with their referents. In the examples:

\[
\begin{align*}
\text{winner} & \leftarrow \text{loser}; \\
\text{winner} & \leftarrow \text{loser};
\end{align*}
\]

the first sets \text{winner} to point to the same \text{Person} as \text{loser}; the second assigns the referent of \text{loser} to the referent of \text{winner}, and thus has a quite different effect. The full set of relational operators can be applied to pointers declared to be \text{ordered}; for example:

\[
\text{orderedPtr} : \text{ORDERED POINTER TO Person};
\]

The ordering is determined by the memory addresses that represent the pointers, not by the properties of the referents. Pointers not declared to be ordered can be only be compared using the operators = and #.

There is one pointer literal, \text{NIL}. It conforms to any unordered pointer type and denotes a pointer value that has no valid referent. For example:

\[
\text{IF intPtr} = \text{NIL} \text{ THEN boolPtr} \leftarrow \text{NIL};
\]

A pointer with value \text{NIL} should not be dereferenced; the result is undefined.

Pointer values are most commonly obtained from allocators that provide and manage storage for a class of objects. The unary prefix operator @ also generates pointers. When applied to a variable with type \(T\), it yields a pointer to that variable with type \text{POINTER TO} \(T\); for example:

\[
\text{winner} \leftarrow @\text{candidate1};
\]

Pointer generation should be done with caution; it is possible for the resulting pointer to outlive the referenced object. A non-NIL pointer value with no valid referent is said to be a dangling reference. The language does not prevent dereferencing such a pointer, but doing so produces an undefined result. \text{It is the user's responsibility to avoid dereferencing a dangling (or uninitialized) reference.}

### 3.4.1. Constructing pointer types

The type constructor for pointers is defined as follows:

\[
\begin{align*}
\text{PointerTC} & ::= \text{Ordered Base POINTER TO ReadOnly TypeSpecification} | \text{Ordered Base POINTER Interval TO ReadOnly TypeSpecification} \\
\text{Ordered} & ::= \text{empty} | \text{ORDERED} \\
\text{Base} & ::= \text{empty} | \text{BASE} \\
\text{ReadOnly} & ::= \text{empty} | \text{READONLY}
\end{align*}
\]

The \text{TypeSpecification} in a \text{PointerTC} specifies the reference type of the pointer type. Two pointer types are equivalent if their reference types are equivalent and if their attributes \text{ReadOnly} and \text{Ordered} are specified identically. Thus equivalent pointer types can be constructed in separate places, but they must have the same structure. One pointer type conforms to another if the two reference types are equivalent, if either the \text{ReadOnly} attributes are identical or the second is \text{READONLY} and the first is not, and if either the \text{Ordered} attributes are identical or the first is \text{ORDERED} and the second is not. The \text{Base} attribute is ignored in determining conformance.

In the following examples, the first type in each pair conforms to the second, but the second does not conform to the first:

\[
\begin{align*}
\text{POINTER TO FamilyMember} & \quad \text{POINTER TO READONLY FamilyMember} \\
\text{ORDERED POINTER TO Person} & \quad \text{POINTER TO Person} \\
\text{ORDERED POINTER TO Date} & \quad \text{POINTER TO READONLY Date}
\end{align*}
\]
Chapter 3: Common Constructed Data Types

Fine points:

If one pointer type conforms to another, it conforms freely (section 3.5.3). Conformance of pointer types is extended by the following rule: one pointer type conforms freely to another if the second is READONLY, the reference type of the first conforms freely to the reference type of the second, and the Ordered attributes satisfy the restriction above.

The second form of PointerTC constructs a subrange of a pointer type. Subranges of pointers have the usual properties of subranges; e.g., a pointer subrange type and its base type mutually conform (but not freely). The values of a subrange pointer are restricted to the given interval (and can potentially be stored in smaller fields). Subrange pointer types are not recommended for general use. They are intended primarily for constructing relative pointer types (section 6.3) which, unlike the subrange types, do not allow dereferencing without relocation.

The attribute BASE specifies that values with that pointer type are to be used as base values for relocating relative pointers (section 6.3). Such values may also be used as ordinary pointers.

3.4.2. Pointer operations

The general form of an indirect reference is:

\[
\text{IndirectReference} \ ::= \ Variable \dagger | (\text{Expression}) \dagger
\]

\[
\text{LeftSide} \ ::= \ldots | \text{IndirectReference}
\]

The postfix operator \(\dagger\) performs explicit dereferencing of the pointer expression it follows. Its precedence is the same as indexing and qualification (the highest possible), and these operations can be intermixed. For example:

```
... group: ARRAY [0..10] OF POINTER TO FamilyMember;
... group[i] \dagger.mothert.somone \quad -- (((group[i])\dagger.mother)\dagger).someone
```

If \(p\) is an arbitrary pointer expression, then \(pt\) can be read as "p's referent" or "referent of p". Application of the \(\dagger\) operator produces a variable that may be used as a Primary. Unless \(p\) is a readonly pointer, \(pt\) (or any of its components) may also be used as a LeftSide. The definition of conformance implies that an ordinary pointer can be assigned to a readonly pointer, but not vice versa. Thus the referent of a readonly pointer is not necessarily immutable; i.e., its value might change during the lifetime of the readonly pointer. The Mesa language only prevents updates of the object through those pointers to it that are declared to be readonly.

The syntax used for address generation is

\[
\text{Primary} \ ::= \ldots | @ \text{LeftSide}
\]

The prefix operator \(@\) produces the address of its operand. If \(x\) is a variable of type \(T\), the value of \(@x\) is a pointer to \(x\), and its type is POINTER TO \(T\). \(@x\) can be read as "address of \(x\)". The operand for \(@\) must be a valid LeftSide (it cannot be a constant or an arbitrary expression, for instance). The operator's precedence is lower than that of \(\dagger\); e.g., \(@x\dagger\) is equivalent to \(@(x\dagger)\) (or simply \(x\)).

Some fine points:

There are variables that cannot be the referents of pointers and thus cannot be the operands of \(@\). These include all "variables" with fixed initialization and components of such variables. In addition, a pointer value is represented by a word address. Therefore, a referent must lie on a word boundary; an object having this property is called aligned. Variables are aligned except in the following cases:

Elements of packed arrays are not aligned.

Any component of a record that occupies less than a single word is not aligned (but arrays, even if packed, are always aligned).
Care must be taken so that a pointer to a declared variable does not exist longer than the variable to which it points. Consider the following example (which assumes familiarity with procedures, local variables and global variables):

```plaintext
pointer1, pointer2: POINTER TO INTEGER;          -- two global variables

RiskyProc: PROCEDURE [i: INTEGER] =            -- i is a local variable
BEGIN
  local: INTEGER;                                -- and so is local
  pointer1 := @i;                                -- risky: i will disappear upon RETURN
  pointer2 := @local;                            -- also risky
  RETURN                                          -- the "risky" pointers are valid up to this point, but
  -- NOT after this statement is executed.
END;
```

After the RETURN statement is executed, local storage is released for other purposes; thus the pointers will reference unpredictable data when that storage is reused. One should use pointers with referents existing at least as long as the pointers.

Pointers that are declared to be ORDERED may be used as operands of all the relational operators (section 2.5.1). For this purpose, they behave as unsigned numeric values. The definition of conformance implies that an ordered pointer can be assigned to an unordered pointer variable, but not vice versa. NIL is not a valid ordered pointer constant, and the relation of its value to other pointer values is undefined. Also, the @ operator always produces an unordered pointer value.

The following fine points cover pointer capabilities that should be used with caution (and avoided when possible). Some of these capabilities circumvent normal type-checking, and may result in unpredictable results if used.

The type POINTER TO UNSPECIFIED (or simply POINTER) can access actual data of any type. Pointers of this type conform to any other pointer type, and vice-versa.

Limited arithmetic can be performed on pointers, but programmers are encouraged to use BASE and RELATIVE pointers (chapter 6) if the purpose of the arithmetic is simple relocation. A short numeric value added to, or subtracted from, a pointer produces another pointer with the same type. Also, the difference of two pointer values with equivalent types is a CARDINAL.

### 3.4.3. Long Pointers *

Long pointers provide indirect access to objects having memory addresses that cannot be represented within a single machine word. Like LONG INTEGERS, they are essentially concessions to the limitations of hardware. Again, the long variant provides somewhat greater generality at somewhat greater cost.

Long pointer types are constructed as follows:

```
LongTC ::= LONG TypeSpecification
```

The type constructor LONG can be applied to INTEGER (chapter 2), any pointer type, or any array descriptor type (chapter 6). No other type can be lengthened.

Long pointers are typically created by lengthening (short) pointers as described below. In particular, NIL is automatically lengthened to provide a null long pointer when required by context. The standard operations on pointers (dereferencing, assignment, testing equality, comparing ordered pointers) all extend to long pointers.

Both automatic and explicit lengthening (using the operator LONG) are provided for pointer types, and the type POINTER TO T conforms to (but is not equivalent to) the type LONG POINTER TO T. Lengthening an expression with the first of these types produces a value with the second; i.e., the reference type and the Base, Ordered and ReadOnly attributes are unchanged.
Chapter 3: Common Constructed Data Types

The operator @ applied to a variable of type $T$ produces a pointer of type LONG POINTER TO $T$ if the access path to that variable itself involves a long pointer and of type POINTER TO $T$ otherwise.

Some fine points:

Two conforming pointer types conform freely only if both are long pointers or both are not.

NIL is lengthened in a standard way and has a universal representation. All other pointers are lengthened in a hardware dependent way. There is no normalization prior to operations on long pointers; and such pointers constructed other than by lengthening may give anomalous results (e.g., in comparisons).

If either operand in a pointer addition or subtraction is long, all operands are lengthened and the result is long.

Examples:

$$R: \text{TYPE} = \text{RECORD} [\{ T, \ldots \};$$
$$p, q: \text{POINTER TO } R;$$
$$p, q: \text{LONG POINTER TO } R;$$
$$pT: \text{POINTER TO } T;$$
$$pT: \text{LONG POINTER TO } T;$$

-- the following are valid.

$$pp = qq; \quad pp = \text{NIL}; \quad pp = p;$$

-- long comparisons

$$pT = @p.f; \quad ppT = @pp.f;$$

-- pointer lengthened

-- the following are not valid.

$$pp = ppT;$$
$$p + pp; \quad pT = @pp.f;$$

-- incompatible types

-- no automatic shortening

3.4.4. Automatic dereferencing

Automatic dereferencing converts a pointer RightSide of type POINTER TO $T$ into one of type $T$ if that RightSide is followed by dot qualification (section 3.3.3), a bracketed array index, or a bracketed argument list (the last two are syntactically identical). For example, in the following two statements, the LeftSides are equivalent:

$$\text{winner.party} \quad \text{winnert .party}$$

Automatic multilevel dereferencing is possible. Given the following declarations, the three final assignment statements have the same effect:

$$\text{actualArray: ARRAY [0..20] OF INTEGER;}$$
$$\text{arrayPtr: POINTER TO ARRAY [0..20] OF INTEGER + @actualArray;}$$
$$\text{arrayFinger: POINTER TO POINTER TO ARRAY [0..20] OF INTEGER + @arrayPtr;}$$

$$\text{actualArray[1]} = 3; \quad \text{arrayPtr[1]} = 3; \quad \text{arrayFinger[1]} = 3;$$

A fine point:

The pointer attribute BASE inhibits automatic dereferencing in the context of subscript or argument brackets. See section 6.3.
3.5. Type determination

Every expression in a Mesa program has a type that can be deduced by static analysis of the program text. Such analysis is called type determination. The language imposes constraints on the type of each expression according to the context in which it is used. A program that does not violate any of these constraints is type-correct; every valid Mesa program must be type-correct.

In principle, every variable and every expression has an inherent type derived from its structure. The inherent type of a variable is established by declaration; the form of a literal implies its type, and each operator produces a result with a type that is a function of the types of the operands. Inherent types of some expression forms are listed below:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Inherent Type of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>[34..34]</td>
</tr>
<tr>
<td>NIL</td>
<td>POINTER TO UNSPECIFIED</td>
</tr>
<tr>
<td>x &lt; y</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>x</td>
<td>declared type of x</td>
</tr>
<tr>
<td>array[i]</td>
<td>type specified for the components of array</td>
</tr>
<tr>
<td>@x</td>
<td>POINTER TO type of x</td>
</tr>
<tr>
<td>(x + e)</td>
<td>type of x</td>
</tr>
</tbody>
</table>

The type rules in Mesa take two general forms, which are the following:

- The exact type required by the context is known, and a given type must conform to it. The required type is called the target type.
- The exact type required is not implied by context, but a relation that must be satisfied by a set of types is known. The process of satisfying that relation is called balancing.

Situations in which the target type is known are simpler and more common; they will be discussed first.

All assignment-like contexts establish a target type for the expression to be assigned. These contexts include not only assignment itself (where the target type is the type of the LeftSide) but also initialization, record construction (where the target type for each component expression is the declared type of the corresponding field), array construction, parameter list construction, and the like.

Example:

```mesa
LType: TYPE = RECORD[c: CType];
ivar: LType;
ivar + anyExp; -- target type of anyExp is LType
ivar + LType[c: someExp]; -- target type of someExp is CType ...
ivar.c + someExp; -- ... which is more obvious here
```

The following rule applies to assignments:

There is never any automatic dereferencing or type conversion of any kind for the LeftSide of an assignment, and the inherent type of the LeftSide is the target type of the right side. (Of course, a LeftSide may contain subexpressions, such as array subscripts, that are themselves right sides and subject to conversion.)

Certain other contexts imply a target type. For example, the target type for an array subscript is the index type of the array. Also, the target type of the expression following IF, WHILE, etc., is BOOLEAN.
If the inherent type of an expression is equivalent to the target type, the use of that expression is type-correct. If it is not equivalent, it may still be possible to obtain conformance by applying various type conversions, which are sometimes called coercions. In Mesa, there is at most one sequence of conversions that can be applied automatically to convert a value from one type to another. When implicit conversion from the inherent type to the target type is impossible, the program is in error; e.g., assigning a BOOLEAN value to an INTEGER variable is never valid.

Some fine points:

When the target type is well defined, certain expression forms may be abbreviated. Identifier constants need not be qualified, and explicit identification of the type of a constructor is optional. The abbreviated constructs have no inherent type when viewed out of context, and they cannot be used in situations requiring implicit conversion. For example,

\[ R: \text{TYPE} = \text{RECORD} [i: \text{INTEGER}]; \]
\[ v: R + [R[3]]; \quad \text{-- the second } R \text{ cannot be omitted} \]

An Extractor never has an inherent type; the extraction is controlled by the inherent type of the RightSide, which therefore cannot be abbreviated or converted. For example,

\[ r: \text{RECORD} [\text{inner: RECORD}[j7, j2: \text{INTEGER}]]; \]
\[ [i, j] + r.\text{inner}; \quad \text{-- the field selection cannot be omitted} \]

3.5.1. Type conversion

There are four automatic type conversions that can be applied to establish type conformance. All have been discussed in preceding sections. They are the following:

1. A value with a subrange type may be converted to a value with its base type, and vice versa (section 3.1.2).
2. A value with a single-component record type may be converted to a value with the type of that component (section 3.3.2).
3. A value with a short numeric, pointer or array descriptor type may be lengthened to a value with the corresponding long type (section 2.4.5).
4. A value with any numeric type may be converted to type REAL (section 2.4.5).

The first of these is a somewhat special case; as mentioned in section 3.1.2, it is more accurate to view this as a pair of conversions that are applied unconditionally when evaluating, and assigning to, a subrange variable.

Examples:

\[ r: \text{RECORD}[f: \text{INTEGER}]; \]
\[ i: \text{INTEGER}; \]
\[ ii: \text{LONG INTEGER}; \]
\[ i + r, \quad \text{-- } i \leftarrow r.f \]
\[ ii \leftarrow r, \quad \text{-- } ii \leftarrow \text{LONG}[r.f] \]

Some fine points:

A number of the conversions used to achieve conformance require computation and cannot be applied recursively to the constituents of constructed types. For example, INTEGER conforms to LONG INTEGER, but ARRAY IndexType OF INTEGER does not conform to ARRAY IndexType OF LONG INTEGER. Section 3.5.3 discusses the concept of "free" conformance and the rules governing such cases.

There is one other automatic conversion, dereferencing, that is applied only in certain syntactic contexts (section 3.4.4). It is never applied automatically to achieve type conformance in an assignment.
Sometimes it is necessary to subvert Mesa's type checking, particularly in programs that manipulate low-level representations of objects. A Primary with the form

```
LOOPHOLE [ Expression, TypeSpecification ]
```

has the same value as the Expression (viewed as a sequence of bits) and the type denoted by TypeSpecification. This "conversion" never requires any computation. The only restriction is that values with the inherent type of Expression must be represented in the same number of machine words as values of the type TypeSpecification. When the target type is well-defined, the TypeSpecification may be omitted. For example:

```
S: BOOLEAN;  N: CARDINAL;
N + LOOPHOLE[S, CARDINAL]; -- to discover the representation
N + LOOPHOLE[S]; -- also acceptable
```

Since LOOPHOLE bypasses most checking, its use should be limited as much as possible.

3.5.2. Balancing *

Many of Mesa's operators are generic; i.e., the operation performed depends upon the types of the operands. Examples are the fundamental operators = and #, which accept two operands with arbitrary (but compatible) types and produce a BOOLEAN result. In this case, neither operand has a defined target type. Instead, it is necessary to find some type to which the inherent type of each operand conforms; any automatic type conversions are applied to the operands as necessary to produce values of that type; and the operation is then performed. The common type is the "least upper bound", i.e., the one requiring the fewest conversions.

Examples:

```
R: TYPE = RECORD[fi: INTEGER];
RR: TYPE = RECORD[ff: LONG INTEGER];
i: INTEGER;
ii: LONG INTEGER;
rl, r2: R;
rr: RR;
...  
i = ii  -- LONG[i] = ii
rl = r2  -- compared as records
rl = i  -- rl.f = i
rl = rr  -- LONG[rl.f] = rr.f
```

Balancing is also applied to IF expressions (section 4.2.1), SELECT expressions (section 4.3.3), and the arithmetic and relational operators.

Fine points:

Many generic operators do not propagate the target type of the expression in which they appear; instead, the operands are balanced and combined to produce a result that is converted further if necessary. For example,

```
i + i + r;  -- ii + LONG[i] + rf
ii + LONG[i] + r;  -- ii + LONG[i] + LONG[rf]
```

The current version of Mesa does not fully implement balancing when lengthening (or conversion to REAL) is required. The restrictions are:

Operands of MIN and MAX and the alternatives of conditional expressions are lengthened to match the expression's target type, if any, and otherwise to match the type of the first operand.

The endpoints of an interval in the right operand of IN are lengthened to match the type of the left operand, but the left operand is never lengthened.

The expressions selecting the arms of a selection (section 4.3) are lengthened to match the type of the selecting expression, but that expression is never lengthened.
3.5.3. Free conformance *

A number of the conversions used to achieve conformance require computation and cannot be applied recursively to establish the conformance of types constructed from pairwise conforming types. For example, INTEGER conforms to REAL, but the conversion from INTEGER to REAL transforms the representation. Thus a POINTER TO INTEGER and POINTER TO REAL cannot validly have the same referent, and these types do not conform.

The relation of free conformance is less restrictive than strict type equivalence but is defined so that it can be computed recursively. Loosely speaking, one type freely conforms to another if a value of the first can always be used as a value of the second without any computation or run-time check of validity. The relations of equivalence, free conformance and conformance are not independent. Equivalence always implies free conformance; if two types are equivalent, each freely conforms to the other. Also, free conformance implies conformance; if one type freely conforms to another, the first also conforms to the second.

Of the automatic conversions discussed in section 3.5.1, only a restricted form of the first (subrange conversion) can be applied to establish free conformance. The restriction (which arises from the representation of subrange values in Mesa) is the following:

The subrange type \( T[i..j] \) conforms freely to \( T \) if \( i = \text{FIRST}[T] \) and to \( T[i..k] \) if \( j \leq k \).

If automatic conversion (1) of section 3.5.1 must be applied in any other circumstance or if application of conversion (2), (3) or (4) of that section is required to establish the conformance of two types, they do not conform freely.

Of the constructed types discussed in this chapter, array and pointer types also have rules for free conformance less restrictive than equivalence. To summarize:

One array type conforms freely to another if the index types are equivalent and the component type of the first freely conforms to the component type of the second (section 3.2.1).

One pointer type freely conforms to another whenever the first pointer type conforms to the second as defined in section 3.4.1.

Free conformance is also important for procedure types (section 5.1) and variant records (section 6.4).

In the following pairs of types, the first conforms to the second (but does not freely conform):

- \([0..100] \) to \([0..10] \)
- \([5..10] \) to \([0..10] \)
- INTEGER to REAL
- POINTER TO Person to LONG POINTER TO Person

In the following pairs, the first type freely conforms to the second (but is not equivalent):

- POINTER TO \([0..10] \) to POINTER TO READONLY \([0..10] \)
- POINTER TO READONLY \([0..10] \) to POINTER TO READONLY \([0..100] \)
- ARRAY \([0..10] \) OF \([0..10] \) to ARRAY \([0..10] \) OF CARDINAL

Fine point:

Note that POINTER TO \([0..10] \) does not conform to POINTER TO \([0..100] \) so that the following is illegal:

\[ p: \text{POINTER TO} \,[0..10]; \quad q: \text{POINTER TO} \,[0..100]; \]
3.6. Determination of representation

This section discusses the rules used by Mesa for choosing between signed and unsigned versions of the numeric operations. These rules assume that there are conversion functions (taking the form of range assertions, section 3.1.2.2) that convert values from CARDINAL to INTEGER (from LONG CARDINAL to LONG INTEGER) and vice versa. In both directions, the "conversion" amounts to an assertion that the value is an element of INTEGER \( \cap \) CARDINAL (LONG INTEGER \( \cap \) LONG CARDINAL). Such assertions must be verified by the programmer.

For any arithmetic expression, the inherent representations of the operands and the target representation of the result are used to choose between the signed and unsigned versions of the arithmetic and relational operators.

The target type determines the target representation. The preceding section describes the derivation of target types; in addition, a range assertion establishes the asserted type as the target type of its operand. If all valid values of the target type are nonnegative, the target representation is unsigned; otherwise, it is signed. The arithmetic operators propagate target representations unchanged to their operands, but the target representation of an operand of a relational operator is undefined. The target representation is also undefined in all other cases in which the target type is undefined. Thus each (sub)expression has at most one target representation.

The inherent representation of a Primary is determined by its type (if a variable, function call, etc.), by its value (if a compile-time constant), or explicitly (if a range assertion). Possible inherent representations are signed and unsigned; in addition, a compile-time constant in INTEGER \( \cap \) CARDINAL or a Primary with an inherent type that is a subrange of INTEGER \( \cap \) CARDINAL is considered to have both inherent representations. Inherent representations of operands are propagated to results as described below.

The operation denoted by a generic operator is chosen by considering first the inherent representations of its operands, next the target representation, and finally a preferred default. If the operation cannot be disambiguated in any of these ways, the expression is considered to be in error. The exact rules follow:

- If the operands have exactly one common inherent representation, the operation defined for that representation is selected (and the target representation is ignored).
- If the operands have no common inherent representation but the target representation is well-defined, the operation yielding that representation is chosen, and each operand is "converted" to that representation (in the weak sense discussed above).
- If the operands have both inherent representations in common, then if the target representation is well-defined it selects the operation; otherwise the signed operation is chosen.
- If the operands have no representation in common and the target representation is ill-defined, the expression is in error.

In all cases, the inherent representation of the result is determined by the selected operation.

The unary operators require special mention. Unary minus converts its argument to a signed representation if necessary and produces a signed result.
Example:

If $m$ and $n$ have unsigned representation, both the following are legal and assign the same bit pattern to $i$, but the first overflows if $m < n$.

$$i ← m+n; \quad i ← IF \; m ≥ n \; THEN \; m+n \; ELSE \; (n+m);$$

ABS is a null operation on an operand with an unsigned representation; it always yields a value with unsigned representation. The target representation for the operand of LONG (or of an implied lengthening operation) is unsigned.

Examples:

$$i, j: INTEGER; \quad m, n: CARDINAL; \quad s, t: [0..77777B]; \quad b: BOOLEAN$$

-- the statements on each of the following lines are equivalent.

$$i ← m+n; \quad i ← INTEGER[m+n] \quad -- \text{unsigned addition}$$
$$i ← j+n; \quad i ← n+j; \quad i ← j+INTEGER[n] \quad -- \text{signed addition}$$
$$i ← s+t; \quad i ← INTEGER[s]+INTEGER[t] \quad -- \text{signed (overflow possible)}$$
$$n ← s+t; \quad n ← CARDINAL[s]+CARDINAL[t] \quad -- \text{unsigned (overflow impossible)}$$
$$s ← st; \quad s ← CARDINAL[s]CARDINAL[t] \quad -- \text{unsigned (overflow possible)}$$
$$b ← st > 0; \quad b ← INTEGER[s]INTEGER[t] > 0 \quad -- \text{signed (overflow impossible)}$$
$$i ← m; \quad i ← INTEGER[m]$$
$$i ← m+n*(j+n); \quad i ← INTEGER[m] + (INTEGER[n]*(j+INTEGER[n]))$$
$$n ← m+n*(j+n); \quad n ← m + (n*(CARDINAL[j]+n))$$
$$i ← m+n*(s+n); \quad i ← INTEGER[m+(n*(CARDINAL[s]+n))]$$
$$b ← s \; \text{IN} \; [1 .. t+1]; \quad b ← INTEGER[s] \; \text{IN} \; [INTEGER[1] .. INTEGER[t+1]]$$

The following statements are incorrect because of representational ambiguities.

$$b ← i > n; \quad b ← i+n \; \text{IN} \; [s .. f]$$

SELECT $i$ FROM $m \Rightarrow \ldots; \quad t \Rightarrow \ldots; \quad \text{ENDCASE}$
CHAPTER 4.

ORDINARY STATEMENTS

Statements are the units of action in Mesa; they control the flow of execution and the updating of variables. This chapter treats ordinary statements: those statements having wide applicability (such as assignment statements); later chapters cover the remaining statements. The following syntax lists the phrase names of all the statement forms covered in this chapter:

\[
\text{Statement} ::= \text{AssignmentStmt} | \text{IfStmt} | \text{SelectStmt} | \text{NullStmt} | \text{Block} | \text{GotoStmt} | \text{LoopStmt} | \text{ExitStmt} | \ldots
\]

Some statements have expression counterparts, with the same general purposes but slightly different constraints. For instance, assignment can be performed by an expression as well as a statement. The expression forms covered in this chapter are

\[
\text{Expression} ::= \ldots | \text{AssignmentExpr} | \text{IfExpr} | \text{SelectExpr}
\]

In Mesa, certain statement forms such as the IF statement contain other statements. These statements in turn may contain still other statements, and so forth. Consequently, the term "statement" should be understood to encompass the large and small alike.

The dynamic successor of a statement embedded within another depends upon the embedding form. For simplicity, the discussion assumes that most statements occur in the middle of a hypothetical series of statements. Execution paths within a statement are described for each form of control statement, and the successor is described in terms of a postulated "Next-Statement". Next-Statement represents nothing more than completion of a given statement; another statement may or may not appear at that point in an actual program.

Although execution of a statement can be aborted prior to its normal completion, the discussion of statement sequencing also assumes normal termination of each statement unless otherwise stated.

In the examples, \text{Stmt-0}, \text{Stmt-1}, \text{Stmt-2}, etc. denote arbitrary statements, the details of which are irrelevant.

4.1. Assignment statements

Syntax:

\[
\text{AssignmentStmt} ::= \text{LeftSide} + \text{RightSide} | \text{Extractor} + \text{RightSide}
\]

The \text{RightSide} must be an expression with a type conforming to the type of the left-hand side. The left-hand side must be a valid recipient of data such as a declared variable or a component. For assignment statements, a left-hand side may also be an extractor (section 3.4.5).
Examples:

\[ i \leftarrow 3; \quad a \leftarrow b + c; \]

\[ \text{birthDay.month} \leftarrow \text{Apr}; \quad \text{birthTable[Tom].year} \leftarrow 1955; \]

\[ [\text{mm, dd, yy}] \leftarrow \text{birthDay}; \quad \text{-- an extractor as the LeftSide} \]

4.1.1. Assignment expressions

Assignment operations may be carried out by expressions, as well as by assignment statements. The syntax for an assignment expression is

\[
\text{AssignmentExpr ::= LeftSide} \leftarrow \text{RightSide}
\]

Assignment expressions can be used for performing multiple assignments in a single statement, and for saving the value of an intermediate expression without having to write a separate statement:

\[ x2 \leftarrow x1 \leftarrow x0 \leftarrow v; \quad \text{-- set } x0, x1, \text{ and } x2 \text{ to the value in } v \]

\[ \text{array}[j \leftarrow j+1] \leftarrow x[i]; \quad \text{-- } j \text{ is changed while changing the array component} \]

Evaluation of the first statement proceeds as if it were written:

\[ x2 \leftarrow (x1 \leftarrow (x0 \leftarrow v)) \]

Note that \( x2 \leftarrow (\ldots) \) is an assignment statement. The assignment expression, \( x0 \leftarrow v \), yields the value assigned to \( x0 \), this becomes the RightSide value for the other assignment expression, and so on.

There are two differences between an assignment expression and an assignment statement:

The expression yields a value (in addition to performing assignment).

The LeftSide of an assignment expression cannot be an extractor.

An AssignmentExpr is an Expression. Its type is the type of the LeftSide, and its value is the value actually assigned (possibly after type conversion) of the RightSide. The assignment operator has the lowest possible precedence. As a rule, an assignment expression embedded in another expression is enclosed in parentheses.

A fine point:

In an expression such as the following:

\[ a[k*k+1] + b[k]; \]

the order of evaluation is undefined, and the embedded assignment may be executed either before or after evaluation of \( b[k] \). Such use of embedded assignments should be avoided.

4.2. IF statements

An IF statement is a control statement that functions as a two-way switch:

\[
\text{IfStmt ::= \text{IF Predicate ThenClause ElseClause}}
\]

\[
\text{Predicate ::= Expression}
\]

\[
\text{ThenClause ::= \text{THEN Statement}}
\]

\[
\text{ElseClause ::= empty | ELSE Statement}
\]

A simple IF statement is shown below.

\[
\text{IF } v \leftarrow 0 \text{ THEN } \text{WriteString["Done."]} \text{ ELSE } v \leftarrow v1; \quad \text{Next-Statement}
\]
The BOOLEAN expression \((v = 0)\) is called the **Predicate** of the IF statement. The **Predicate** is evaluated first, and if **TRUE**, the **Statement** in the ThenClause is executed (in this case a call on the procedure **WriteString**). Upon its completion, execution continues at Next-Statement. If the **Predicate** value is **FALSE**, the **Statement** in the ElseClause, \("v \neq v1\", is executed; if there is no ElseClause control goes directly to Next-Statement.

Other examples:

```
IF (flag = on) AND i IN \([m..n]\) THEN i + iDelta ELSE i + m;
IF winner = NIL THEN
  BEGIN -- this Statement is a block (section 4.4)
    totalAge + totalAge + winner.age;
    IF winner.party = Democratic THEN demoScore + demoScore + 1
    ELSE gopScore + gopScore + 1;
  END; -- end of the ThenClause
```

**Note that a semicolon cannot follow a ThenClause when an ElseClause is present.**

If the **Statement** in a ThenClause is a second IF statement, then the outer IF may have an ElseClause only if the inner one does; i.e., an ElseClause "belongs" to the innermost possible IF. For example:

```
IF a >= 0 THEN
  IF a > 0 THEN b + 1 -- a > 0 means set b to 1
  ELSE b + 0; -- a = 0 means set b to 0
  ...
```

It is recommended that "IF...THEN IF" combinations be avoided entirely unless the second IF has an ElseClause. Often, a single IF statement is sufficient. For example, let \(p1\) and \(p2\) be arbitrary predicates. Then the following two statements have identical effect:

```
IF p1 AND p2 THEN Stmt;
IF p1 THEN IF p2 THEN Stmt; -- recommended form (see section 2.5.3)
```

**Fine point:**

If the **Predicate** is a compile-time constant, the compiler does not produce object code for the text that would never be executed. This also holds for IF expressions.

### 4.2.1. IF expressions

The IF statement has a counterpart that is an expression. Its syntax is similar to that of an IfStmt:

```
IfExpr ::= IF Predicate THEN Expression ELSE Expression
```

There are two differences between an IfExpr and an IfStmt:

- The clauses of an IF expression contain expressions, not statements;
- An IF expression must have an ELSE-clause.

Examples:

- \(slope + IF y = 0 THEN \text{max} ELSE x/y; \) -- avoid division by zero.
- \(b + IF a >= 0 THEN (IF a > 0 THEN 1 ELSE 0 ELSE 1; \)
Evaluation of an IF expression begins with evaluation of the Predicate (in the first example, \( y = 0 \)). If it is TRUE, the expression in the ThenClause (i.e., \( \text{max} \)) is evaluated, and its value becomes the value of the IF expression. If the predicate is FALSE, the ElseClause expression (i.e., \( x/y \)) is evaluated, and its value becomes the value of the IF expression. The second example sets the value of \( b \) to 1, 0, or +1, depending on whether \( a \) is negative, zero, or positive, respectively.

The ThenClause and ElseClause expressions must conform to some common type (possibly after type conversion, as outlined in section 3.5.3). The type to which they conform is the IF expression's inherent type.

An IF operator has the same precedence as an assignment operator, i.e., the lowest possible precedence. IF expressions should be enclosed in parentheses when embedded in other expressions.

### 4.3. SELECT statements

The SELECT statement chooses for execution at most one statement from an ordered list of statements. The choice is based upon the relation between a given expression and expressions associated with each selectable statement. Thus, this statement form permits multiway branching, not just the two-way branching of an IF statement.

A SELECT statement is shown below. The separator "\( = > \)" should be read as "chooses." The entire statement may be read as follows: "Select, using \( x \)'s value, from the comparisons preceding the substatements. First, \( (x \)'s value) 'equal to zero' chooses Stmt-1. Second, 'in subrange \( m \) through \( n \)' chooses Stmt-2. Third, 'less than \( m \)' chooses Stmt-3. Otherwise, choose nothing."

```
SELECT x FROM
  = 0  => Stmt-1;
IN [m..n]  => Stmt-2;
< m   => Stmt-3;
ENOCASE
```

The next four sections cover various forms of SELECT, their precise syntax, and the expression counterpart of the SELECT statement. The term "SELECT", used by itself, includes both statement and expression forms.

### 4.3.1. Forms and options for SELECT

Syntax equations:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectStmt</td>
<td>( \text{SELECT } \text{LeftItem FROM } \text{StmtChoiceSeries ENOCASE FinalStmtChoice} )</td>
</tr>
<tr>
<td>LeftItem</td>
<td>( \text{Expression} )</td>
</tr>
<tr>
<td>StmtChoiceSeries</td>
<td>( \text{TestList ( = &gt; ) Statement }</td>
</tr>
<tr>
<td>FinalStmtChoice</td>
<td>( \text{empty }</td>
</tr>
<tr>
<td>TestList</td>
<td>( \text{Test }</td>
</tr>
<tr>
<td>Test</td>
<td>( \text{Expression }</td>
</tr>
</tbody>
</table>

The term \( = > \) is used in the SELECT statement to represent the "chooses" operator. The separate 

It is important to note that the SELECT statement includes both statement and expression forms, allowing for multiway branching in the selection process.
Example:

\[ i: [0..5]; \]

\[
\text{SELECT } i \text{ FROM } \\
\quad 0 \Rightarrow i \leftarrow i+1; \quad \text{-- } i=0 \\
\quad < 3 \Rightarrow \text{BEGIN } j \leftarrow i; \ i \leftarrow \not n \ \text{END}; \quad \text{-- } i=1 \text{ or } i=2 \\
\quad = 5 \Rightarrow i \leftarrow 0; \quad \text{-- } i=5 \\
\quad \text{ENDCASE } \Rightarrow i \leftarrow 2; \quad \text{-- } i=3 \text{ or } i=4 \text{ (none of the above)}
\]

Next-Statement

In the execution of a SELECT statement, the \textbf{LeftItem} is evaluated first; a sequence of comparisons then follows. Each arm of the SELECT statement begins with one or more Tests. The Expression in each Test is evaluated and compared with the value of the \textbf{LeftItem}. The evaluation occurs in order, from left to right, and continues until a comparison succeeds or the \textbf{TestList} for that particular arm is exhausted. If a test succeeds, control passes immediately to the statement following the \textbf{TestList} for that arm (no further Tests are evaluated, even in that same list). If all Tests in a given arm fail, the next arm in the series is tried. After a test succeeds and its associated statement is executed, control passes to \textbf{Next-Statement}. Thus at most one statement can be chosen in a given execution of a SELECT statement.

When combined with the \textbf{LeftItem} (perhaps with an implied "=", each Test must be a valid \textbf{Relation}. The type of the Expression in a Test must conform to the type of the \textbf{LeftItem}. If a Test uses "\texttt{IN Subrange}", the base type of the subrange must conform to the type of the \textbf{LeftItem}.

A single SELECT arm may specify more than one test:

\[
\text{SELECT } i^*j+k \text{ FROM } \\
\quad 1, \text{ IN } [7..10] \Rightarrow \text{Stmt-1; \ -- values: } 1, \ 7, \ 8, \ 9, \ 10 \\
\quad 2, 5, > 10 \Rightarrow \text{Stmt-2; \ -- values: } 2, \ 5, \ 11, \ 12, \ ...
\]

A final choice may be appended to a SELECT to handle all remaining cases; it follows ENDCASE. For example:

\texttt{PriorityState: TYPE } = \texttt{RECORD}[i0, i1, i2, i3: BOOLEAN];
\texttt{oldState, newState: PriorityState;}

\[
\text{SELECT TRUE FROM } \quad \text{-- picks the first TRUE state:} \\
\quad \text{oldState.i0 } \Rightarrow \text{Stmt-0;} \\
\quad \text{oldState.i1, newState.i0 } \Rightarrow \text{Stmt-1;} \\
\quad \text{oldState.i2, newState.i1 } \Rightarrow \text{Stmt-2;} \\
\quad \text{oldState.i3, newState.i2 } \Rightarrow \text{Stmt-3;} \\
\quad \text{ENDCASE } \Rightarrow \text{Stmt-99;}
\]

If this SELECT statement does not choose one of the first four statements, the final statement (\texttt{Stmt-99}) is executed.

Fine points:

If all SELECT arms (or those in some contiguous subseries) specify constant values in each Test, the compiler can produce code using a "jump table" for efficient selection.

The other alternatives for SelectStmt apply to variant records and are discussed in Chapter 6.
4.3.2. The NULL statement

The NULL statement, which serves only as a placeholder, is often useful as the statement in an arm of a SELECT statement:

```
NullStmt ::= NULL
```

For example:

```
SELECT currentChar FROM
  IN ['0..9'] => Stmt-1;  --Handle digits.
  IN ['A..Z'] => Stmt-2;  --Handle capital letters.
  IN ['a..z'] => Stmt-3;  --Handle small letters.
  SP => NULL;            --Ignore blanks.
ENDCASE => Stmt-99;      --Handle all other chars.
```

4.3.3. SELECT expressions

The SELECT statement has an expression counterpart. There are three differences between the expression and statement forms of SELECT:

1. The choices in each arm must be expressions, not statements.
2. The arms are terminated by commas, not semicolons.
3. ENDCASE must be followed by "=)" and a final (expression) choice.

Its syntax is defined by

```
SelectExpr ::= SELECT LeftItem FROM ExprChoiceList ENDCASE => Expression | ...
ExprChoiceList ::= TestList => Expression, | ExprChoiceList TestList => Expression,
```

`LeftItem` and `TestList` are defined in section 4.3.1.

For example:

```
pt: INTEGER;
lo, hi: INTEGER + 0;  -- Bounds for a line segment, initially a null segment
...
PointPosition: TYPE = {leftMargin, rightMargin, inside, outside, degenerate};
position: PointPosition;
...
position + SELECT pt FROM
  IN (lo..hi) => inside,
  NOT IN [lo..hi] => outside,
  < hi  => leftMargin,  -- = lo but # hi
  > lo  => rightMargin,  -- = hi but # lo
ENDCASE => degenerate;  -- = lo and = hi
```

A SELECT expression is executed just as a SELECT statement, except that the selected arm yields a value, which becomes the value of the SELECT expression as a whole. The inherent type of a SELECT expression is the one to which all the expressions in the arms conform (section 3.5.3).

A SELECT operator has the same precedence as an assignment operator, i.e., the lowest possible precedence. SELECT expressions should be enclosed in parentheses when embedded in other expressions.
4.4. Blocks

A block is a way of packaging a series of statements so that they can be used where only a single statement is permitted syntactically. In its simplest form a block is a pair of "brackets", BEGIN and END, with a series of statements (of any form) between them. The general syntax is

```
Block :: = BEGIN
    OpenClause  -- optional; section 4.4.2
    EnableClause -- optional; section 8.2.1
    DeclarationSeries  -- optional
    StatementSeries
    ExitsClause  -- optional; section 4.4.1
END
```

```
StatementSeries :: = empty | Statement | Statement ; StatementSeries
```

```
DeclarationSeries :: = empty | DeclarationSeries Declaration
```

A fine point:

A semicolon terminates every declaration and therefore is not mentioned as a separator here.

In the following IF statement, a block takes the place of the single Statement normally allowed in a ThenClause:

```
IF lo > hi THEN
    BEGIN -- Exchange lo and hi.
        temp: INTEGER +- lo;
        lo +- hi;
        hi +- temp;
    END
```

A semicolon must separate each statement in the StatementSeries but is optional after the last statement.

The optional DeclarationSeries in a block introduces new identifiers, such as temp above, with scope smaller than an entire procedure (or module) body. Scope is discussed further in sections 4.4.2 and 5.5.1 and in chapter 7.

A fine point:

During the execution of a Mesa program, frames are allocated at the procedure and module level only (section 5.2). Any storage required by variables declared in an internal Block (one used as a Statement) is allocated in the frame of the smallest enclosing procedure or module. When such internal blocks are disjoint, the areas of the frame used for their variables overlay one another.

Ordinarily, when a block is executed, every statement in its StatementSeries is executed, and Next-Statement is the successor of the entire block. It is possible, however, to jump out of a block, as described in the next section on GOTOS.

4.4.1. GOTO statements

A more general form of a block allows a series of labeled statements to be written immediately preceding its END. One can jump to any one of these statements from within the block only, using a GOTO statement. There are two consequences of this way of constraining the GOTO:

- A GOTO may only jump forward in the program, never backward.
- A GOTO may only jump out of a block, never into one.

The syntax for the ExitsClause of a block and for the GOTO statement is the following:
A simple example:

\[
\text{IF } \text{input.status} \neq \text{open} \text{ THEN}
\]

\[
\text{BEGIN}
\]

\[
\text{...}
\]

\[
\text{IF } \text{input.fileHandle} = \text{defaultInput} \text{ THEN GOTO useDefault;}
\]

\[
\text{...}
\]

\[
\text{-- processing for non-default file}
\]

\[
\text{IF } \text{input.fileNumber} = \text{ttyNumber} \text{ THEN GOTO fileIsDefault;}
\]

\[
\text{IF } \text{input.length} = 0 \text{ THEN GOTO newFile;}
\]

\[
\text{...}
\]

\[
\text{-- compute number of pages in the file}
\]

\[
\text{EXITS useDefault, fileIsDefault} \Rightarrow
\]

\[
\text{BEGIN input} \leftarrow \text{ttyInput}; \text{pages} \leftarrow \text{maxPages END;}
\]

\[
\text{newFile} \Rightarrow \text{pages} \leftarrow 0;
\]

\[
\text{-- end of the ThenClause and the IF statement}
\]

\[
\text{Next-Statement}
\]

The Labels in this example are useDefault, fileIsDefault, and newFile (it is helpful to view the labels as the names of conditions or reasons for which the block is being left). If any one of the GOTOS is executed, control transfers immediately to the statement labeled with the identifier used in the GOTO. The normal successor of any one of the labeled statements is Next-Statement, which is also the normal successor of the last statement in the main body of the block (i.e., the one just before EXITS).

Since one block can appear within the body of another, a GOTO can jump directly out of one (or more) blocks to the ExitsClause of an enclosing block. For example,

\[
\text{BEGIN}
\]

\[
\text{...}
\]

\[
\text{BEGIN}
\]

\[
\text{...}
\]

\[
\text{IF } i = iMax \text{ THEN GOTO endOfArray;}
\]

\[
\text{...}
\]

\[
\text{-- jump to end of outer compound}
\]

\[
\text{END;}
\]

\[
\text{endOfArray} \Rightarrow i \leftarrow 0;
\]

\[
\text{-- end of outer}
\]

\[
\text{Next-Statement}
\]

If the GOTO statement is executed, control jumps to the exit labeled endOfArray. The chosen statement (i+0) is executed and control then goes to Next-Statement. The identifiers used as Labels are only known inside the block in which they appear, and it is possible to use the same identifier as a label in a number of blocks. If this is done in nested blocks, a GOTO naming that identifier will always go to the statement with that label in the smallest enclosing block. Generally, using the same label in nested blocks is a bad idea.
Since Mesa allows declarations in any block, it is possible to declare a procedure (section 5.5) within the scope of the Labels of a block. Jumping out of a procedure into a surrounding block is disallowed. Such a result may be obtained, however, by use of the SIGNAL machinery (see chapter 8). For example, the following is illegal:

```
BEGIN
  p: PROCEDURE =
  BEGIN
    . . . GOTO panicExit; -- illegal GOTO -- . . .
  END;
  . . . p[i]; . . .
EXITS
  panicExit => ...
END
```

The desired result is achieved with the following program (see chapter 8 for a description of signals and catch phrases):

```
BEGIN
  Panic: SIGNAL = CODE;
  . . .
  p: PROCEDURE =
  BEGIN
    . . . SIGNAL Panic; . . .
  END;
  . . . p[i] ! Panic => GOTO panicExit]; . . .
EXITS
  panicExit => ...
END
```

A statement in an ExitsClause may contain a GOTO, but the label in the GOTO can only refer to labels in surrounding blocks, not to labels in the same ExitsClause as the GOTO. For example, the following is legal:

```
BEGIN
  -- outer
  . . .
BEGIN
  -- inner
  . . .
EXITS
  endOfFileReached => BEGIN . . . GOTO outOfData END;
  END;
  -- end of inner
  . . .
EXITS
  outOfData => . . .
END
  -- end of outer
```

4.4.2. OPEN clauses

An OPEN clause allows more convenient reference to the fields of a record. In the simplest form, it allows fieldname as an abbreviation for recordname.fieldname. If the name of the record is complicated (e.g., candidateList[tableOfObjects[i]]), this can make programs much more readable. The programmer should be cautioned, however, that this is merely a syntactic shorthand; the code generated is actually recordname.fieldname. Thus in the example above, if i or tableOfObjects is changing within the scope of the OPEN, each reference to a field can potentially access a different element of candidateList. The syntax for OPEN follows:
Chapter 4: Ordinary Statements

OpenClause ::= empty | OPEN OpenList ; -- note the terminal semicolon
OpenList ::= OpenItem | OpenList , OpenItem
OpenItem ::= AlternateName : Expression | Expression
AlternateName ::= identifier

The scope of an OPEN clause (the portion of the program over which the synonym can be used) is the body of a block or loop, including the optional exits clause (section 4.5). The following diagram summarizes the scope of the various parts of a Block. The scope of each phrase extends over others with greater indentation.

BEGIN
OpenClause
   EnableClause
   DeclarationSeries
   StatementSeries
   ExitsClause
END

An OpenItem using an AlternateName allows a simple identifier to replace an expression as the designator of some record object. For example, the two blocks below are equivalent:

```
PersonChain: TYPE = RECORD [p:POINTER TO Person, next: POINTER TO PersonChain]  
candidateList: POINTER TO PersonChain; -- Person is defined in section 3.5
BEGIN OPEN c: candidateList.p;
   IF c.party = Republican AND c.age < 30 THEN youngRepublicans ← youngRepublicans+1;
   IF c.sex = Female THEN women ← women+1;
   ...  
END

BEGIN
   IF candidateList.p.party = Republican AND candidateList.p.age < 30 THEN  
      youngRepublicans ← youngRepublicans+1;
   IF candidateList.p.sex = Female THEN women ← women+1;
   ...  
END
```

The OPEN statement does not provide a general renaming capability; it merely allows more convenient access to the fields of a record. Each Expression in an OpenList must either have a record type or be a pointer to a record. When the AlternateName form is used, the alternate identifier always designates the opened record, even if the Expression is a pointer to that record.

The form of OpenClause without an AlternateName allows access to the fields of a record object as though they were simple variables. For example, using this feature in the above example allows omission of the "c."'s:

```
BEGIN OPEN candidateList.p;
   IF party = Republican AND age < 30 THEN youngRepublicans ← youngRepublicans+1;
   IF sex = Female THEN women ← women+1;
   ...  
END
```

Note, if the AlternateName form is used, qualification of record fields using the alternate name is mandatory.
Besides record objects, one can open a module (chapter 7) to simplify access to the identifiers available from the module.

If an OpenClause contains multiple OpenItems, the opened expressions might refer to records having some selector names the same. In the example below, $x$ is a selector name for two records, $\text{recVar}$ and $\text{recVar.subRecord}$. An unqualified occurrence of $x$ is taken to be the $x$ component of the rightmost opened record ($\text{recVar.subRecord}$). To refer to an earlier opened record, explicit qualification is necessary (the AlternateName form should be used).

```plaintext
i, j: INTEGER;
RecordType: TYPE = RECORD[
  a, b, x: INTEGER,
  subRecord: RECORD[x, y: INTEGER]
];
recVar: RecordType;
BEGIN OPEN rl: recVar, recVar.subRecord;
i' = rl.a + rl.b * rl.x;
j' = xji;
END;
```

The above block is equivalent to:

```plaintext
BEGIN
  i' = recVar.a + recVar.b * recVar.x;
  j' = recVar.subRecord.x + recVar.subRecord.y;
END;
```

Fine points:

- The range of text affected by an OpenItem includes any further items in the OpenList. The OpenClause itself may use implied qualification or alternate names (from earlier OpenItems).
- Opened expressions are evaluated at each use, whether used implicitly or explicitly under an alternate name.
- This is essential for dealing with relocating allocation schemes. To avoid confusion, however, it is recommended that ordinary pointers be updated before entering the statement sequence headed by an OpenClause. In that way, names in the statement sequence will remain consistent, i.e., will apply to the same objects throughout.

### 4.5. Loop statements

In Mesa, a loop is a statement containing a series of statements that are to be executed repeatedly. All the ways of controlling how many times a loop should be repeated include the ability to repeat it zero times: i.e., to bypass it entirely. Example 1 in section 2.1 contains the following loop statement:

```
UNTIL n = 0
  DO
    r = m MOD n; -- r gets remainder of $m/n$
    m = n; n = r;
  ENDLOOP
```

"UNTIL $n=0"$ is the loop control for this loop. A variety of loop controls are available in Mesa: they include control by a Boolean expression, as above, and control by iteration over a subrange, as in the following example:

```
FOR i IN [0..N) DO a[i] = a[i] + b[i] ENDLOOP
```

This will execute the assignment $N$ times, with $i$ taking the values 0, 1, ..., $N-1$ on successive iterations. If $N = 0$, the assignment is not executed at all.
The formal syntax of loop statements is

\[
\text{LoopStmt} \quad ::= \quad \text{LoopControl} \quad \text{DO} \quad \text{OpenClause} \quad \text{EnableClause} \quad \text{StatementSeries} \quad \text{LoopExitsClause} \quad \text{ENDLOOP}
\]

-- optional; may be empty

DO

-- optional (section 4.4.2)

-- optional (section 8.2.1)

-- optional; may be empty

The portion between DO and ENDLOOP is the body of a loop. Subsequent sections discuss the forms of LoopControl, the LoopExitsClause and GOTOS in loops.

The scopes of identifiers introduced in the various components of a loop are summarized by the following diagram (cf. Block, section 4.4.2):

\[
\text{LoopControl} \quad \text{DO} \quad \text{OpenClause} \quad \text{EnableClause} \quad \text{StatementSeries} \quad \text{LoopExitsClause} \quad \text{END LOOP}
\]

As in the case of a block, any exit labels are visible within the EnableClause, and any catch phrase in the EnableClause is not enabled within the LoopExitsClause.

4.5.1. Loop control

The syntax for LoopControl is

\[
\text{LoopControl} \quad ::= \quad \text{IterativeControl ConditionTest} \quad -- \text{either may be empty}
\]

\[
\text{ConditionTest} \quad ::= \quad \text{empty} \quad | \quad \text{WHILE Expression} \quad | \quad \text{UNTIL Expression}
\]

If both the IterativeControl and the ConditionTest are missing from a loop, it will repeat indefinitely (unless terminated by an embedded GOTO or EXIT, section 4.5.2).

If a LoopControl includes a ConditionTest, the Boolean expression in the test is (re)evaluated before each execution of the loop body, including the first. If the ConditionTest succeeds, the body of the loop is executed; if it fails, the loop is finished (terminates conditionally) and control continues at Next-Statement (or at a FINISHED clause, see section 4.5.2). A WHILE test succeeds if the value of the expression is TRUE. In the following example, \(i\) has the values 1, 2, 3, ..., 9 in successive executions of the body of the loop, and the value 10 when Next-Statement is reached (assuming that there are no other assignments to \(i\)):

\[
i + 1; \quad -- \text{this statement is not part of the loop}
\]

\[
\text{WHILE } i < 10 \\
\text{DO } \ldots i = i + 1; \quad \ldots \text{ENDLOOP};
\]

Next-Statement

An UNTIL test succeeds if the value of the expression is FALSE: i.e., it is the opposite of WHILE. The following loop is equivalent to the one above:

\[
i + 1; \quad -- \text{this statement is not part of the loop}
\]

\[
\text{UNTIL } i \geq 10 \\
\text{DO } \ldots i = i + 1; \quad \ldots \text{ENDLOOP};
\]

Next-Statement
An **IterativeControl** provides a way of executing a loop (no more than) a precomputed number of times. It may be followed by a **ConditionTest**. It optionally updates a specified **ControlVariable** prior to each iteration so that, e.g., statements in the body have access to (a simple function of) the number of iterations. A loop that finishes by satisfying the implicit test associated with an **Iteration** or a **Repetition** is said to **terminate normally**.

\[
\text{IterativeControl ::= empty | Repetition | Iteration | Assignation}
\]

\[
\text{Repetition ::= THROUGH LoopRange}
\]

\[
\text{Iteration ::= FOR ControlVariable Direction IN LoopRange}
\]

\[
\text{LoopRange ::= SubrangeTC | Typeldentifier | BOOLEAN | CHARACTER}
\]

\[
\text{Direction ::= empty | DECREASING}
\]

\[
\text{Assignation ::= FOR ControlVariable + InitialExpr, NextExpr}
\]

\[
\text{ControlVariable ::= identifier}
\]

\[
\text{InitialExpr ::= Expression}
\]

\[
\text{NextExpr ::= Expression}
\]

In the **Repetition** form of **IterativeControl**, a **LoopRange** specifies how many times the loop body is to be executed. For example,

\[
\text{THROUGH [1..100] DO ... ENDLOOP}
\]

executes the body 100 times. A loop range can have any element type (section 3.1) except **INTEGER** or **CARDINAL**. The bounds of a subrange can be arbitrary expressions and do not have to be compile-time constants (as they do in a **SubrangeTC** used to define a type).

A **Repetition** and a **ConditionTest** may be combined in a single loop control. For example,

\[
\text{THROUGH [low..high] WHILE lineIsConnected DO ... ENDLOOP}
\]

Normal termination occurs after \( high - low + 1 \) iterations; conditional termination can occur sooner if \( lineIsConnected \) is **FALSE** prior to some iteration. Note that if \( low > high \), the interval \([low..high]\) is empty and the loop body is not executed.

**Iteration** and **Assignation**, the two forms of **IterativeControl** that include a **ControlVariable**, begin with the keyword **FOR**. The **control variable** must be a variable declared separately in the program. Its type becomes the target type for the various expressions in the remainder of the **IterativeControl**.

An **Iteration** steps through a subrange much as a **Repetition**, which is described above. In addition, it may specify a **Direction**: whether to begin at the lower bound of the range and step up (empty) or at the upper bound and step down (DECREASING). In any case, the size of the step is always one; for (a subrange of) an enumerated type, this really means stepping from an element to its successor (if the direction is increasing) or to its predecessor (if the direction is **DECREASING**). The control variable is assigned the current control value each time around the loop.

When a loop terminates **normally**, the final value of the control variable is **not** defined. The only way to ensure that the control variable final value is well defined is to terminate the loop conditionally or **forcibly** (e.g., using **EXIT** or **GOTO**, section 4.5.2).
The following examples shift the components of an array \( \text{vec} \) left or right one position, leaving one element unchanged:

```fortran
FOR \( i \) \text{IN} [1..\text{LENGTH}[\text{vec}])
    \text{do}
    \text{vec}[i] \leftarrow \text{vec}[i];
    \text{ENDLOOP};
FOR \( i \) \text{DECREASING IN} [1..\text{LENGTH}[\text{vec}])
    \text{do}
    \text{vec}[i] \leftarrow \text{vec}[i];
    \text{ENDLOOP};
```

"Left-shift" \( \text{vec} \)'s elements.

"Right-shift" \( \text{vec} \)'s elements.

In the second case, \( i \) is initially set to the value \( \text{LENGTH}[\text{vec}] \) and decremented by one for each subsequent iteration. During the last execution of the loop, \( i \) has the value 1.

Bounds expressions in a LoopRange are evaluated exactly once, before the first execution of the loop body. Subsequent alteration of variables used in those expressions does not affect the number of iterations. When an Iteration is combined with a ConditionTest in a single loop control, the control variable is updated and tested before the ConditionTest is evaluated.

In an Assignation, the value of the InitialExpr is assigned to the control variable prior to the first iteration. Before each subsequent iteration, the NextExpr is (re)evaluated and assigned to that variable. There is no implicit test associated with an Assignation as there is for an Iteration; thus, the user must either use a GOTO (section 4.5.2) to terminate the loop or include a ConditionTest in the LoopControl with the Assignation. As with an Iteration, the control variable is updated for each iteration before any ConditionTest is evaluated. This form is useful for scanning a list structure, as in the following example:

```fortran
NodeLink: TYPE = POINTER TO Node;
node, headOfList: NodeLink;
Node: TYPE = RECORD
    listValue: SomeType,
    next: NodeLink
    \text{-- either NIL (end of list) or pointer to next element}
\};

\text{FOR node + headOfList, node.next until node=NIL
    \text{do} \ldots \text{ENDLOOP};}
```

A fine point:

The control variable can be altered within a loop, but this is not recommended. An iterative loop control updates the variable according to its current value. If the statement sequence assigns a new value to the control variable, the expected series of values may be disrupted (by omission or duplication).

### 4.5.2. GOTOS, LOOPS, EXITS, and loops

A loop may be forcibly terminated by a GOTO (or an EXIT, see below). The LoopExitsClause serves the same purpose as the ExitsClause in a Block; there are just three differences:

1. The LoopExitsClause is bracketed by REPEAT and ENDLOOP instead of EXITS and END;
2. The LoopExitsClause may contain a final statement labeled with the keyword FINISHED; this statement is executed if the loop terminates normally or conditionally, but not if it is forcibly terminated.
3. There is a special case of the more general GOTO, called EXIT, which simply terminates a loop forcibly without giving control to any statement in the LoopExitsClause.
There is another kind of GOTO statement, LOOP, which does not terminate the loop but skips the remainder of the loop body in the current iteration.

Syntax equations:

- **LoopExitsClause** :: = empty | REPEAT LoopExits
- **LoopExits** :: = ExitSeries | ExitSeries ; FinishedExit | ExitSeries ; FinishedExit
- **FinishedExit** :: = FINISHED => Statement | FINISHED => Statement ;
- **LoopCloseStmt** :: = LOOP
- **ExitStmt** :: = EXIT

The LOOP statement is used when there is nothing more to do in the iteration, and the programmer wishes to go on to the next repetition, if any. For example,

```plaintext
stuff: ARRAY [0..100] OF PotentiallyInterestingData;
Interesting: PROCEDURE [PotentiallyInterestingData] RETURNS [BOOLEAN];
i: CARDINAL;

FOR i IN [0..100) DO
    -- some processing for each value of i
    ...
    IF ~Interesting[stuff[i]] THEN LOOP;
    -- process stuff[i]
    ...
ENDLOOP;
```

The example used in the previous section to illustrate ConditionTests can be rewritten using a GOTO and a LoopExitsClause as follows:

```plaintext
i+1;
DO
    IF i >= 10 THEN GOTO quit;          -- first statement in the body
    ... i = i+1; ...
    REPEAT
        quit => NULL;                     -- do nothing but exit the loop
    ENDLOOP;
ENDLOOP;
```

Frequently, forcible loop termination requires no special processing in the LoopExitsClause. The EXIT statement simplifies this case by not requiring a labeled statement in that clause; in fact, no LoopExitsClause need be present. The above example can be further rewritten to use EXIT, as follows:

```plaintext
i+1;
DO
    IF i >= 10 THEN EXIT;               -- first statement in the body
    ... i = i+1; ...
ENDLOOP;
```
An EXIT is less general than a GOTO. For instance, if one has a loop nested within another and wants to exit from both, EXIT cannot be used because it terminates only the inner loop. A GOTO can jump to the ExitsClause of any enclosing loop or block. The ExitsClause of either a block or a loop is considered to be outside of the block or loop. Thus, an EXIT can appear in any ExitsClause (provided there is an outer loop), and it causes forcible termination of the smallest surrounding loop.

The following example shows a typical loop that is terminated only by execution of an EXIT statement.

```plaintext
BufIndexType: TYPE = [1..max];
buf: ARRAY BufIndexType OF INTEGER;
i, x: BufIndexType;

FOR i = x, (IF i = max THEN 1 ELSE i+1) DO
    -- Starting at point x,
    IF buf[i] = 0 THEN EXIT;
    buf[i] = 0;
ENDLOOP;
```

The NextExpr, "IF i = max THEN 1 ELSE i+1", makes buf behave as a ring buffer.

Sometimes one must detect normal (as opposed to forcible) termination of a loop, perhaps to take some "finishing" action. A final labeled statement with the label FINISHED (which may not appear as the identifier in a GOTO) provides this facility. For example,

```plaintext
FOR i IN [0..nEntries) DO
    IF a[i] = x THEN GO TO found;
    REPEAT
        found => old := TRUE;
        FINISHED => BEGIN a[i + nEntries] := x; nEntries := nEntries + 1; old := FALSE END;
    ENDLOOP;
```

The FINISHED exit is taken if and only if the loop terminates normally or conditionally (i.e., when the loop range is exhausted in the case above). Upon entry to a FINISHED exit, the value of the ControlVariable is undefined. Note that if an EXIT statement is executed, the FINISHED statement is not executed.
CHAPTER 5.

PROCEDURES

Procedures provide one of the most important abstraction mechanisms available in Mesa. The definition of a procedure assigns a name to a function or action. The computation performed by a procedure is specified by a series of statements and can be expressed in terms of parameters of the procedure. In addition, a procedure can produce one or more values, called its results. To invoke or call a procedure, the programmer simply names it and supplies arguments corresponding to the parameters. He need not be concerned with the internal workings of the procedure and can use a meaningful name to denote the function or action.

The GCD computation in section 2.1 is of limited use as it stands because it depends upon (and changes) variables \( m, n \) and \( \text{gcd} \) declared somewhere in its environment. It might usefully be packaged as a procedure with parameters \( m \) and \( n \). Such a procedure is declared as follows:

\[
Gcd: \text{PROCEDURE } [m, n: \text{INTEGER}] \text{RETURNS } \text{[CARDINAL]} =
\]

\[
\begin{align*}
\text{BEGIN} \\
\quad r: \text{INTEGER}; \\
\quad \text{UNTIL } n = 0 \\
\quad \quad \text{DO} \\
\quad \quad \quad r \triangleq m \mod n; \quad m \triangleq n; \quad n \triangleq r; \\
\quad \quad \text{ENDLOOP;} \\
\quad \text{RETURN } [\text{ABS}[m]] \\
\text{END;}
\end{align*}
\]

The parameters of a procedure constitute the fields of a record, called the parameter record of the procedure. When calling a procedure, the arguments are evaluated and assembled into an argument record using a constructor (section 3.3.4). "Applying" a procedure value to that argument record invokes the procedure. Consider the procedure call \( Gcd[x+1, y] \). This evaluates \( x+1 \) and \( y \), constructs an argument record from these values, and then calls procedure \( Gcd \), passing it the argument record.

Within the procedure, the argument record is assigned to the parameter record, and fields of the parameter record are accessed as simple variables (i.e., that record is "opened"). Thus the effect of the call above is to assign the value of \( x+1 \) to \( m \) and the value of \( y \) to \( n \) before the statements within \( Gcd \) are executed.

A procedure may return values to the point of its call. These results constitute a result record. There may be any number of results, and their types may differ. Within a procedure, a RETURN statement assembles the results into a record and then returns control to the caller. The procedure \( Gcd \) returns a result record with one component, of type CARDINAL. Thus the form \( Gcd[x+1, y] \) is an expression with a record type; because of the automatic conversion from a single-component record to the component (section 3.5.1), it can also be used in any context accepting a value of type CARDINAL.
The following assignment has an effect similar to that of the entire example in section 2.1:

\[ \text{gcd} \leftarrow \text{Gcd}[m, n] \]

Note that arguments are always passed by value in Mesa. The arguments \( m \) and \( n \) (for which declarations must exist at the point of call in this case) are completely distinct from the parameters \( m \) and \( n \), and execution of \( \text{Gcd} \) does not change the values of the former.

A procedure declaration with the form illustrated above is said to define an actual procedure. It introduces an identifier, supplies some procedure type for that identifier, and defines the computation to be performed by specifying a block called the procedure body. Such a declaration uses fixed form \text{Initialization} and closely parallels the declaration of an ordinary variable with \text{= initialization}, e.g.,

\[ \text{octalRadix: CARDINAL} = 8; \]

Other declaration forms may also be used, and this allows one to have procedure variables with values that can be updated to designate different actual procedures. In Mesa, procedures are full-fledged data objects.

A procedure type is defined by specifying its parameter and result records. For example, the type of \( \text{Gcd} \) is

\[
\text{PROCEDURE} [m, n: \text{INTEGER}] \text{ RETURNS} [\text{CARDINAL}]
\]

Procedure types constructed with different parameter and result records are different; thus the type system helps to guarantee that, even when procedure variables are used, a proper argument record is constructed for each procedure call (i.e., that the number and types of the arguments are correct) and that the result record is used correctly in the text surrounding the procedure call.

Since a procedure body is a block, it may contain declarations. These declare local variables for that procedure, variables that are created when the procedure is called, may be directly accessed only from within it, and are destroyed when the procedure returns. Within a procedure body, the named fields of the parameter and result records are also considered local variables; they have the same lifetimes and can be referenced without qualification. The local variables of \( \text{Gcd} \) are \( m, n \) and \( r \).

Because this local storage is allocated and released dynamically, any Mesa procedure can be invoked recursively and used in a reentrant fashion. Thus the following alternative declaration of \( \text{Gcd} \), which directly mirrors a recursive definition of the greatest common divisor, is valid:

\[
\text{Gcd: PROCEDURE} [m, n: \text{INTEGER}] \text{ RETURNS} [\text{CARDINAL}] = \\
\begin{align*}
\text{BEGIN} \\
\text{RETURN} \left\{ \begin{array}{ll}
\text{IF} & n = 0 \text{ THEN ABS}[m] \\
\text{ELSE} & \text{Gcd}[n, m \mod n]
\end{array} \right. \\
\text{END};
\end{align*}
\]

A fine point:

Although both versions of \( \text{Gcd} \) compute the same function, the recursive one is extravagant in its use of time and space (especially since an iterative version is so easy to derive). This demonstrates an advantage of procedural abstraction; the second definition of \( \text{Gcd} \) could be replaced by the first without effect on any caller of \( \text{Gcd} \). Examples in section 5.4 demonstrate more appropriate uses of recursion.

A procedure body may also access variables declared outside of the actual procedure. Such variables are nonlocal to the procedure; they exist longer than any single invocation of the procedure and must be defined in the enclosing program text.

Mesa also has extensive facilities supporting the separate compilation of packages of procedures and variables; these packages are called modules (chapter 7). Some of these facilities allow one module to name and use the procedures in another, but the type-correct usage of argument and result records is still checked at compile-time.
If a procedure is called from many places, the "packaging" of code provided by the procedure body makes a program more compact. Procedure calls and returns do, however, introduce some runtime overhead. If a procedure is called from exactly one place, that overhead is unnecessary; if it is called from time-critical code or if the body of the procedure is very simple, the overhead can be unacceptable. Mesa provides *inline* procedures for such applications. The call of an inline procedure is replaced by a modified copy of its body. This mechanism eliminates most of the overhead but retains many of the advantages of procedures, such as introducing structure, improving readability and isolating detail.

The foregoing discussion is only an introduction to procedures. The rest of this chapter provides further detail.

5.1. Procedure types

Procedure types are constructed by the syntactic form ProcedureTC, which is defined as follows:

\[
\begin{align*}
\text{ProcedureTC} & \quad ::= \ \text{PROCEDURE} \ \text{ParameterList} \ \text{ReturnsClause} \\
\text{ParameterList} & \quad ::= \ \text{empty} | \ FieldList \\
\text{ReturnsClause} & \quad ::= \ \text{empty} | \ \text{RETURNS} \ \text{ResultList} \\
\text{ResultList} & \quad ::= \ FieldList
\end{align*}
\]

The ParameterList and ResultList are FieldLists and define record types. If either is missing, the corresponding record type is "empty". A procedure type is fully determined by its parameter and result record types.

Default specifications are permitted for fields of a ParameterList, but every field in a ResultList must have an empty DefaultOption, i.e., no default value can be specified for a result.

Some fine points:

Notice that constructors of procedure types require specification of the field lists; it is not possible to use a separately defined record type to specify the complete parameter or result record.

These records, unlike regular records, are not packed; every component is aligned (begins on a word boundary) to allow efficient passing of arguments and results.

A few typical procedure types are shown below:

\[
\begin{align*}
\text{PROCEDURE} & \quad -- \text{takes no arguments; returns no results} \\
\text{PROCEDURE} [x: \text{INTEGER}, flag: \text{BOOLEAN}] & \quad -- \text{takes two arguments} \\
\text{PROCEDURE} \ \text{RETURNS} \ [f: \text{INTEGER}] & \quad -- \text{returns a single value} \\
\text{PROCEDURE} \ \text{RETURNS} \ [f: \text{INTEGER}, b: \text{BOOLEAN}] & \quad -- \text{returns two results} \\
\text{PROCEDURE} [x: \text{INTEGER}] \ \text{RETURNS} \ [y: \text{INTEGER}] & \quad -- \text{takes and returns one value}
\end{align*}
\]

These are all distinct types; none conforms to any of the others.

Values with procedure types are allowed in Mesa; one may have procedure variables, arrays of procedures, records with components that have procedure types, and procedures with procedure parameters or results. The fundamental operations =, # and + may be applied to procedure values with conforming types.

Constructors of procedure types appear most commonly in the declarations of actual procedures, but they may occur wherever a TypeSpecification is valid. Thus a ProcedureTC can appear in such constructs as:

A variable declaration:

\[
\text{ErrorHandler}: \text{PROCEDURE} \ [\text{which}: \text{ErrorCode}] \ \leftarrow \ \text{DefaultHandler};
\]
Chapter 5: Procedures

A type declaration:

First, Rest, Last: ListProc;

A field list (notice the parameters lessThan and swap):

Sort: PROCEDURE [
  first, last: CARDINAL,
  lessThan: PROCEDURE [CARDINAL, CARDINAL] RETURNS [BOOLEAN],
  swap: PROCEDURE [CARDINAL, CARDINAL]
];

An array declaration:

iOps: ARRAY OpNames OF PROCEDURE [T, T] RETURNS [T];

5.1.1. Procedure values and compatibility *

Equivalence and conformance of procedure types are defined in terms of relations between fields of their ParameterLists and ResultLists. If the number of parameters or results differs, one procedure type neither conforms to, nor is equivalent to, another. Otherwise, corresponding fields, matched according to position, are considered. Two procedure types are equivalent if, for each pair of fields, the names are identical (or both are unnamed), the types are equivalent, and both DefaultOptions are empty. One field is compatible with another if the names are identical or if either is unnamed, and if the types are equivalent. A procedure type conforms to another if all corresponding fields are compatible. Default specifications do not affect conformance.

All the assignments in the following example are valid because the types of the procedures conform:

Handle: TYPE = POINTER TO Person;
SignedNumber: TYPE = INTEGER;
Int: TYPE = INTEGER;
ProcA: PROCEDURE [h: Handle, v: SignedNumber];
ProcB: PROCEDURE [h: Handle, v: Int];
ProcC: PROCEDURE [POINTER TO Person, INTEGER];
ProcA - ProcB; ProcA - ProcC;
ProcC - IF flag THEN ProcA ELSE ProcB;

Fine points:

In the current version of Mesa, the name of the component of a single-element parameter or result record is ignored when comparing two procedure types for conformance.

If one procedure type conforms to another, it also conforms freely (section 3.5.3). Free conformance of procedure types is actually defined by the following less restrictive rule: One field is compatible with another if the names are identical or either is unnamed, and if the type of the first freely conforms to the type of the second. One procedure type freely conforms to another if, for the ParameterList, each field of the second is compatible with the corresponding field of the first and, for the ResultList, each field of the first is compatible with the corresponding field of the second.

In the following example, recall that Handle conforms freely to ReadOnlyHandle but not vice versa:

ReadOnlyHandle: TYPE = POINTER TO READONLY Person;
ProcX: PROCEDURE [in: ReadOnlyHandle] RETURNS [out: Handle];
ProcY: PROCEDURE [in: ReadOnlyHandle] RETURNS [out: ReadOnlyHandle];
ProcZ: PROCEDURE [in: Handle] RETURNS [out: ReadOnlyHandle];

-- valid assignments
ProcY - ProcX; ProcZ - ProcY;

-- invalid assignments
ProcX - ProcY; ProcX - ProcZ; ProcY - ProcZ;
In determining the conformance of two procedure types, any default specifications are ignored. Thus it is possible to assign a procedure value to a procedure variable with differently specified defaults. In a procedure call, the type of the variable appearing in the call, not the declaration of the actual procedure, determines the treatment of defaults. Thus the initializing declarations in the following example are valid. Note that the declaration of \textit{Proc2} declares a procedure constant that is indistinguishable from \textit{Proc1} except for the default value of its argument.

\begin{verbatim}
Which: TYPE = \{proc1, proc2, proc3\};
Proc1: PROCEDURE \[p: Which \rightarrow proc1\] =
BEGIN
... END;
Proc2: PROCEDURE \[p: Which \rightarrow proc2\] = Proc1;
Proc3: PROCEDURE \[p: Which\] \rightarrow Proc1;
-- some calls
Proc1[ ]; -- equivalent to Proc1[proc1]
Proc2[ ]; -- equivalent to Proc1[proc2]
Proc3[proc3]; -- note that Proc3[ ] is not legal
\end{verbatim}

5.2. Procedure calls

The syntax for calling a procedure is

\begin{verbatim}
CallStmt :: = Variable \[ComponentList\] . . .
Call :: = Variable \[ComponentList\] . . .
\end{verbatim}

where the \textit{Variable} has some procedure type. Other forms of \textit{Call} are discussed in chapter 8. These specify "catch phrases" for dealing with signals (or errors) that are generated because of the call.

\textbf{ParameterLists} and \textbf{ResultLists} are \textbf{FieldLists} (section 3.4.1). In a call of a procedure, the arguments are packaged into a record. Therefore, a procedure call may use all the syntax for record constructors in passing arguments. Components (arguments) may be specified using either keyword or positional notation; arguments not explicitly specified may be supplied by default. The following calls of \textit{Gcd} are equivalent:

\begin{verbatim}
Gcd[x+1, y] Gcd[m: x+1, n: y] Gcd[n: y, m: x+1]
\end{verbatim}

If the \textbf{ReturnsClause} in a \textbf{ProcedureTC} is not empty, then its \textbf{ResultList} specifies the number and types of the results returned by a procedure of that type. It may be a named or an unnamed \textbf{FieldList} (section 5.3.1 on the \textit{RETURN} statement discusses how it is used).

Procedures that return results must be called from within \textbf{Expressions} that use the results in some way. Such \textit{function references} are valid \textbf{Expressions}. Procedures that do not return results are used in call statements. A procedure that does not return results is called by simply writing a \textbf{CallStmt} as a statement by itself. For example,

\begin{verbatim}
Sort\[first: 1, last: N, lessThan: Younger, swap: Exchange\];
\end{verbatim}
A call statement is ordinarily used to obtain side effects. Most often, these take the form of changes to variables that are not local to the invoked procedure, but they may also involve input or output. A function may also have side effects as well as return results. On occasion, only the side effects are important, and the user wishes to ignore the returned results. An easy way to do this is to assign the result record to an empty extractor:

\[ \{ \} \leftarrow F(x); \]

-- call F and discard its result record.

A call that supplies no arguments is written with an empty constructor, "[ ]". When such a call is itself a statement, the empty brackets may be omitted.

A fine point:

When the call is used as an expression, the empty brackets are mandatory; otherwise, the value of the expression is the value of the procedure, not the value of its results. For example, consider the two procedure variables in the following:

\[
\begin{align*}
\text{Proc1:} & \quad \text{PROCEDURE RETURNS [INTEGER];} \\
\text{Proc2:} & \quad \text{PROCEDURE RETURNS [INTEGER];} \\
\text{IF} \ Proc1 = \text{Proc2} \text{ THEN} \cdots & \quad \text{-- compare the procedure variables} \\
\cdots & \quad \text{-- here the program assigns values to the procedure variables} \\
\text{IF} \ Proc1[1] = \text{Proc2}[1] \text{ THEN} \cdots & \quad \text{-- compare their results (integers)}
\end{align*}
\]

At the time a call occurs, a specific activation is executing, the caller's activation. The effect of a call is to suspend execution of the caller, to create a new activation of the called procedure (including new storage for all parameters and local variables), and to begin execution of that activation. An important consequence of this structuring of procedure control is that all Mesa procedures are inherently capable of being recursive and reentrant.

5.2.1. Arguments and parameters

Arguments are values supplied at call-time; parameters are variables that are local to a given activation. The association of arguments with their parameters amounts to assignment, much as if the following were written:

\[
\begin{align*}
\text{InRec:} & \quad \text{REcord}[\text{arg1: Type1, arg2: Type2, ... }]; \\
\cdots & \quad \text{-- in the caller} \\
\text{InRec} & \leftarrow [\text{arg1: val1, arg2: val2, ... }]; \\
\cdots & \quad \text{-- in the caller} \\
\text{param1: Type1;} & \quad \text{-- in the called procedure} \\
\text{param2: Type2;} & \quad \text{-- in the called procedure} \\
\text{[param1, param2, ... } ] & \leftarrow \text{InRec};
\end{align*}
\]

This is not just an idle analogy. The semantics of assignment accurately describe how arguments are associated with parameters. The following are direct consequences of this:

An argument of a procedure need only conform to its parameter, just as for assignment. All arguments are passed by value in Mesa: i.e., the value of an argument, not its address, is assigned to the parameter. Of course, this value itself can be an address (e.g., if Type1 were POINTER TO TypeX).

5.2.2. Termination and results

A procedure terminates by executing a RETURN statement, which constructs a (perhaps empty) result record. The return operation then terminates execution of the current procedure activation and
restarts the caller from the point at which it was suspended by the call. As part of the return, storage for the parameters and local variables of the returning procedure is released.

Since the value of a procedure is its result record, the components of that record can be assigned to variables using an extractor; alternatively, any single component (if named) can be referenced by a field selector. The procedure \texttt{ReturnExample} returns three integer results and may be used as indicated:

\begin{verbatim}
ReturnExample: PROCEDURE [option: \{1..4\}] RETURNS \{a, b, c: INTEGER\} =
BEGIN ...
-- body defined in section 5.3.1 ...
END;
\end{verbatim}

\begin{verbatim}
x, y, z: INTEGER;
case: \{1..4\};
...
x \leftarrow ReturnExample[case].a;
-- get a component only
[b; y, c; z] \leftarrow ReturnExample[case];
-- assign results by extractor
x \leftarrow (ReturnExample[case].c+1) \mod 10;
-- use c component
\end{verbatim}

If a procedure returns an empty result record, the call does not have a value and can only be used as a statement.

If a procedure returns a single-component result record, extraction and selection are valid. In addition, the component may be (and usually is) accessed directly because of the automatic coercion from a single-component record to its single component. In the following example, the first two calls of \texttt{Gcd} are valid and equivalent; the third illustrates typical use within an expression:

\begin{verbatim}
gcd \leftarrow Gcd[m, n];
-- (coercion)
[gcd] \leftarrow Gcd[m, n];
-- (explicit extraction)
relPrime \leftarrow Gcd[m, n]=1;
-- (coercion)
\end{verbatim}

Some fine points:

In the declaration of \texttt{ReturnExample}, \{a, b, c: INTEGER\} defines a unique type for the result record. Because of the conformance rule for record types (section 3.3.2), it is impossible to declare a variable with that type. If a procedure is to return a record value with a particular type \(T\), it must return a single-component record where that component is a record of type \(T\).

For similar reasons, the result record of \texttt{G} below is not acceptable as the argument record of \texttt{F}.

\begin{verbatim}
F: PROCEDURE [x, y: INTEGER];
G: PROCEDURE [i: INTEGER] RETURNS [x, y: INTEGER];
\end{verbatim}

With these declarations, the call \texttt{F[G[i]]} is not legal; it would be with the following declarations:

\begin{verbatim}
T: TYPE = RECORD [x, y: INTEGER];
F: PROCEDURE [in: T];
G: PROCEDURE [i: INTEGER] RETURNS [out: T];
\end{verbatim}

5.3. Procedure bodies

An actual procedure declaration looks like the declaration of a procedure variable followed by a special kind of = initialization, a \texttt{ProcedureBody}. The \texttt{TypeSpecification} appearing in the declaration determines the type of the body as well as that of the procedure identifier. It may be any \texttt{TypeSpecification} equivalent to a \texttt{ProcedureTC}. \texttt{ProcedureBody} is a special form of initialization defined as follows:

\begin{verbatim}
Initialization \ ::= \ldots | = ProcedureBody | \leftarrow ProcedureBody
ProcedureBody \ ::= \texttt{InlineOption Block}\quad\text{-- see section 4.4 for Block}
InlineOption \ ::= \texttt{empty | INLINE}
\end{verbatim}
If the attribute \texttt{INLINE} appears, the procedure body is an \textit{inline} one; any call of the procedure is replaced by a modified copy of the body (section 5.6).

Only a procedure initialized with \texttt{=} to a \texttt{ProcedureBody} is called an \textit{actual procedure}; its meaning cannot change because it cannot be assigned to. If, however, it is initialized to a \texttt{ProcedureBody} using \texttt{+-} initialization, its value can be changed by assignment, and it is considered a procedure variable. Initialization using \texttt{+-} is not permitted for an inline procedure.

In addition to other statement forms, a procedure body can contain \texttt{RETURN} statements (described in the next section). There is an implicit \texttt{RETURN} at the end of each procedure body if one does not appear explicitly.

A \texttt{ProcedureBody} defines a \textit{scope} for declarations; i.e., identifiers declared within it are local to the procedure and are unknown outside it. There must be no duplicates among the names in a procedure's \texttt{ParameterList}, \texttt{ResultList} and local variables. Names in the \texttt{ParameterList} can be used to write a keyword constructor (section 3.3.4) in a call of a procedure. Similarly, names in the \texttt{ResultList} can be used in keyword extractors (section 3.3.5) and as qualifiers (section 3.3.3) to access the results returned by a procedure. Within a procedure, any named fields of parameter and result records act just as local variables; the former are initialized with the values of the actual parameters. A \texttt{ParameterList} for an actual procedure should be a \textit{named} field list so that the procedure body can reference the parameters.

A fine point:

Although the parameters and results act as local variables within the block that is the procedure body, the scopes are slightly different. The scope of the named parameters and results includes any \texttt{OpenClause}, \texttt{EnableClause} or \texttt{ExitsClause} of that block; the scope of the local variables does not (section 4.4.2).

5.3.1. \texttt{RETURN} statements

There are two basic forms of \texttt{RETURN} statement: \texttt{RETURN} and \texttt{RETURN} followed by a constructor. When either form is executed, control returns to the point from which the procedure was called. In addition, the \texttt{RETURN} can supply results in the form of a constructor conforming to the type of the procedure's \texttt{ResultList}:

\begin{verbatim}
  ReturnStmt ::= RETURN | RETURN [ ComponentList ]
\end{verbatim}

There may be any number of \texttt{RETURN} statements in a procedure body. The form of a \texttt{RETURN} statement depends upon the \texttt{ReturnsClause} in the definition of the procedure type. There are three cases to be considered:

\texttt{no ReturnsClause} (empty result record)

an \textit{unnamed} field list as the \texttt{ResultList}

a \textit{named} field list as the \texttt{ResultList}

If there is \texttt{no ReturnsClause}, the \texttt{ReturnStmt} must be just "\texttt{RETURN}". An explicit \texttt{RETURN} statement can be omitted at the end of the procedure in this case.

If an \texttt{unnamed} field list is used for the \texttt{ResultList}, each \texttt{ReturnStmt} must include a positional constructor. That constructor must match the field list exactly, with one component for every field (omission, elision and voiding are not allowed). In this case, there is no implied return at the end of the procedure.

If the \texttt{ResultList} is a \textit{named} field list, either form of \texttt{ReturnStmt} is acceptable. If no explicit constructor appears, the current values of the named result variables define the value of the result record. An explicit constructor may use either positional or keyword notation; again, omission, elision and voiding are disallowed. A \texttt{RETURN} statement is optional at the end of the procedure; if omitted, an implicit \texttt{RETURN} of the result variables is provided. An example follows:
PROCEDURE [option: [1..A]] RETURNS [a, b, c: INTEGER] =
  BEGIN
    a+b+c=0;
    SELECT option FROM
    1 => RETURN [a: 1, b: 2, c: 3];
    2 => RETURN [1, 2, 3];
    3 => RETURN;
    ENDCASE => b+4;
    c=9;
  END;

5.4. A package of procedures

This section contains an example of a simple module, BinaryTree, which is designed to create and manage a data base structured as a binary tree. It is typical of the ways in which related procedures are packaged together. The example illustrates many of the issues discussed in the previous sections and also introduces the use of modules and interfaces in Mesa.

The binary tree implemented by the example is a data structure containing nodes linked by pointers. Any node points to at most two others (its sons), and a node is pointed to by exactly one other node (its parent). A special root node exists and is referenced by a pointer not in the tree. Every node also contains a value, which for simplicity in the example is just an INTEGER. When the program starts, the tree is empty, and any call to SeekValue will return a count of zero.

The nodes in this particular binary tree are records with four components:

- value  -- an integer value (with unspecified interpretation),
- count  -- the number of duplications of the value in the data base,
- left   -- pointer to a "left" son node (or NIL), and
- right  -- pointer to a "right" son node (or NIL).

There are rules of association between the values and the nodes:

- The first supplied value is entered into the root node.
- A given value may exist in only one node; duplications are counted.
- If node E points to "left" son L, then all the values in the subtree rooted at L are less than the value in E. If node E points to "right" son G, then the values in the subtree rooted at G are greater than the value in E.

When the module is started, the tree is initialized to be empty. Thereafter, the module itself executes no code, but its procedures can be called to alter the tree that it manages. For instance, other modules call PutNewValue to insert new values into the tree.

PutNewValue calls another of BinaryTree's procedures, FindValue, which traverses the tree seeking a node that already has a given value. FindValue may find such a node, or it may fail by reaching a higher-valued node with a NIL left son or a lower-valued node with a NIL right son. If FindValue finds a node with the given value, PutNewValue increments that node's count. Otherwise, PutNewValue sets up a new node and attaches it to the node returned by FindValue.

This strategy is chosen for simplicity, but it can be a poor way to construct a binary tree. For instance, if the values are entered in strictly decreasing order, the tree becomes a linear list of left nodes. To find the lowest-valued node, every node must be examined.

The reader should read the explanation following the example in conjunction with the example itself.
Example 2. A Package of Procedures

DIRECTORY

Storage: FROM "storage" USING [Allocate],
OrderedTable: FROM "orderedtable" USING [UserProc];

BinaryTree: PROGRAM IMPORTS Storage EXPORTS OrderedTable =
BEGIN

-- type definitions and compile-time constants
Node: TYPE = POINTER TO BinaryNode;
BinaryNode: TYPE = RECORD[value: INTEGER, count: CARDINAL, left, right: Node];
nodeSize: CARDINAL = SIZE [BinaryNode];

-- a global variable
root: Node;

-- public (exported) procedures:
Seek Value: PUBLIC PROCEDURE [val: INTEGER] RETURNS [count: CARDINAL] =
BEGIN
  node, found: BOOLEAN;
  RETURN [IF found THEN node.count ELSE 0]
END;

PutNewValue: PUBLIC PROCEDURE [val: INTEGER] =
BEGIN
  alreadyInTree: BOOLEAN;
  [alreadyInTree, node] <- FindValue[val];
  IF alreadyInTree THEN node.count + node.count + 1
  ELSE BEGIN
    nextNode <- Storage.Allocate[nodeSize];
    nextNode <- BinaryNode[val, NIL, NIL];
    IF root = NIL THEN root <- nextNode
    ELSE IF val < node.value THEN node.left <- nextNode
    ELSE node.right <- nextNode;
  END;
END;

EnumerateValues: PUBLIC PROCEDURE[userProc: OrderedTable.UserProc] =
BEGIN
  -- a local procedure (sec. 5.6)
Walk: PROCEDURE [node: Node] RETURNS [keepGoing: BOOLEAN] =
BEGIN -- walk through the tree in order by increasing value using recursion
  RETURN [node = NIL -- don't examine empty (sub)trees
  OR (
      Walk[node.left] AND userProc[node.value, node.count] AND Walk[node.right]) ]
END; -- of Walk
Walk[root]; -- just start enumerating at the root
END;
54: -- a procedure that is private to this module
56: \begin{verbatim}
57: nextNode: Node \leftarrow root; -- always start at the root
58: IF root=NIL THEN RETURN [FALSE, NIL];
59: UNTIL nextNode=NIL DO
60: node \leftarrow nextNode;
61: nextNode \leftarrow SELECT val FROM
62: \begin{cases}
63: \text{node.value} \rightarrow \text{node.left}, \\
64: \text{node.value} \rightarrow \text{node.right},
65: END\text{CASE} \rightarrow \text{NIL};
66: END\text{LOOP};
67: RETURN [val= node.value, node]
68: END;
69: END.
\end{verbatim}
70: -- mainline statements
71: root \leftarrow NIL; -- make tree initially empty
72: END.

5.4.1. The example

Each line of the source code in Example 2 is numbered for convenient reference; other than that, the code could be compiled as it stands.

The body of a \texttt{PROGRAM} module resembles a procedure body: \texttt{BEGIN}, followed by declarations, then some statements, and \texttt{END}. The declarations and statements are both optional, but it would be unusual to omit the declarations.

In this example, the module \texttt{BinaryTree} declares five actual procedures: \texttt{SeekValue} (lines 17-23), \texttt{PutNewValue} (lines 25-38), \texttt{EnumerateValues} (lines 40-53), \texttt{Walk} (lines 43-51) and \texttt{FindValue} (lines 55-68). It also declares two types (\texttt{Node} and \texttt{BinaryNode}), a constant (\texttt{nodeSize}), and a single global variable (\texttt{root}). The scope of these declarations is the entire body of the module (lines 6-72). For example, \texttt{PutNewValue}, \texttt{EnumerateValues} and \texttt{FindValue} all reference the global variable \texttt{root}.

When a module is created and started (chapter 7), the global variables are created and any statements in its body are executed. \texttt{BinaryTree} has just one such statement (line 71), which creates the initial empty tree by assigning \texttt{NIL} to \texttt{root}. Storage for activations of modules is not released when control reaches the end of the main body. Global variables such as \texttt{root} continue to exist and may be used to retain data shared by the actual procedures in the module.

The procedure \texttt{EnumerateValues} has two major distinguishing features: it takes a procedure value as a parameter, and it contains the declaration of a nested procedure (\texttt{Walk}). For each node in the tree, \texttt{EnumerateValues} calls the procedure value \texttt{userProc} that it received as an argument, passing it the value in that node and its replication count. If \texttt{userProc} returns \texttt{TRUE}, the enumeration of the values continues; if it returns \texttt{FALSE}, \texttt{EnumerateValues} terminates and returns to its caller. The values are generated in order from least to greatest.

The nested procedure \texttt{Walk} is recursive and traverses the tree by first traversing the left subtree, then visiting the root, and finally traversing the right subtree. This postorder traversal delivers the values in increasing order (the reader should convince himself that it does). The expression in lines 47-50 depends upon the definitions of \texttt{AND} and \texttt{OR} (section 2.5.3) to terminate the traversal as soon as \texttt{userProc} returns \texttt{FALSE}. The first procedure call occurs only if \texttt{node} is not \texttt{NULL}; the second only if the first is called and returns \texttt{TRUE}; the third only if the first and second are called and both return \texttt{TRUE}. Section 5.6 treats local procedures in more detail.
5.4.2. **Invoking procedures in other modules**

The `DIRECTORY` section at the beginning of a module associates identifiers with file names. The identifier `Storage`, for example, must be the name of a (DEFINITIONS) module that is stored on the file named "storage". Such modules allow the independent development of interface definitions and the sharing of such definitions. `Storage` and `OrderedTable` are said to be included by `BinaryTree`. The optional `USING` clause provides compiler-checked documentation of exactly which identifiers are used in a module but defined in the associated interface.

The `IMPORTS` list (section 7.4.1) on line 5 allows `BinaryTree` to access a procedure (Allocate) defined in the interface `Storage`, which has the following (skeletal) form:

```plaintext
Storage: DEFINITIONS =
  BEGIN
    Allocate: PROCEDURE [size: CARDINAL] RETURNS [POINTER TO UNSPECIFIED];
  END.
```

The example uses explicit qualification (dot notation) to name the `Allocate` procedure (line 33).

The `EXPORTS` list (section 7.4.3) names the single interface `OrderedTable`, which is defined as follows:

```plaintext
OrderedTable: DEFINITIONS =
  BEGIN
    -- types
    -- the interface
    SeekValue: PROCEDURE [val: INTEGER] RETURNS [count: CARDINAL];
    PutNewValue: PROCEDURE [val: INTEGER];
    EnumerateValues: PROCEDURE [userProc: UserProc];
  END.
```

Other modules access the `PUBLIC` procedures in `BinaryTree` (SeekValue, PutNewValue and EnumerateValues) by importing this interface (just as `BinaryTree` imports `Storage`); they have no other access to `BinaryTree`. For example, FindValue is private to `BinaryTree`, so it is only called from within the module (lines 21 and 30). The definition of the type UserProc is included in the interface so that it is publicly available for defining procedures to be passed to `EnumerateValues`. Note that `BinaryTree` also obtains the definition of this type from the interface (line 40).

5.5. **Nested procedures**

Actual procedures may be declared within procedure bodies. A *nested procedure* is one declared within (and local to) some enclosing procedure. Nesting of procedure declarations restricts the scope of the names of the inner procedures. In addition, the enclosing procedure establishes an environment for the inner; this is especially useful when the inner procedure is passed as a parameter.

The value of a nested procedure (and any activation of that value) is "tied" to the local variables of the enclosing procedure and, indirectly, to the local variables of the procedure or module in which the enclosing one is declared, etc. An activation of the nested procedure references those variables available at its point of declaration. A different activation of the enclosing procedure declares a nested procedure with a *different* value, one with its nonlocal variables tied to that other instance of the enclosing procedure.
The following example uses the interface *OrderedTable* defined in section 5.4 and illustrates a typical application of a nested procedure.

**AverageValue**: PROCEDURE RETURNS [INTEGER] =

BEGIN
  sum, n: INTEGER;
  AddValue: OrderedTable.UserProc = -- a nested procedure
    BEGIN
      n = n + count; sum = sum + count'val;
      RETURN [continue: TRUE]
    END;
  sum = n + 0;
  OrderedTable.EnumerateValues[AddValue]
  RETURN [IF n = 0 THEN 0 ELSE (IF sum<0 THEN sum \ (n/2) ELSE sum + (n/2))/n]
END;

The procedure *AverageValue* computes the average value of the *value* fields in the binary tree. It declares and initializes a pair of local variables (n and sum) that are updated by the nested procedure *AddValue* but must have a greater lifetime than any individual activation of *AddValue*. Note that a similar effect could be achieved by making n and sum global variables in this case; the suggested solution restricts their scope (and thus the opportunity for accidental misuse).

Execution of *AverageValue* involves a second nested procedure, the procedure *Walk* within *EnumerateValues*. The latter's parameter userProc serves a purpose similar to that of sum or n in *AverageValue*. Since there is nothing to prevent a recursive call of *EnumerateValues* from some actual procedure corresponding to userProc, making userProc a global variable in the module *BinaryTree* could be disastrous.

**A fine point:**

Because a nested procedure is tied to an activation of the enclosing procedure (even when it references no nonlocal variables), the value of a nested procedure should not be assigned to a variable with a lifetime greater than that of the enclosing procedure instance.

In a sense, all procedures are "local" procedures. They are either local to some enclosing procedure or local to some module (recall that static variables are local to the module declaring them). This nesting can continue to an arbitrary number of levels. (The level is important only to the extent that it influences name scopes, a topic covered in the next section.)

5.5.1. **Scopes defined by procedures**

Each procedure body defines a new scope for names declared in that procedure. Such names represent variables that are local to the body. The scope for a local variable is such that:

(1) the local variable is unknown outside of that procedure body, and
(2) a non-local variable is unknown inside the procedure if its name matches some local variable's name.

Within a procedure body, a block (section 4.4) can be used to further restrict the scope of a local variable. In the following example, scopes for the procedures are indicated by comments:
SomeModule: PROGRAM =
  BEGIN
  var: INTEGER;
  ...
  OUTERPROC: PROCEDURE =
    BEGIN
    var: BOOLEAN;
    ...
  LOCALPROC: PROCEDURE =
    BEGIN
    var: CHARACTER;
    ...
    END;
    ...
    END;
    ...
  END.

Chapter 5: Procedures

5.6. Inline procedures *

An actual procedure is said to be inline if the attribute INLINE appears before the body in the
declaration of that procedure. Any call of the procedure is replaced by an inline expansion, which is
a modified copy of the procedure's body. The code of the procedure and any storage required for
local variables are merged with the code and storage of the calling procedure or module. Thus
inline procedures can be used to eliminate the overhead of a procedure call and return (usually at
the cost of a longer object program).

The rules for creating the expansion are defined so that the presence or absence of the INLINE
attribute has no effect upon the meaning of a program. Execution of the expansion must always
produce a result with the same logical behavior as the result of applying the following operations:

1. For each argument, create a uniquely named variable local to the caller, and initialize that
   variable with the value of the argument.

2. If there is a result record with named fields, enclose the body of the inline procedure in a
   block containing a declaration of each such field.

3. In the resulting block, replace each reference to a field of the parameter list by the identifier
   introduced in the first step for the corresponding argument.

Any global variables of the procedure body refer to the corresponding variables accessible at its
point of declaration, not the point of call.

Some fine points:

A catch phrase can be attached to the call of an inline procedure (section 8.2.1). The arguments are evaluated
outside the scope of the catch phrase.

The Mesa compiler attempts to discover many of the common cases in which "call by name" is equivalent to
the "call by value" substitution described above. When it discovers such a case, the argument is substituted
directly for the corresponding parameter.

The attribute INLINE is never mandatory. Deleting INLINE is always valid, but adding it is not. No
inline procedure can be recursive, either directly or indirectly through a chain of inline procedure
calls. Consider a procedure Proc declared as follows:

  BEGIN
  RETURN [v^v + 3*v + 1]
  END;
Because of its INLINE attribute, *Proc* cannot be used in any of the following situations:

When *Proc* itself is the operand of one of the fundamental operations of assignment (*procVar ← Proc, GeneratorProc[Proc], etc.*) or comparison (*Proc = AnotherProc*).

When *Proc* itself is used as an alternative in a conditional expression, e.g.,

```
(IF predicate THEN Proc ELSE AnotherProc)[x].
```

When *Proc* is the operand of FORK (section 10.1).

When *Proc* is to be exported to an interface (section 7.4.3).

Some fine points:

Since arguments are evaluated before procedures are called, usage such as *Proc[Proc[x]]* does not make *Proc* recursive.

Additional restrictions apply when an inline procedure is declared in a DEFINITIONS module (see section 7.3.3).
CHAPTER 6.

STRINGS, ARRAY DESCRIPTORS,
RELATIVE POINTERS, AND VARIANT RECORDS

This chapter introduces two new data types, strings and array descriptors, discusses relative pointers, and also extends the definition of record types to include variant records.

In Mesa, the type STRING is really "POINTER TO StringBody"; a StringBody contains a length field indicating how long the string currently is, a maxlength field giving the length of that array, and a packed array of characters.

An array descriptor describes the location and length of an array. For ordinary arrays, these are fixed at compile-time. Values of array descriptor type, however, have location and length items that can vary. These array descriptors may represent arrays that are dynamic, but they may also represent ordinary arrays. For efficiency, users often pass array descriptors to procedures instead of passing the entire arrays themselves.

Relative pointers require the addition of a base pointer to obtain an absolute pointer. This allows data structures with internal references that are independent of memory location.

Variant records contain a set of common fields and a variant portion with a specified set of different possible interpretations.

6.1. Strings

In Mesa, a string represents a finite, possibly empty, sequence of characters. Associated with a string are the following:

- length: the number of characters represented. The length may vary at run-time (except for constant strings).
- maxlength: the maximum length. This guarantees that the string is finite. A string's length may vary from zero up to its maximum length.
- text: an indexable sequence of characters.

STRING is a predefined type in Mesa. Each program contains the following relevant pre-declarations:

```plaintext
STRING: TYPE = POINTER TO StringBody;
StringBody: TYPE = MACHINE DEPENDENT RECORD [  
  length: CARDINAL,  
  maxlength: --read only-- CARDINAL,  
  text: PACKED ARRAY [0..0) OF CHARACTER];
```
Suppose $s$ is a STRING variable. Then $s.length$ and $s.maxlength$ refer to the first two components of the string structure currently pointed to by $s$. The type \texttt{StringBody} is "built into" the Mesa language so that the $i$th character of the $text$ array, $s.text[i]$, may be abbreviated $s[i]$. The index type of $text$ in the declaration is used only to specify a starting index of 0. It is better to think of a particular \texttt{STRING} as having an index type $[0..s.maxlength)$.

The value of $s.maxlength$ is assigned when a string structure is created and is a constant: \textit{it may not appear as a LeftSide in the user's program}. However, $s.length$ can be used as a LeftSide. In fact, the user is responsible for setting and changing the length when appropriate (i.e., $s.length$ is meant to reflect the "meaningful" length of the character sequence). Suppose, for instance, that $s$ initially points to an empty string. Then the user might append characters as follows:

$$s[s.length] \gets \text{anotherChar}; s.length \gets s.length + 1;$$

Actually, characters are seldom appended in this manner. The \textit{recommended practice is to use string-handling procedures provided by the Mesa system}. These are documented in the \textit{Mesa System Documentation} and in appendix \textit{C} of this manual.

Since strings in Mesa are actually pointers to string bodies, several strings may refer to the same body. Therefore, a change to that structure would manifest itself in all such strings. Keep the following in mind:

\begin{quote}
\textit{When an item has type STRING, think "string-pointer".}
\end{quote}

A fine point:

While the programmer cannot assign to the $maxlength$ field with an assignment statement, it can be set (along with the length) in a constructor, i.e.

\begin{verbatim}
AllocateWords: PROCEDURE [n: CARDINAL] RETURNS [POINTER TO UNSPECIFIED];
  s: STRING;
  k: CARDINAL;

  s \gets AllocateWords[StringDefs.WordsForString[k]];
  s\rightarrow StringBody[length: 0, maxlength: k, text: ];
\end{verbatim}

This is the way to initialize a \texttt{StringBody} when the space for it comes from some general storage allocator. Note that the $text$ field cannot be set with the constructor since the $ARRAY$ is of length zero in the declaration.

\subsection{String literals and string Expressions}

String literals are written by enclosing the desired sequence of characters in quotation marks, "...". A quotation mark within a string constant is represented by a pair of quotation marks (""). Here are some examples of string literals:

"The first example contains some embedded carriage-returns."

"A quote mark (') isn't a quotation mark('')..."

"!"

"" -- an empty string

A string literal is an \texttt{Expression} of type \texttt{STRING}. Its value is a constant pointer to a constant \texttt{StringBody} in which:

\begin{verbatim}
length = number of characters given, and
maxlength = length
\end{verbatim}
The fundamental operations are defined for string Expressions. They deal with them as pointer values; e.g., \( \oplus \) assigns one string pointer to another string pointer, \( = \) compares two strings for the same pointer value, and \( \# \) compares two strings for different pointer values.

A fine point:

The body of a string literal is ordinarily placed in the global frame of the module in which the literal appears (it is copied from the code when the module is started). Pointers to that body (the actual STRING values) can then be used freely with little danger that the body will move or be destroyed. Unfortunately, this scheme can consume substantial amounts of space in the (permanent and unmovable) global frame area.

If a string literal is followed by 'L (e.g., "abc"L), a copy of the string body is moved from the code to the local frame of the smallest enclosing procedure whenever an instance of that procedure is created. As a corollary, the space is freed and the string body disappears when the procedure returns. This allows smaller global frames, but it is important to insure that pointers to local string literals are not assigned to STRING variables with lifetimes longer than that of the procedure. Programmers should avoid using local string literals until performance tuning is necessary (except perhaps in calls of straightforward output procedures).

### 6.1.2. Declaring strings

String variables are declared like ordinary variables, but there is one additional form of initialization (for strings only):

\[
\text{Initialization} \quad ::= \quad \ldots \; \oplus \; [\text{Expression}] \; \# \; [\text{Expression}]
\]

The Expression must be a compile-time constant Expression of type CARDINAL. At run-time, Mesa creates a string structure with \( \text{maxlength} \) equal to this Expression's value, \( \text{length} \) equal to zero, and text uninitialized. The declared string variable is then set to point to this string structure. If an \text{IdList} is declared with this form of initialization, all of the listed variables initially point to the same string structure.

Some examples:

\[
currentLine: \quad \text{STRING} \; \oplus \; [256];
\]

\[
stringBuffer: \quad \text{STRING} \; \oplus \; [\text{stringMax}+ \text{someExtra}];
\]

This would cause allocation of two string structures in the frame of the program or procedure containing the declarations. The string \text{currentLine} would point to one whose \text{maxlength} is 256. The string \text{stringBuffer} points to the other string structure. (Note that \text{stringMax} and \text{someExtra} must be compile-time constants.) Since the initialization is done with \( \oplus \), it is legal to assign new pointer values to these string variables.

The following are examples of fixed form string initialization:

\[
\text{whatWasThat}: \quad \text{STRING} \; = \; "\text{Eh}?";
\]

\[
\text{goofed}: \quad \text{STRING} \; = \; \text{whatWasThat};
\]

In this case, Mesa would allocate and fill in a string structure for string constant "Eh?". \text{whatWasThat} and \text{goofed}, would be compile-time constants having the same string value; i.e., they would both point to the same string structure. In fact, any other references to the same string literal will point to the same string structure. For example:

\[
\text{huh}: \quad \text{STRING} \; = \; "\text{Eh}?";
\]

String variables can be declared with \( \oplus \) initialization or without any initialization:

\[
\text{stdErrorMsg}: \quad \text{STRING} \; \oplus \; "\text{It seems that we have made a mistake.}"
\]

\[
\text{firstReply, reply}: \quad \text{STRING} \; \oplus \; "\text{Yes}";
\]

\[
\text{oldBuffer, newBuffer}: \quad \text{STRING};
\]
A fine point:
The Mesa system contains procedures you should use when allocating blocks of data. These procedures are helpful for applications involving an arbitrary number of strings or strings of arbitrary length. The procedures are documented in the Mesa System Documentation.

6.1.3. Long strings *

A STRING is just a pointer, so LONG STRING is also a predefined type:

```
LONG STRING: TYPE = LONG POINTER TO StringBody;
```

It is perhaps curious to note that declaring a LONG string says nothing about its actual or potential length.

6.2. Array descriptors

A full description of an array contains several items of information. Consider a typical array declaration:

```
schedule: ARRAY [1..999] OF Date;
```

The following things are known about schedule:

- `base = @schedule[1]`,
- `index type = [1..999]` (a subrange of INTEGER or CARDINAL),
- `minIndex = 1`,
- `length = 999`,
- `component type = Date`

All of these items except `base` are compile time constants, and the value of `base` is the address of a fixed place in the frame, chosen by the compiler. Mesa provides a mechanism for dynamic arrays, where the `base` and `length` can vary at run-time. The implementation does not allow for a variable `minIndex`. Dynamic arrays are implemented by means of Array descriptors.

6.2.1. Array descriptor types

An array descriptor type is constructed much like an array type:

```
DescriptorTC ::= DESCRIPTR FOR ReadOnlyOption ArrayTC |
                DESCRIPTR FOR ReadOnlyOption PackingOption ARRAY OF
                TypeSpecification
ReadOnlyOption ::= empty | READONLY
PackingOption ::= empty | PACKED
```
For example,

\[
\text{events: descriptor for array [1..999] of Date;}
\]

If \text{READONLY} is specified, the contents of the array cannot be changed via the descriptor. In the second form (where no \text{IndexType} is given) the index type is an integer subrange starting at zero.

A value for \text{events} is an array descriptor (a record-like object containing items similar to those described previously for \text{schedule} except that the base is not fixed). The next declaration specifies an array descriptor in which the \text{base} and the \text{length} are variable:

\[
\text{history: descriptor for array of Date;}
\]

Indexing can be used to access components of \text{events} and \text{history} as if they were actual arrays instead of array descriptors (see sec. 3.2.1). Since no index type is specified for \text{history}, it has an \text{indefinite} index type \text{starting at zero with no specified upper bound}.

Two array descriptor types are equivalent if they specify equivalent types for their array elements and if they have equivalent index-sets (or if both index-sets are unspecified). Note that \text{descriptor for array [0..2] of T} and \text{descriptor for array [1..3] of T} are different types, even though the lengths and element types are the same. \text{Expressions} of equivalent descriptor types may be compared for equality (\text{=} or \text{#}).

The rules for assignment are somewhat more relaxed. If \text{a1} has type \text{descriptor for array of T}, and \text{a2} has type \text{descriptor for array [0..10] of T}, then the assignment \text{a1 + a2} is legal, but the assignment \text{a2 + a1} is not.

In any case, for assignments and comparisons, both operands must be array descriptors, and it is the descriptors themselves, not the arrays that they describe which are the values operated on. It would be an error to attempt to assign \text{events} to \text{schedule} because the first is a descriptor and the second is an actual array.

There are three function-like operators relevant to array descriptors: \text{descriptor}, \text{base}, and \text{length}. \text{descriptor} returns an array descriptor result and has three distinct forms which are treated syntactically as built in functions:

\[
\text{BuiltinCall} ::= \text{descriptor[Expression]} | \text{descriptor[Expression,Expression]} | \text{descriptor[Expression,Expression,TypeSpecification]} | \text{base[Expression]} | \text{length[Expression]} | \ldots
\]

The first form takes an argument of some array type; e.g.,

\[
\text{events + descriptor[schedule]};
\]

The result is an array descriptor for \text{schedule}. The second form needs two arguments:

\[
\text{base: pointer to unspecified} -- \text{address of the minIndex component}
\]
\[
\text{length: cardinal} -- \text{number of components}
\]

This form may only be assigned to an array descriptor variable which was declared without an explicit index type.

In those rare situations where the compiler cannot deduce the component type of the descriptor from context, a form of the \text{descriptor} construct is provided which takes three arguments. The third one is a \text{TypeSpecification}, the component type.
The following example provides a fresh array of 64 Dates:

```
Allocate: PROCEDURE [blkSize: CARDINAL] RETURNS [POINTER TO UNSPECIFIED];

... history ← DESCRIPTOR[Allocate[64*SIZE[Date]], 64];
```

The expressions BASE[] and LENGTH[] take one argument (of array descriptor or array type). BASE yields the base of the described array, and LENGTH yields its length. For example:

```
events ← DESCRIPTOR[schedule];  -- describe the entire array
events ← DESCRIPTOR[BASE[schedule], 5];  -- describe the first 5 elements
```

There is no special form for constructing DESCRIPTORS for packed arrays. The PACKED attribute is deduced from context. In the two or three argument form of DESCRIPTOR for packed arrays, the second argument (the LENGTH) is the number of elements.

It is usually more efficient to pass array descriptors as arguments, rather than arrays. Since arguments are passed by value, an array argument causes a copy of the entire array to be made twice (once to put it into an argument record, and once to copy it into a local variable in the called procedure). The next example shows a case in which array descriptors must be used, since passing by value would not work:

```
SortInPlace: PROCEDURE[Table];  -- sorts in situ
Table: TYPE = DESCRIPTOR FOR ARRAY OF INTEGER;
thisArray: ARRAY [0..this) OF INTEGER;
thatArray: ARRAY [0..that) OF INTEGER;
anyTable: Table + DESCRIPTOR[thisArray];

... SortInPlace[anyTable];  -- sorts thisArray
...
SortInPlace[DESCRIPTOR[thatArray]];  -- sorts thatArray
```

A StringBody (sec. 6.1) contains an array, text, of characters. One must be careful when constructing a DESCRIPTOR for this array. Recall that the bounds of text are [0..0). This declaration is used since the actual length of text varies from STRING to STRING. For this reason, the "one argument" form should not be used to construct a DESCRIPTOR for text.

```
textarray: DESCRIPTOR FOR PACKED ARRAY OF CHARACTER;
s: STRING;

textarray ← DESCRIPTOR[s.text];  -- LENGTH[textarray] is incorrect

textarray ← DESCRIPTOR[BASE[s.text], s.length];  -- correct
```

6.2.2. Long descriptors *

The BASE portion of an array descriptor is essentially a pointer. Just as the language allows the type LONG POINTER, it also allows the type LONG DESCRIPTOR. The syntax is straightforward:

```
TypeConstructor :: = ... | LongTC
LongTC :: = LONG TypeSpecification
TypeSpecification :: = ... | DescriptorTC
```

All the standard operations on array descriptors (indexing, assignments, testing equality, LENGTH, etc.) extend to long array descriptors. The type of BASE[desc] is long if the type of desc is long. The LENGTH of an array descriptor is a CARDINAL, whether the descriptor (i.e. its BASE) is LONG or short.

Long array descriptors are created by applying DESCRIPTOR[] to an array that is only accessible
through a long pointer, or by applying \texttt{DESCRIPTOR[,]} or \texttt{DESCRIPTOR[]} to operands the first of which is long. Alternatively, when a short array descriptor is assigned to a long one, the pointer portion is automatically lengthened. Consider the following examples:

\begin{verbatim}
  \begin{verbatim}
  \texttt{d: DESCRIPTOR FOR ARRAY OF \textit{T};}
  \texttt{dd: LONG DESCRIPTOR FOR ARRAY OF \textit{T};}
  \texttt{i, n: CARDINAL;}
  \texttt{pp: LONG POINTER TO ARRAY \{0..10\} OF \textit{T};}

  \texttt{dd \rightarrow DESCRIPTOR[\texttt{pp}];} \quad \text{-- descriptor for the entire array}
  \texttt{dd \rightarrow DESCRIPTOR[\texttt{pp, 5}];} \quad \text{-- descriptor for half of the array}
  \texttt{dd \rightarrow d;} \quad \text{-- automatic lengthening}
  \texttt{pp \rightarrow BASE[dd];} \quad \text{-- BASE of long is long}
  \texttt{n \rightarrow LENGTH[dd];} \quad \text{-- LENGTH is always a CARDINAL}
\end{verbatim}
\end{verbatim}

6.3. Base and relative pointers

Mesa provides relative pointers, i.e., pointers that are \textit{relocated} by adding some base value before they are dereferenced. Relocation has the further effect of mapping a value with some pointer type into a value with a possibly different pointer type. Relative pointers are expected to be useful in such applications as the following:

\textit{Conserving Storage.} Relative pointers can adequately identify objects stored within a zone of storage if the base of that zone is known from context. If the zone is of known and relatively small maximum size, fewer bits are needed to encode the relative pointers. Since a relative pointer and the corresponding base value can have different lengths, relative pointers can be shorter than absolute pointers to the same objects. Overall storage savings are possible when all the base values can be contained in a small number of variables shared among many different object references.

\textit{Providing Movable Storage Zones.} If all interobject references within a storage zone are encoded as zone-relative pointers, the zone itself can be organized to contain only location-independent values. Moving the zone, possibly via external storage, requires only that a set of base pointers be updated.

\textit{Designating Record Extensions.} Sometimes it is convenient to extend a record by appending information (especially variable-length information) to it. Pointers stored in, and relative to the base of, the extended record provide type-safe access to the extensions.

6.3.1. Syntax for base and relative pointers

The syntax for \texttt{BASE} and \texttt{RELATIVE} pointer type constructors is as follows:

\begin{verbatim}
  \texttt{PointerTC ::= Ordered BaseOption POINTER OptionalInterval PointerTail}
  \texttt{BaseOption ::= empty | BASE}
  \texttt{TypeConstructor ::= ... | RelativeTC}
  \texttt{RelativeTC ::= TypeIdentifier RELATIVE TypeSpecification}
\end{verbatim}

In a \texttt{PointerTC}, a nonempty \texttt{OptionalInterval} declares a subrange of a pointer type, the values of which are restricted to the indicated interval (and can potentially be stored in smaller fields). Normally, such a subrange type should be used only in constructing a relative pointer type as described below, since its values cannot span all of memory.
The **BaseOption** BASE indicates that pointer values of that type can be used to relocate relative pointers. Such values behave as ordinary pointers in all other respects with one exception: subscript brackets never force implicit dereferencing (see below). The attribute BASE is ignored in determining the assignability of pointer types.

A **RelativeTC** constructs a *relative pointer* or *relative array descriptor* type. The **Typelentifier** must evaluate to some (possibly long) pointer type which is the type of the basc, and the **TypeSpecification** must evaluate to a (possibly long) pointer or array descriptor type.

Relocation of a relative pointer is specified by using subscript-like notation in which the type of the "array" is the base type and that of the "index" is the relative pointer type. Thus if base is a base pointer and offset is a relative pointer (to T), the form

\[ \text{base}[\text{offset}] \]

denotes an expression of type T, and the value of that expression is \((\text{LOOPHOLE}[\text{base}]+\text{offset})\).

### 6.3.2. A relative pointer example

Consider the **BinaryTree** example from section 5.5. In this program, an ordered table is stored as a binary tree. The tree is stored in the following Mesa data structure:

```markdown
Node: TYPE = POINTER TO BinaryNode;
BinaryNode: TYPE = RECORD[value: INTEGER, count: CARDINAL, left, right: Node];
```

Suppose that the *BinaryNode*’s are allocated from a contiguous region of memory. If the programmer now wishes to put the current state of the ordered table on secondary storage, it is not sufficient to simply write out the region of memory containing the *BinaryNode*’s. This is because the data would make sense only if read back into exactly the same place in memory, a restriction that is difficult to live with. The difficulty stems from the absolute pointers used in the nodes. The problem can be solved by changing the definition of *Node*. If the *BinaryNode*’s are allocated from a region of type **TreeZone**, let

```markdown
TZHandle: TYPE = BASE POINTER TO TreeZone;
Node: TYPE = TZHandle RELATIVEPOINTER TO BinaryNode;
```

The procedure **FindValue** would be written as follows:

```markdown
NullNode: Node = <some value never allocated>;
tb: TZHandle;
root: Node +- NullNode; -- list is initially empty

... FindValue: PROCEDURE [val: INTEGER] RETURNS [inTree: BOOLEAN, node: Node] =
BEGIN
nextNode: Node +- root;
IF root = NullNode THEN RETURN [FALSE, NullNode];
UNTIL nextNode = NullNode DO
  node +- nextNode;
nextNode +- SELECT val FROM
  < tb[node].value => tb[node].left,
  > tb[node].value => tb[node].right,
ENDCASE +- NullNode;
ENDLOOP;
RETURN [val = tb[node].value, node];
END;
```
The other procedures of *BinaryTree* can easily be rewritten to use the new definition of *Node*. The compiler would aid in the translation, since any unrelocated dereferencing of a *Node* would be a compile-time error.

This new implementation of *BinaryTree* has the feature that the *TreeZone* could be moved around in memory, or written and read on secondary storage, and only the base pointer *tb* need be updated to reflect the new position of the *TreeZone*.

### 6.3.3. Relative pointer types

An important topic to consider is the interaction of the relative pointer constructs with the type machinery of Mesa.

A *RelativeTC* constructs a relative pointer type whenever both the *Typelentifier* and the *TypeSpecification* evaluate to pointer types. Let a *RelativeTC* be

\[ \text{Typelentifier RELATIVE TypeSpecification}, \]

where

- *Typelentifier* is of type
  - \[ [\text{LONG}] \text{BASE POINTER} [\text{SubRange}_b] \text{TO} [\text{READONLY}] \ T_b \]
- *TypeSpecification* is of type
  - \[ [\text{LONG}] [\text{ORDERED}] [\text{BASE}] \text{POINTER} [\text{SubRange}_r] \text{TO} [\text{READONLY}] \ T_r \]

and the brackets indicate optional attributes. Relative pointer values must be relocated before they are dereferenced. If *base* and *offset* are base and relative pointers respectively, *offset*, *offset.field*, etc. are compiler-time errors.

If the *TypeSpecification* says *READONLY*, a relocated pointer cannot be a *LeftSide*.

The base type must be designated by an identifier (rather than a *TypeSpecification*) to avoid syntactic ambiguities. Note that the form

\[ \text{LONG Typelentifier RELATIVE TypeSpecification} \]

does not have the effect of lengthening the base type and furthermore is always in error, since *LONG* cannot be applied to a relative type. The type designated by the *TypeSpecification* can be lengthened (to give a relative long pointer) using the form

\[ \text{Typelentifier RELATIVE LONG TypeSpecification} \]

Short relative pointers are never made long automatically. With respect to other operations (assignment, testing equality, comparison if ordered, etc.), relative pointers behave like ordinary pointers. In particular, the amount of storage required to store such a pointer is determined by the *TypeSpecification*.

Some fine points:

In some applications, there is no obvious type for the base pointer, i.e., it might not be possible or desirable to describe a storage zone using a Mesa type declaration. In such cases, a declaration such as

\[ \text{BaseType: TYPE = BASE POINTER TO RECORD [UNSPECIFIED]} \]

generates a unique type that will not be confused with other base types.

The declaration of a relative pointer does not associate a particular base value with that pointer, only a basing type. Thus some care is necessary if multiple base values are in use. Note that the final type of the relocated pointer is largely independent of the type of the base pointer. Sometimes this observation can be used to help
distinguish different classes of base values without producing relocated pointers with incompatible types. Consider the following declarations:

\[
\begin{align*}
\text{baseA}: \text{BaseA} ; \\
\text{baseB}: \text{BaseB} ; \\
\text{OffsetA}: \text{TYPE} = \text{BaseA RELATIVE POINTER TO } T ; \\
\text{OffsetB}: \text{TYPE} = \text{BaseB RELATIVE POINTER TO } T ; \\
\text{offsetA}: \text{OffsetA} ; \\
\text{offsetB}: \text{OffsetB} .
\end{align*}
\]

If \text{BaseA} and \text{BaseB} are distinct types (see the preceding point), so are \text{OffsetA} and \text{OffsetB}. Expressions such as \text{baseA}[\text{offsetB}] and \text{offsetA} + \text{offsetB} are then errors, but \text{baseA}[\text{offsetA}] and \text{baseB}[\text{offsetB}] have the same type.

The base type must have the attribute BASE. Conversely, the attribute BASE always takes precedence in the interpretation of brackets following a pointer expression. Consider the following declarations:

\[
\begin{align*}
\text{p}: \text{POINTER TO ARRAY IndexType OF } \ldots ; \\
\text{q}: \text{BASE POINTER TO ARRAY IndexType OF } \ldots .
\end{align*}
\]

The expression \text{p}[e] will cause implicit dereferencing of \text{p} and is equivalent to \text{p}[e]. On the other hand, \text{q}[e] is taken to specify relocation of a pointer, even if the type of \text{e} is \text{IndexType} and not an appropriate relative pointer type. In such cases, the array must (and always can) be accessed by adding sufficient qualification, e.g., \text{q}[e]; nevertheless, users should exercise caution in using pointers to arrays as base pointers.

Mesa currently supplies no special mechanisms for constructing relative pointers. It is expected that such values will be created by user-supplied allocators that pass their results through a LOOPHOLE or from pointer arithmetic involving LOOPHOLES.

6.3.4. Relative array descriptors

A \text{RelativeTC} constructs a relative array descriptor type whenever the \text{Typedef} evaluates to a pointer type and the \text{TypeSpecific} evaluates to an array descriptor type. Let a \text{RelativeTC} be

\[
\text{Typedef} \text{ RELATIVE TypeSpecific},
\]

where

\[
\begin{align*}
\text{Typedef} & \text{ is of type} \\
& \text{[LONG] BASE POINTER \text{[SubRange]} \text{ TO [READONLY] } T_b .} \\
\text{TypeSpecific} & \text{ is of type} \\
& \text{[LONG] DESCRIPTOR FOR [READONLY] ARRAY } T_i \text{ OF } T_c ,
\end{align*}
\]

and the brackets indicate optional attributes. Relative array descriptor values must be relocated before they are indexed. The relocation yields an expression with type

\[
\text{ARRAY } T_i \text{ OF } T_c .
\]

Relative array descriptor types are entirely analogous to relative pointer types; indeed, values of such types can be viewed as array descriptors in which the base components are relative pointers. If the \text{TypeSpecific} says \text{READONLY}, the relocated array (or its elements) cannot be a \text{LeftSide}.

In the constructor of a relative array descriptor type, the \text{TypeSpecific} must evaluate to a (possibly long) array descriptor type.

In the notation introduced above, a reference to an element of the described array has the form
where $i$ is the index of the element.

Currently, relative array descriptors must be constructed using LOOPHOLES.

### 6.4. Variant records

Section 3.4 discussed "ordinary" record types, where every record object of a single type has the same number and types of components. Such records are not always adequate for programming applications. For example, in the symbol table for a compiler, all the records could have certain components in common: some standard linkage, a string representing the symbol, and a category field indicating whether the symbol stands for an operator, constant, variable, label, etc. Different categories of symbols would then need further components that were not the same in all the records.

Variant records are designed for such applications: a variant record consists of an optional common part followed by a variant part. The common part contains components that are common to all records of this type. The variant part contains the components of each variant of the record.

The specification of a variant record type has the outward appearance of an ordinary record specification: RECORD[field list]. If the record has any common components, these are specified first; then the variant part is specified. (The next section shows how this is done.)

The variant part really represents a set of alternative extensions to the common part. The record type as a whole can be viewed as follows:

```
Common Part                      Variant Part
Field list for the common part ---- |--- field list for variant 1
                                       |--- field list for variant 2
                                       |--- \ldots
                                       |--- field list for variant n
```

Each individual variant is identified by one (or more) adjectives. Suppose defined record type DeRec is declared to have a set of variants named class1, class2, and class3. Then variables could be declared as follows:

```
someClass: DeRec; -- sometimes one class, sometimes another
firstClass: class1 DeRec; -- strictly a class1 DeRec
secondClass: class2 DeRec; -- strictly a class2 DeRec
thirdClass: class3 DeRec; -- strictly a class3 DeRec
```

Types like class3 DeRec are bound variant types. DeRec and class3 DeRec are both type specifications, but the latter is bound to a particular variant. A variable which is declared as a bound variant contains a definite variant; these components can be accessed as if they were common components.

The field list for any variant may, itself, have a variant part (and a variant in that part may have its own variant part, etc.). It is possible to have a type like small class3 DeRec (i.e., the field list for the class3 variant has a variant part which, in turn, has a small variant).

The record, someClass, presents a problem. During the course of execution, someClass might contain a class1, class2, or class3 variant record. (Mesa allocates enough storage to hold the largest variant specified for DeRec type records.) The problem is to determine which variant applies at a given time.
To decide which kind of variant a record object contains, some form of tag is needed. This tag can be specified as part of the record, in which case every such record object will contain an "actual tag" denoting the variant it represents. Instead of storing a simple tag, it may be possible to "compute" the tag value whenever it is needed (possibly by inspecting some values in the common part). Such computed tags are much less safe than explicit ones. For instance, you could refer incorrectly to a "class2" component of someClass when it held a class1 variant record. The result would be undefined.

It is possible to construct an entire variant for the variant part (sec. 6.4.3) by qualifying a constructor (for that variant) with the variant's name (an adjective, in other words). Suppose for example that DeRec has common components c1 and c2 followed by a variant part named vp, and that the class1 variant has components x and y. Then the record constructor below constructs an entire class1 variant:

\[
\text{DeRec}[c1: v1, c2: v2, vp: \text{class1}[x: v3, y: v4]]
\]

Components of an unbound variant can be accessed using the record's tag value (whether actual or computed). A variation of SELECT beginning with the keyword WITH is used for this purpose (sec. 6.4.4). An example follows (given that DeRec has a computed tag):

\[
\text{WITH someClass SELECT currentTag FROM}
\text{
\hspace{1cm}class1 => Stmt1; -- someClass is a bound class1 variant here}
\text{
\hspace{1cm}class2 => Stmt2; -- someClass is a bound class2 variant here}
\text{
\hspace{1cm}class3 => Stmt3; -- someClass is a bound class3 variant here}
\text{ENDCASE;}
\]

### 6.4.1. Declaring variant records

Variant records, like ordinary records, are usually declared in two steps:

- `identifier : TYPE = RecordTC ;` -- define record type
- `idList : TypedList Initialization ;` -- declare the records

**Initialization** for variant records (sec. 6.4.3) is similar to that for ordinary records. The (now complete) definition of RecordTC follows. It extends the partial definition given in section 3.4.2 and includes machine-dependent record types:

\[
\text{RecordTC ::= MachineDependent RECORD [ VariantFieldList ]}
\]

- **MachineDependent**
  - `:: = empty | MACHINE DEPENDENT`
- **VariantFieldList**
  - `:: = CommonPart identifier : Access VariantPart | VariantPart | NamedFieldList | UnnamedFieldList`
- **CommonPart**
  - `:: = empty | NamedFieldList`
- **VariantPart**
  - `:: = SELECT Tag FROM VariantList ENDCASE`
- **Access**
  - `:: = empty | PUBLIC | PRIVATE`
- **Tag**
  - `:: = identifier : Access TagType | COMPUTED TagType | OVERLAID TagType`
The `TypeSpecification` in `TagType` must be equivalent to some enumeration or enumerated subrange type. If the `CommonPart` is not empty, it must be a `NamedFieldList`. If there is no `CommonPart`, the `VariantPart` itself need not be named.

The following example shows many of the possible variations resulting from the above syntax definitions. It is unnecessarily complex for the application, but does show a number of features. It would be worthwhile to parse the declaration yourself using the definitions given above. The example might be used to describe the various "accounts" in a bank; there would supposedly be a table of such entries, one per account.

```plaintext
Service: TYPE = {savings, checking, depositBox};
Account: TYPE = RECORD
  [ number: CARDINAL,
    specifics: SELECT type: Service FROM
      savings => [term: [30..365], intRate: PERCENT, balance: Money],
      checking =>
        [ balance: Money,
          monthlyFee: SELECT COMPUTED {free, notfree} FROM
          notfree => [monthlyFee: Money],
          free => NULL,
          ENDCASE
        ],
    depositBox => [fee: Money, dueDate: Date, paid: BOOLEAN],
  ENDCASE -- no variant can be attached to the ENDCASE
];
```

Each arm of a `VariantPart` specifies a single variant, even if a list of `adjectives` precedes the "=>". An arm may specify `NULL` (as in the case of a `free` checking `Account`) if that variant needs no components of its own. Note that all the arms, including the final one, must end with a comma.

The `adjectives` are identifier constants from some enumeration. Their type can be given explicitly, or implicitly as an enumeration whose members are the adjectives used in the variant part. In any case, the enumerated type is the "tag's" type for a variant part. There are three possible forms for the tag, and they represent:

- an actual tag with an explicit enumerated type (e.g., `type` in `Account`),
- an actual tag implicitly defined (e.g., `easyTag` in `NoCommon` below), or
- a computed tag (e.g., the `monthlyFee` for a `checking Account`).

If an actual tag is used, it is allocated in the common part of the record and may be accessed and used like any other common component, but it may not appear as a `LeftSide`, since that would compromise the type-safeness of such variant records. Not all possible values from the tag's enumeration type have to be used in a variant part; some may be omitted.

An asterisk, ",*", is used to indicate that the type of an actual tag is being defined implicitly by the set of adjectives naming the variants in that tag's variant part. For example, consider the record declaration below:
NoCommon: TYPE = RECORD

[ -- no common part
variantPart: SELECT easyTag: * FROM
  i => [compI: INTEGER],
  j, k => [x, compI: STRING],
ENDCASE];

The implicit type of easyTag is \(\{i,j,k\}\); note: you can’t declare variables of the same type as easyTag.

Computed tags are always unnamed. In fact, they are not really tags at all: when one needs to know which variant a record with a computed tag contains, some computation must be done. Exactly how the variant "tag" is computed is strictly up to the program using it. For instance, to determine whether a checking Account was free or not, the program might look at some property of the Account number (such as whether it was odd or even).

An OVERLAID tag is a special case of a computed tag. The differences occur in the ways in which fields of the record are accessed. See section 6.4.4.

A fine point
Special care must be exercised when declaring a MACHINE DEPENDENT variant record. Recall that MACHINE DEPENDENT records can contain no “holes” between fields. For variant records, this leads to the following rules: if the minimum amount of storage required for each variant is a word or less, each variant must be “padded” to occupy the same number of bits as the longest. Otherwise, each variant must occupy an integral number of words.

6.4.2. Bound variant types

The declaration of a variant record specifies a type, as usual. This is the type of the whole record. The variant record type, itself, defines some other types: one for each variant in the record. Consider the following example:

StreamType: TYPE = {disk, display, keyboard};
StreamHandle: TYPE = POINTER TO Stream;
Stream: TYPE = RECORD
[Get: PROCEDURE[StreamHandle] RETURNS[Item],
Put: PROCEDURE[StreamHandle, Item],
body: SELECT type: StreamType FROM
  disk =>
    [file: FilePointer,
     position: Position,
     setPosition: PROCEDURE[POINTER TO disk Stream, Position],
     buffer: SELECT size: * FROM
       short => [b: ShortArray],
       long => [b: LongArray],
     ENDCASE
    ],
  display =>
    [first: DisplayControlBlock,
     last: DisplayControlBlock,
     height: ScreenPosition,
     nLines: [0..100]
    ],
  keyboard => NULL,
ENDCASE
];
The record type has three main variants: disk, display, and keyboard. Furthermore, the disk variant has two variants of its own: short and long. The total number of type variations is therefore six, and they are used in the following declarations:

```plaintext
r: Stream;
rDisk: disk Stream;
rDisplay: display Stream;
rKeyb: keyboard Stream;
rShort: short disk Stream;
rLong: long disk Stream;
```

The last five types are called **bound variant types**. The rightmost name must be the type identifier for a variant record. The other names are *adjectives* modifying the type identified to their right. Thus, disk modifies the type Stream and identifies a new type. Further, short modifies the type disk Stream and identifies still another type. Names must occur in order and may not be skipped. (For instance, short Stream would be wrong since short does not identify a Stream variant.)

The formal definition of `Typeldentifier` can now be completed (it is only partially defined in Section 2.6.1):

```plaintext
Typeldentifier ::= ... | Adjective Typeldentifier
Adjective ::= identifier
```

where `Adjective` is an adjective of the variant part in the type specified by `Typeldentifier`. Note that the recursive use of `Typeldentifier` in the first line allows a sequence of adjectives.

6.4.3. Accessing entire variant parts, and variant constructors

This section considers accesses to entire variant records (e.g., for initialization), common components of the record (including an actual tag, if present), and the variant part of the record as a whole. The next section covers accesses to individual components in a variant part.

The common parts of each of the variations of a Stream declared in the previous section can be accessed by the normal means (qualification and extraction):

```plaintext
rDisk[..] rLong;  -- aggregate access
rDisk.Get[..] rShort.Get;
.r.body[..] rDisplay.body;
[rDisk.Get, , rDisk.body] rLong;  -- extractor access
```

The actual tag, `type`, in the `body` variant part may also be accessed by qualification:

```plaintext
if r.type=StreamType[keyboard] then Stmt-1;
```

It is also possible to construct values of a variant record type. The syntax of a constructor for a variant part is no different than a normal constructor except that the identifier preceding the "[" must be present and must be one of the adjectives used in defining the variant. For example, some of the following declarations use constructors to initialize the variables (others use different forms of initialization):

```plaintext
myDisplay: display Stream += [myGet, myPut, display[dl,d,h,8]];  
yourDisplay: display Stream += myDisplay;  
currentStream: Stream += myDisplay;  
s: Stream += [SysGet, SysPut, disk[fp, 0, SysSetPos, long[a]]];
```
The keyboard variant of Stream is a NULL variant; so there are no components for that variant in a keyboard constructor:

\[
\text{rKeyb} \leftarrow \text{Stream}[^{\text{Get: Kget, Put: Kput, body: keyboard[]}];}
\]

A side effect of assigning a bound variant value to a variable is that the actual tag of the record is also changed. This is the only way to change the variant contained in a variable (except in the case of a COMPUTED tag) -- it ensures type-safeness. For example, both the following assignments change the type tag for \( r \):

\[
\begin{align*}
\text{r.body} & \leftarrow \text{keyboard[]} ; \\
\text{r.body} & \leftarrow \text{rKeyb.body} ; \quad \text{-- always a keyboard variant}
\end{align*}
\]

If one is assigning a completely bound variant value, \( bv \), say (which could be a constructor, of course) in a AssignmentExpr (section 2.5.4.), then the type of the AssignmentExpr is the type of \( bv \), not the type of the LeftSide, which might not be a bound variant.

A fine point:

The Mesa compiler does not currently allow an entire variant part to occur on the right of an assignment as in the fragment above. Thus, the only way to assign to an entire variant part is via a constructor, not by copying the variant part of an already initialized record. This restriction should be lifted in a later release of Mesa.

6.4.4. Accessing components of variants

When a record is a bound variant, the components of its variant part may be accessed as if they were common components. For example, the following assignments are legal:

\[
\begin{align*}
\text{rDisplay.last} & \leftarrow \text{rDisplay.first} ; \\
\text{rDisk.position} & \leftarrow \text{rShort.position} ;
\end{align*}
\]

If a record is not a bound variant (e.g., \( r \) in the previous section), the program needs a way to decide which variant it is before accessing variant components. More importantly, however, this must be type-safe. For this reason, the process of discriminating among possible variants and then accessing within a variant part is combined in one syntactic form, called a discrimination, which is a generalization of SELECT.

A discrimination closely mirrors the form of SELECT used to declare a variant part. However, the arms in a discriminating SELECT contain statements or Expressions, and, within a given arm, the discriminated record value is viewed as a bound variant. Therefore, within that arm, its variant components may be accessed. The following syntax equations complete the earlier partial definitions given in sections 4.3.1 and 4.3.3:

\[
\begin{align*}
\text{SelectVariant} & ::= \text{WITH OpenItem SELECT TagItem FROM ChoiceSeries ENDCASE FinalStmtChoice} \\
\text{ChoiceSeries} & ::= \text{AdjectiveList} \rightarrow \text{Statement} ; | \\
\text{TagItem} & ::= \text{empty} | \text{Expression} \quad \text{-- the actual tag is used} \\
\text{FinalStmtChoice} & ::= \text{empty} | \rightarrow \text{Statement}
\end{align*}
\]
Chapter 6: Strings, Array Descriptors, Relative Pointers, and Variant Records

SelectExprVariant ::= WITH OpenItem SELECT TagItem FROM ChoiceList
| ... ENDCASE => Expression

ChoiceList ::= AdjectiveList => Expression, |
| AdjectiveList => Expression, ChoiceList

ChoiceList ::= AdjectiveList => Expression, |
| ChoiceList AdjectiveList => Expression, |
| Expression

OpenItem ::= Expression | AlternateName: Expression

-- from sec. 4.4.2

The value discriminated is the one given in the WITH clause, which behaves just like an OPEN clause (sec. 4.4.2) to simplify naming the record value in the arms of the SELECT. The following example discriminates on \( r \).

WITH \( \text{strm} \): \( r \) SELECT FROM
| display => BEGIN
| \( \text{strm}.\text{first} \)+= \( \text{strm}.\text{last} \);
| \( \text{strm}.\text{height} \)+= 73;
| \( \text{strm}.\text{nLines} \)+= 4;
| END;
| \( \text{disk} \) => WITH \( \text{strm} \) SELECT FROM
| \( \text{short} \) => \( b[0] \)+= 10;
| \( \text{long} \) => \( b[0] \)+= 100;
| ENDCASE;
| ENDCASE => \( \text{strm}.\text{body} \) += \( \text{disk} \) \([\text{GetFp}["\text{Alpha}"], 0, \text{SysSetPos}, \text{short}[\text{]}] \);

In the first example, suppose \( r \) contains a variant record of \( \text{display} \) \( \text{Stream} \) type. Then the first arm is chosen by this SELECT. Within it, \( \text{strm} \) (but not \( r \)) is considered a record of \( \text{display} \) \( \text{Stream} \) type; so all components of the \( \text{display} \) variant may be accessed in the statement chosen by that arm (as they are in the example).

Suppose \( r \) contains a variant record of \( \text{disk} \) \( \text{Stream} \) type. Then the actual tag has the value \( \text{disk} \), and the second arm is chosen. In this example, only one of the \( \text{disk} \) components is accessed, its variant part. The inner SELECT uses variant record \( \text{strm} \). Within the outer arm, Mesa knows that \( \text{strm} \) is a record of \( \text{disk} \) \( \text{Stream} \) type. Consequently, the tag implicitly used for this SELECT is the tag specified for \( \text{that} \) type (namely, \( \text{size} \)).

If the tag value is \( \text{short} \), then the chosen arm accesses component \( b \) in the \( \text{short} \) \( \text{disk} \) \( \text{Stream} \) variant record; if it is \( \text{long} \), then the chosen arm accesses component \( b \) in the \( \text{long} \) \( \text{disk} \) \( \text{Stream} \) variant record.

However, the ENDCASE for the inner SELECT could have accessed components that are common to a \( \text{disk} \) \( \text{Stream} \) (\( \text{file}, \text{position}, \text{SetPosition}, \) variant part \( \text{buffer} \), and actual tag \( \text{size} \); plus all the original common components: \( \text{Get}, \text{Put}, \) variant part \( \text{body} \), and actual tag \( \text{type} \)).

Suppose, lastly, that \( r \) does not contain a variant record of \( \text{display} \) \( \text{Stream} \) or \( \text{disk} \) \( \text{Stream} \) type. Then the outer ENDCASE statement is chosen. This statement accesses the common component \( \text{body} \) (the entire variant part is considered a common component), and gives the record a specific variant type (\( \text{short} \) \( \text{disk} \) \( \text{Stream} \)) by wholesale assignment. An ENDCASE may only access common components; \( \text{it may not access components of variants in the given type} \).
If the labels on an arm of a discrimination identify more than one variant structure, the record is not considered to be discriminated within that arm and only the common fields are accessible (cf. ENDCASE).

Since the outer variant part of Stream was declared using an actual tag, the tag’s value is obtained from the record itself, and no Expression follows the keyword SELECT (both SELECTs above have this form).

The Expression in the WITH clause (actually in the OpenItem) must represent either a variant record or a pointer to a variant record (e.g., r in the above). The alternate name is essentially a synonym for that Expression (e.g., strm in the above). If it is a pointer, however, the alternate name designates a record value, not a pointer value in each arm of the SELECT. In the following example, the display arm is correct, and the disk arm is in error:

```plaintext
rp: StreamHandle;
proc: PROCEDURE [StreamHandle];
WITH sRec: rp SELECT FROM
    display => proc[@sRec];  -- CORRECT
    disk    => proc[sRec];   -- WRONG
ENDCASE;
```

An open item with no alternative name opens a name scope so that components can be accessed with implicit qualification (as in the inner SELECT of the first example), but then no further levels of WITH...SELECT using the same record can be done within such a WITH...SELECT. The type of the open item’s Expression indicates the nature of the record’s variant part, including whether the tag is an actual or computed tag, its enumerated type, and the names of each variant (i.e., the adjectives) in the variant part.

If a computed tag had been used, the program would have to supply an Expression following SELECT to determine the variant. This Expression’s value would have to be an adjective in the applicable variant part. For example, assume that tbl[i] in the following has type checking Account (sec. 6.4.1); then this is a legal (if not very sophisticated) discrimination for it:

```plaintext
WITH this: tbl[i] SELECT (IF (this.number MOD 2) = 0 THEN free ELSE notfree) FROM
    free     => NULL;
    notfree  => AddToBill[this.monthlyFee];
ENDCASE;
```

If a given arm of a discrimination is labelled by identifier constants corresponding to more than one variant of the record, only the common fields of the record are accessible within that arm.

The record value in a WITH clause must not represent a completely bound variant (which is really not a variant at all). For example, a valid discrimination for a disk Stream record, aDiskStream, follows:

```plaintext
WITH aDiskStream SELECT FROM
    short => b[0]+10;
    long  => b[0]+100;
ENDCASE;
```

It would be illegal to rewrite this as follows:

```plaintext
WITH alt: aDiskStream SELECT FROM -- WRONG!
    disk => WITH alt SELECT FROM
        short => b[0]+10;
        long  => b[0]+100;
ENDCASE;
ENDCASE;
```
An OVERLAID record is a special case of a computed variant record in that there is no explicit tag field in the record. The fields of the individual variants may be accessed using a "computed" WITH construct in the same manner as a COMPUTED record. In addition, any field name of a variant that is unambiguous (i.e., it appears in only one variant) can be referenced without discrimination. In essence, the programmer is telling the compiler "When I use a fieldname, you can trust me that the record has the proper variant." Consider the following example:

```
TrustMe: TYPE = RECORD[
    SELECT OVERLAID * FROM
        one => [c: CHARACTER, i: CARDINAL, next: POINTER TO TrustMe],
        two => [b: BOOLEAN, next: POINTER TO TrustMe],
        three => [s: STRING],
    ENDCASE];

Lc   -- legal
Lb   -- legal
Lnext -- illegal, both variants one and two contain such a field.
```

A fine point

In the declaration of TrustMe above, the two next fields were of the same type, but occupied different positions within the record. Even if they did occupy the same position, one could still not refer to Lnext. The ambiguity is one of variant, not of value.
Large programs in Mesa are constructed by linking or binding together individual modules. A module is the basic unit of compilation and also the smallest, self-contained, executable program unit. Most of this chapter deals with how separate modules are put together to build large systems; i.e., it deals with *programming in the large* as opposed to *programming in the small* (which is what this manual has discussed so far).

There are two fairly distinct kinds of modules. *Definitions* modules serve primarily as "blueprints" or specifications for how the parts of a system will fit together. During compilation they provide a common (and therefore consistent) set of definitions which can be referenced by other modules being compiled. The second kind of modules, called *programs*, contain actual data and executable code. Program modules can be loaded and interconnected to form complete systems.

Mesa compiles a program module's source code (which is just a text file) into an *object module*. An object module is a binary file containing object code, symbol table information, and data structures to be used in connecting (also called *binding*) this module together with others. Compiling a definitions module produces symbol table information only, which may then be used in compiling other modules (either definitions or program modules).

7.1. Interfaces

An *interface* is a connector between programs; it allows code in one module to access parts of other modules—specifically, procedures, signals (chapter 8) and variables. Interfaces are defined by definitions modules (section 7.3). They contain declarations for public items and allows the compiler to check for type matching across inter-module references. The interface, considered as a record, also proves a convenient data structure for efficient binding together of programs.

The procedures that implement a given abstraction are often collected in a single interface. For example, an interface for an allocator might consist of the names and types of the procedures for allocating and freeing blocks of storage, and pointers to shared blocks of storage. The data types required by these procedure types (for parameters and return values) are usually defined in the same definitions module. Such *non-interface* types are available for reference when compiling other modules, but are not considered part of the interface specified by that definitions module.

At compile time, a program module containing calls on procedures defined by some interface must *import* the definitions module that specifies that interface. This enables the compiler to check the agreement of types of parameters and return values on calls from that module with their counterparts in the definitions module (i.e., as defined in the interface). *Importing the interface at compile time does not, however, link the procedure references in the program module to actual
procedures in some other module(s). That actual binding occurs later when the compiled module is linked with other compiled program modules to make a system (sec. 7.7).

The actual implementation of an interface is usually provided by a single program module, although it may be realized by a group of modules, each supplying a part. In any case, if a program module implements (all or part of) the interface specified by a definitions module, it is said to export that interface. The procedures and variables in that program corresponding to the ones in the exported interface must be type-compatible with them (sec. 5.2). The compiler checks that this is so.

After compilation, a program module contains a set of virtual interface records, one for each imported interface, and a set of export records, one for each exported interface (a single program module can implement more than one interface). Binding a group of modules together into a system then involves associating virtual interface records with exported interfaces for all the modules in the group.

The following definitions module, IODefs, provides a minimal (and unrealistic) interface to a computer terminal:

```plaintext
IODefs: DEFINITIONS =
BEGIN
  -- Interface definitions
  ReadChar: PROCEDURE RETURNS [CHARACTER];
  ReadLine: PROCEDURE [input: STRING];  -- reads from terminal into input
  WriteChar: PROCEDURE [output: CHARACTER];
  WriteLine: PROCEDURE [output: STRING];

  IOPkg: PROGRAM;

  -- Non-interface definitions
  CR: CHARACTER = 015C;  -- an ASCII Carriage-Return character
END. -- IODefs
```

The interface record for IODefs is imported by the following Copier program module. The program reads lines from the terminal and retypes them. When the user types a line beginning with a period, it writes a parting message and stops:

```plaintext
DIRECTORY IODefs: FROM "IODefs";

Copier: PROGRAM IMPORTS IODefs =
BEGIN OPEN IODefs;
  -- allows simple references to items from IODefs
  input: STRING ~ [256];  -- 256-character string to hold input lines typed by user

  -- the mainline part of the program starts here:
  DO
    ReadLine[input];  -- read a line into input
    IF input[0] = '.' THEN EXIT;  -- quit if first character is a period
    WriteLine[input];  -- otherwise copy it back to the user
  ENDDO;
  WriteLine["End of example."];
  WriteChar[CR];  -- final output
END.  -- Copier
```

```plaintext
-- leave terminal on a new line
```
The skeleton of a module that implements the IODefs interface follows. It EXPORTS IODefs and IMPORTS nothing:

DIRECTORY
IODefs: FROM "IODefs";

IOPkg: PROGRAM EXPORTS IODefs =
-- this module contains the actual procedures for the interface specified by IODefs.
BEGIN
terminalState: {off, on, hung} = off;  -- initial state of the terminal
ReadChar: PUBLIC PROCEDURE RETURNS [CHARACTER] = BEGIN ... END;
ReadLine: PUBLIC PROCEDURE [input: STRING] = BEGIN ... END;
WriteChar: PUBLIC PROCEDURE [output: CHARACTER] = BEGIN ... END;
WriteLine: PUBLIC PROCEDURE [output: STRING] = BEGIN ... END;
END. -- IOPkg

The next step towards running the above modules as a system requires binding them together. Binding is the process of matching up virtual import records with real export records. A separate language, C/Mesa, is used to describe binding. This language has a syntax similar to Mesa's, but is much smaller. C/Mesa "programs" are compiled ("processed" might be more accurate) by a program called the Binder. The C/Mesa source code is called a Configuration Description (CD), and compiling one results in a Binary Configuration Description (BCD) file. An object file produced by the Mesa compiler is actually a very simple BCD containing just one module's object code and binding information.

BCD files can be loaded and run. (Actually, it is the individual modules in the BCD that are loaded). This loading also alters all the BCD's virtual import records to hold real procedure descriptors (sec. 5.2), signals, and pointers to program frames. Then the modules comprising the BCD can all be started (details in sec. 7.8). The following CD describes a system of three modules: Copier, IOPkg, and Driver.

MakeCopierSystem: CONFIGURATION
CONTROL Driver =
BEGIN Copier, IOPkg; Driver, END.

This configuration specifies how the Copier, IOPkg, and Driver object modules are to be bound together. Simply listing their names is all that is usually required in a CD. Now the Mesa loader could load the complete program using the BCD file for MakeCopierSystem. Driver is named as the CONTROL module for the BCD, so starting the loaded BCD would actually result in starting Driver, which follows:

DIRECTORY
IODefs: FROM "iodefs",
Copier: FROM "copier";

Driver: PROGRAM IMPORTS Copier, IODefs =
BEGIN
START IODefs.IOPkg;  -- so its variables (e.g., terminalState) are initialized
START Copier;  -- to initialize its variables and run its mainline code
END.

This example is simple, but MakeCopierSystem and Driver would still be simple even if the system had 50 modules instead of just two. For this example, they seem like excess baggage, but for a larger system, they are invaluable because:
they describe exactly how the various modules are bound together and initialized;
(b) C/Mesa allows Mesa’s compile-time checks on types to extend to binding time;
(c) loading and linking with this scheme can be very efficient.

We can now give the details of Mesa definitions and program modules. Section 7.7 discusses C/Mesa and how it is used.

### 7.2. The fundamentals of Mesa modules

The complete syntax for a module is the following:

```plaintext
CompilationUnit ::= Directory -- optional
                 ModuleName : ModuleHead = GlobalAccess
                 ModuleBody

Access ::= empty | PUBLIC | PRIVATE

Directory ::= empty | DIRECTORY IncludeList ;

ExportsList ::= empty | EXPORTS IdList -- sec. 7.4.6

FileName ::= stringLiteral -- sec. 6.1.1

GlobalAccess ::= Access -- sec. 7.4.3

ImportsList ::= empty | IMPORTS InterfaceList
                    -- sec. 7.4.6

IncludeItem ::= identifier : FROM FileName |
               identifier : FROM FileName USING [ IdList ]

IncludeList ::= IncludeItem | IncludeItem, IncludeList

ModuleBody ::= Block . -- sec. 4.4
               -- note the terminating period

ModuleHead ::= DEFINITIONS LocksClause ImportsList ShareList |
               ProgramTC ImportsList ExportsList ShareList

InterfaceItem ::= identifier | identifier : identifier

InterfaceList ::= InterfaceItem | InterfaceList , InterfaceItem

LocksClause ::= empty |
               LOCKS Expression |
               LOCKS Expression USING identifier : TypeSpecification

ModuleName ::= identifier

ProgramTC ::= PROGRAM ParameterList ReturnsClause |
              MONITOR ParameterList ReturnsClause LocksClause

ShareList ::= empty | SHARES IdList -- sec. 7.4.6
```

A definitions module can serve a twofold purpose: it can define an interface, and may contain declarations of constants and types. Definitions modules are further discussed in section 7.3.

The text of a program module $X$ implicitly defines a frame type, $FRAME[X]$. Values of this type are created dynamically by loading $X$ and can only be accessed indirectly; i.e., a program may have variables of type $POINTER$ TO $FRAME[X]$, but never of type $FRAME[X]$. A module’s frame contains storage for its variables, along with some system overhead.

We will first deal with the initial syntactic unit which is common to all modules (the Directory clause), then with definitions modules as a whole. After these sections there is a complete example
including
  a \texttt{DEFINITIONS} module,
a \texttt{PROGRAM} module that implements it,
a client program that uses it, and
a configuration that binds the programs together into a system.

7.2.1. Including modules: the \texttt{DIRECTORY} clause

The source code for a given module may tell the compiler to \textit{include} previously compiled modules for one or more of the following reasons:

- It might need to use some of the symbols defined by those modules.
- It may need to import the interfaces defined by those modules.
- It might refer to instances of such modules after they are loaded in order to \texttt{START} them, to make new instances of them, or to access their data.

Suppose module \texttt{A} is included in module \texttt{B}. This means that when compiling \texttt{B}, the compiler must have access to \texttt{A}'s object file (i.e., \texttt{A} must have been compiled previously) in order to access its symbol table to obtain information needed by \texttt{B}. \textit{Warning: including a module is not simply an insertion of text from one module into another—it is important to read these sections carefully to use this capability correctly.}

The following is a simple, but complete \texttt{DEFINITIONS} module:

\begin{verbatim}
SimpleDefs: DEFINITIONS =
BEGIN
  limit: INTEGER = 86;
  Range: TYPE = [limit .. limit];
  Pair: TYPE = RECORD[first, second: Range];
  PairPtr: TYPE = POINTER TO Pair;
END.
\end{verbatim}

Suppose that the above source code is contained in a file named "SimpleDefs.mesa". After compilation, anyone who has a copy of the object file for \texttt{SimpleDefs} (which will be named "SimpleDefs.bcd" by the compiler), may then include it in other modules. The ".bcd" portion of the file name stands for Binary Configuration Description (sec. 7.6.3) of which a compiled module is the simplest example. The ".bcd" part of the name need not be specified in the directory section (see below).

A module that includes other modules begins with a \texttt{Directory}, which performs two functions:

1. It associates a Mesa identifier with the name of an object module (which does not necessarily look like a Mesa identifier).
2. It checks that the given identifier matches the \texttt{ModuleName} in the module whose object file is named.

Here is an example of a \texttt{DIRECTORY}:

\begin{verbatim}
DIRECTORY
  SimpleDefs: FROM "simpledefs",
  StringDefs: FROM "stringdefs";
\end{verbatim}
7.2.1.1. Enumerating items from an included module: the USING clause

A module may list the symbols it expects to access from an included module in the USING clause of an Includetem. If a USING clause is present, it must list all of the symbols to be included; the compiler will not allow access to any symbol not in the list. Warnings will be issued for symbols appearing in the list which are not referenced in the module. In this way, the USING clause accurately documents which symbols are defined in each included module.

Here is an example of a DIRECTORY with a USING clause:

DIRECTORY
  SimpleDefs: FROM "simpledefs" USING [Range, Pair],
  StringDefs: FROM "stringdefs'';

A module with this DIRECTORY statement would be allowed to use the symbols Range and Pair defined in SimpleDefs, but would not be allowed to use any other symbols defined in SimpleDefs. Access to symbols defined in StringDefs is not restricted.

The USING clause only allows and restricts access to symbols. Actual references to the symbols must be made in one of the ways described below.

7.2.2. Accessing items from an included module

This section describes the ways of accessing symbols defined in an included module:

An identifier, p, defined in a definitions module, Defs, can be named in an including module, User, in one of two ways.

Explicit qualification: p can be named as Defs.p in User.

OPEN clauses: In the scope of an OPEN clause of the form "OPEN Defs", the simple name p suffices.

The remainder of this section gives more detail on these methods.

7.2.2.1. Qualification

In the following example, qualification is the only access method used:

DIRECTORY
  SimpleDefs: FROM "SimpleDefs'';
TableDefs: DEFINITIONS =
BEGIN
  limit: INTEGER = 256; -- this has no connection with SimpleDefs.limit
  Index: TYPE = [0..limit);
  StringTable: TYPE = ARRAY Index OF STRING;
  PairTable: TYPE = ARRAY Index OF SimpleDefs.Pair;
END.

SimpleDefs.Pair means "the item named Pair in SimpleDefs." As a rule, qualification provides more readable code than do the other methods for specifying the use of predefined symbols. However, it can be inconvenient if there are many such occurrences because two identifiers have to be written instead of one.

No names are included automatically when only explicit qualification is used. For example, if
TableDefs had not declared limit, Mesa would not have used the one in SimpleDefs. An error would then result when TableDefs was compiled (because limit is needed in the declaration of the type Index).

Any module that includes TableDefs may use only the symbols defined by it, but to use Pair, that module would have to include SimpleDefs (as in the next section’s example). Declared symbols in the included module do not include record component names: they are part of a record’s type specification and can be used wherever the record type is known.

A qualified name may denote a type defined in an included module (e.g., the type SimpleDefs.Pair in the example in the previous section). Thus the syntax for Typeldentifier includes the case

Typeldentifier :: = ... | identifier . identifier

7.2.2.2. OPEN clauses

The following program, TableUser, includes both SimpleDefs and TableDefs. It accesses names from SimpleDefs by qualification, but uses an OPEN clause to access items from TableDefs:

DIRECTORY
SimpleDefs: FROM "simpJedefs" USING [Pair],
TableDefs: FROM "tabledefs";
TableUser: PROGRAM =
BEGIN OPEN TableDefs;
  vIndex: INTEGER ~ limit;
  vPair: PairTable;
  vString: StringTable;
  StorePair: PUBLIC PROCEDURE[t: SimpleDefs.Pair] RETURNS[ok: BOOLEAN] =
BEGIN
  ok ~ vIndex <= limit;
  IF ok THEN vPair[vIndex] ~ t;
END;
  StoreString: PUBLIC PROCEDURE[s: STRING, v: Index] =
BEGIN
  vString[v] ~ s;
  vPair[vIndex+v] ~ NIL;
END;
END.

In the scope of the OPEN clause, the names limit, StringTable, PairTable, and Index are those in TableDefs. The scope of these OPEN clauses follows the same rules as the OPEN clauses for records described in section 4.4.2. In fact, a single OPEN clause can contain OpenItems that open either modules or records.

TableDefs could have been in an OPEN clause anywhere that one is permitted. This feature can be used to help the readers of a program. For example, if the names from TableDefs were only needed in the procedure StoreString, we could put an "OPEN TableDefs" on its BEGIN rather than on the BEGIN for the whole module. This would localize the region of the program where a reader would have to consider whether an identifier is from an included module or not.

Fine point:

Note that qualification is still required to reference SimpleDefs.Pair even though Pair appears in the USING clause.
7.2.3. **Scopes for identifiers in a module**

The use of identifiers appearing in modules falls into two broad categories, *defining occurrences* (e.g., to the left of the ":" in a declaration), and *name references* (such as the appearance of a name in an *Expression*). *Scope rules* determine which defining occurrence goes with a given reference. In Mesa, these rules are *lexical*, i.e., they depend only on the textual structure of the module at compile-time.

A *name scope* is always a contiguous region of a module (e.g., everything between a `BEGIN ... END` pair, or between a `[..]` pair) and may contain other scopes nested within it. The first scope rule is the following:

Within a single scope (excluding scopes nested within it), there can be at most one defining occurrence of a given identifier.

An important corollary of this rule is that a given identifier is either undefined in a scope or it has exactly one meaning.

A *qualified* name reference demands an exact context for its qualified identifier. For example,

- `SimpleDefs.Pair` -- (sec. 7.2.2.1) qualification by module name; context is `SimpleDefs`
- `rDisk.Get` -- (sec. 6.3.3) record qualification; context is `disk Stream`, the type of `rDisk`
- `winner.party` -- (sec. 3.4) pointer qualification; context is `Person`, the reference type of `winner`

The rule of scope is simple for a qualified reference:

The qualified identifier is associated with its symbol definition in the specified scope (if there is no such defined name, the qualified identifier is undefined and there is an error).

An *unqualified* name reference occurs within a sequence of nested scopes (as indicated below). The rule of scope is

Use the *innermost* scope that defines the referenced identifier (if none of the scopes do so, the identifier is undefined and there is an error).

New name scopes are created by the following:

- `OPEN` clauses
- `Blocks` with declarations
- enumerated types and their subrange types
- record types that use named field lists
- procedure types that use named parameters or results
- actual procedures
- exit regions for loops and compound statements
- the heads and arms of discriminating `SELECT` statements

`OPEN` clauses may introduce multiple name scopes, which are nested (inner-to-outer) in order from right to left. Consider the following revision of the earlier `TableUser` module:
Notice that we no longer need qualification for the parameter type of procedure StorePair. When the compiler encounters identifier Pair, it finds the needed symbol definition in the symbol table for included module SimpleDefs. The path by which it found this is the following: it looked for such a definition in the current module, but failed there; it then tried the next outer scope, which according to the OPEN was TableDefs; not finding Pair there either, it went on to the next (and outermost) scope given by the OPEN, namely, SimpleDefs, at which point a defining occurrence was found.

Localizing the scope of identifiers from included modules is so important that we recommend the following naming guidelines:

1. Place a USING clause on items in the DIRECTORY. This collects in one place a list of all symbols referenced from each included module. The list is always accurate because the compiler checks it on each compilation.
2. Use explicit qualification as the normal way of naming an external item.
3. Use an OPEN clause on the smallest possible scope when explicit qualification becomes too verbose. It is unusual for items from an included module to be accessed with high frequency everywhere in a module; most often, there are clusters of references to them. An OPEN clause takes advantage of this clustering and alerts a reader to it.

7.2.4 Implications of recompiling included modules

Consider a set of modules Adefs, User1, and User2 where Adefs is included in User1 and User2. (For simplicity, assume User1 and User2 include only module Adefs.) Suppose Adefs and User1 have already been compiled, but before User2 is compiled, Adefs is recompiled for some reason. Then User1 must also be recompiled.

In general, recompiling Adefs will invalidate the current version of User1. This is obvious when Adefs undergoes significant change between compilations, but it may also be true when seemingly innocuous changes are made. In fact, if User1 uses record or enumeration types defined by Adefs, the current version of User1 is invalidated when Adefs is recompiled, even if no changes are made to its source code!

For example, suppose Adefs defines RECORD type Account which is used by User1 as the type of r1 and by User2 for r2. Normally, one would expect these records to have the same type. If events occur as follows, however, they will not:

Adefs is compiled.
User1 is compiled including (old) Adefs.
Adefs is recompiled.
User2 is compiled including (new) Adefs.

The record types for r1 and r2 will differ because of the way Mesa guarantees uniqueness for record types. The compiler associates a "time stamp" (e.g., time of definition) with each record type. Old Adefs defined Account at one time and new Adefs defined it a later time; this makes them different (non-equivalent) record types which only "look" the same.
Consider the case where Defs1 is included in Defs2, and Defs2 is included in User1. (For simplicity, assume that Defs2 includes only Defs1 and User1 only Defs2.) Suppose that Defs1 is recompiled and then Defs2 is recompiled. Then User1 should also be recompiled. The reason for this is the uniqueness of record types defined in Defs2 and used by User1.

The (re)compilations of Defs1, Defs2, and User1 must occur in a specific order: first Defs1, then Defs2, and finally User1. Suppose, however, that User1 included Defs2 plus another module Defs3, and suppose Defs3 included Defs1. The following diagram illustrates these dependencies. Modules which are included are above modules which include them. The rule for avoiding errors due to incorrect compilation order is the following: a module may not be (re)compiled until all the modules above it have been.

Thus, User1 should be recompiled after Defs2 and Defs3 have both been (re)compiled. The order in which Defs2 and Defs3 are compiled is unimportant, however. *Moral:* There is an important partial order defined on modules by their inclusion relations.

### 7.3. DEFINITIONS modules

Generally, a DEFINITIONS module contains a set of related definitions. There are compile-time constants, types, and procedure and signal definitions. There are also declarations of so called interface variables (section 7.3.1). These definitions are used by the program(s) that implement those procedures, and they are used by programs that only wish to call on those procedures. Separating definitions from implementations allows programs that call those procedures to be independent of changes in implementation. The definitions in a DEFINITIONS module fall into two classes:

- **Interface elements:** definitions for interfaces (procedures, signals, programs and interface variables), and
- **Non-interface elements:** compile-time constants (this includes TYPE definitions)

There are no special rules about which valid compile-time constants may be used other than the issues surrounding compilation order (sec. 7.2.4). External interface definitions, however, are different. Normally, a declaration such as

```haskell
SampleProc: PROCEDURE [i: INTEGER];
```

declares a procedure variable. In a DEFINITIONS module, however, its effect is to define the type and name of a procedure component of the interface specified by that definitions module. Section 7.6 contains an example of a DEFINITIONS module that defines procedure interfaces.

In the same manner, signals, errors, and programs can be declared in a DEFINITIONS module as elements of its interface type. A signal or error declaration is treated just like a procedure as an interface element. A program definition as an interface element is discussed in section 7.4.3.
7.3.1. Interface variables

Just as procedures are often declared in one program module and called from another, there are
often applications that require sharing of non-procedural variables, such as common data structures.
One means to allow common variables is via interface variables. These are variables that are defined
in a DEFINITIONS module, but must be exported by some implementing program before they actually
exist. Variable declarations in interfaces are like those in programs, with one additional option.

```
Declaration ::= IdList : Access ReadOnlyOption EntryOption
  TypeSpecification Initialization ; | ...

ReadOnlyOption ::= empty | READONLY
```

The READONLY option can be attached to a variable only in an interface. If this option is specified,
importers can read the variable but not update it; otherwise, importers are able to read and update
the variable freely. Note that a READONLY variable is not necessarily constant; it can be changed
from within the program module exporting the variable. Consider the following example:

```
Defs: DEFINITIONS =
  BEGIN
    var1: T;
    var2: READONLY T;
    ...
  END.

Impl: PROGRAM EXPORTS Defs =
  BEGIN
    var1: PUBLIC T;
    var2: PUBLIC T;
    ...
    var1 ~ e1; ... var2 ~ e2;
    ...
  END.

User: PROGRAM IMPORTS Defs =
  BEGIN
    ...
    Defs.var1 ~ e3;
    ...
    IF Defs.var2 = e4 THEN ...
    ...
  END.
```

In this example, the exporter (Impl) provides storage for the variables var1 and var2. Within Impl,
both are ordinary variables, e.g., var2 can be updated. By importing Defs, the client (User) gains
access to the storage for var1 and var2 but cannot update var2. Although Defs.var1 and Defs.var2
are referenced indirectly, through pointers in User initialized by the binder or loader, those pointers
are invisible to the importer and are always automatically dereferenced.

An interface variable must be declared with the attribute PUBLIC in the exporter. It must not be
declared with fixed ("=") initialization. Assignment ("~") initialization is permissible; note,
however, that such initialization is not performed until the exporting module is started, and
reference to an interface variable does not cause a start trap in the exporter (see section 7.8.3).
A fine point: If an interface component has a PROCEDURE, SIGNAL, ERROR or PROGRAM type, any importer expects that component to contain an actual value (e.g., a procedure descriptor), not a pointer to such a value. Thus an interface variable cannot have one of these types. For example, if the declaration

```
Proc: PROCEDURE [...];
```

appears in a DEFINITIONS module_defs, the declaration

```
Proc: PUBLIC PROCEDURE [...] = BEGIN ... END;
```

is valid in an exporter of_defs; the declaration

```
Proc: PUBLIC PROCEDURE [...] + SomeProcedure;
```

is not. (In the latter case, a variable containing a pointer to Proc could be exported.)

In addition to providing common access to data structures, interface variables make default fields (section 7.3.2) and inline procedures (section 7.3.3) more useful in DEFINITIONS modules; that is, interface variables can be used within these definitions to access nonconstant information in the exporter. Interface variables used for this purpose must be PUBLIC in the exporter but can be declared with the attribute PRIVATE in the DEFINITIONS module. This convention is strongly recommended to prevent unintended direct sharing of the variables by clients of the interface. Note that code within an interface cannot update a READONLY component of that interface.

We will see in section 7.4.2 another means of accessing the variables of another program module, that of an implicit POINTER TO FRAME[...], whose value is initialized by the loader. This offers an alternative to interface variables, but leads to compilation dependencies such as those described in section 7.2.4. In choosing between the two methods, note the following:

Each imported interface variable introduces a separate pointer in the link area of the importer (see Section 7.7, page 125). On the other hand, each access to such a variable generally requires less code and is faster than an access to a field of an imported frame.

The use of interface variables introduces less severe compilation dependencies; recompilation of the exporter does not require recompilation of all importers.

Compromise positions are also possible. Information to be shared can be grouped into a smaller number of variables with record or array types, and those variables can be exported. This reduces the number of pointers but increases the amount of (potentially changing) structure in the interface.

### 7.3.2. Default fields in interfaces

One valuable use of interface variables is in default values for procedure argument records (section 5.1). Default arguments can contain references to procedures (INLINE or otherwise), signals and interface variables that are components of the same interface. These references are bound to values in the same instance of the imported interface as the one supplying the procedure definition itself. References to non-constant components of other interfaces require that those interfaces be imported by the DEFINITIONS module and all its users (see section 7.4.4). For example, an interface could contain:

```
  globalQ: PRIVATE Queue;  -- an interface variable
  Add: PROCEDURE [i: Item, q: Queue ~ globalQ];
```

This declaration allow users to Add items to globalQ, but not to access the variable directly. Thus the statement

```
  Defts.Add[myItem];
```
is equivalent to

\[ \text{Defsl.Add}_D[\text{myItem}, \text{Defsl.global}_Q] ; \]

A fine point:

If a program imports two instances of Defsl, say D1 and D2, the defaulted value for q will be from the same instance as the called procedure. In other words,

\[ D1.AAdd[\text{myItem}] \text{ is equivalent to } D1.AAdd[\text{myItem}, D1.global_Q] , \]

\[ D2.AAdd[\text{myItem}] \text{ is equivalent to } D2.AAdd[\text{myItem}, D2.global_Q] , \]

In a program module, default arguments of exported procedures (and signals, etc.) require special attention. Because of the assignment rule, the DefaultOptions specified in the exporter and in the interface are not required to agree. If the implementer and the clients are to behave identically with respect to defaults, the same DefaultSpecification must appear twice.

For example, if the interface has the declaration

\[ \text{Proc: \text{PROCEDURE} \ [x: T + e_1] ;} \]

and the exporter has the declaration

\[ \text{Proc: \text{PUBLIC PROCEDURE} \ [x: T + e_2] } = \text{BEGIN} \ldots \text{END;} \]

then within the exporter, Proc[] means Proc[e2]; within an importer of the interface, Proc[] means Proc[e1]. The language requires only that the types of e1 and e2 be compatible with T; if they should also provide the same value, the programmer must ensure this.

7.3.3. Inline procedures in interfaces

An INLINE procedure can be declared within a DEFINITIONS module. Any caller of that procedure must import an instance of the corresponding interface.

Within a DEFINITIONS module, the body of an INLINE procedure can contain references to procedures (INLINE or otherwise), signals and interface variables that are components of the same interface. These references are bound to values in the same instance of the imported interface as the one supplying the inline procedure itself. References to non-constant components of other interfaces require that those interfaces be imported by the DEFINITIONS module and all its users (see section 7.4.4).

Interface components with the attribute PRIVATE are visible to the bodies of inline procedures declared within the same DEFINITIONS module. An inline procedure referencing such components can be imported into a PROGRAM module in which those components are not visible. Interface components used only as free variables or default arguments of imported procedures should not be mentioned in the corresponding USING clauses. For example, suppose that an interface contains the following declarations:

\[ \text{DomainFault: SIGNAL;} \]
\[ \text{Proc: PROCEDURE [CARDINAL] RETURNS [T];} \]

\[ N: \text{CARDINAL} = 100; \]

\[ \text{Table: PRIVATE ARRAY } [0 \ldots N] \text{ OF POINTER TO T;} \]
Chapter 7: Modules, Programs, and Configurations

BEGIN
  IF i ~ IN [0..N] THEN ERROR DomainFault;
  RETURN IF Table[i] = NIL THEN Proc[i] ELSE Table[i]*
END;

Note that the body of an INLINE procedure (IProc) can contain references to constants (N), interface procedures (Proc), signals (DomainFault) and interface variables (Table) in the same interface. If Defs1 is an instance of this interface, Defs1.IProc references Defs1.Table, calls Defs1.Proc, etc. An importer cannot reference Table directly because of the PRIVATE attribute but can reference it indirectly through IProc.

It is not legal for a PROGRAM module Progl to call an inline procedure defined in some other PROGRAM module Prog2, even if Progl imports a POINTER TO FRAME[Prog2].

### 7.3.4 Usage hints for INLINE procedures in interfaces *

Expansion of inline procedures can cause internal data structures of the compiler to grow rapidly; indiscriminate use of the INLINE attribute can substantially degrade compiler performance or cause tables to overflow. The current compiler has been organized so that inline expansion is particularly efficient, and incurs little added overhead, in the following circumstances:

The INLINE procedure, with an arbitrarily complex body, is defined within a PROGRAM module and called exactly once in that same module. (Thus introducing named procedures for clarifying and structuring a program can be cheap when such procedures are called only once.)

The INLINE procedure, defined either in a DEFINITIONS module or a PROGRAM module and called an arbitrary number of times, is very simple, with no local variables, no named output parameters, and no side effects.

The debugger cannot set breakpoints within, or display the expanded source text of, an INLINE procedure (although it can display the local variables resulting from the expansion). Debugging can be easier if the INLINE attribute is used only as needed and is specified after initial testing has been successfully completed.

### 7.4. PROGRAM modules: IMPORTS and EXPORTS

A PROGRAM module may contain
- definitions of constants and types (just like a DEFINITIONS module),
- declarations of variables,
- actual procedures and signals (chapter 8), and
- executable statements of its own (i.e., not part of procedure bodies within it).

At run time, a loaded module (also called an instance of the module—sec. 7.6.3) has a frame which provides storage for its declared variables and for connections to other modules' procedures and signals.
These connections are called *interface records*, and there is one for each interface imported by the module. The Mesa binding process fills in these interface records with procedure descriptors, signal codes, and pointers to program frames and exported variables in other modules.

### 7.4.1. IMPORTS, interface types, and interface records

The IMPORTS list for a program declares which interface records the program needs and associates them with DEFINITIONS modules (called *interface types*). *Interface records and interface types are different! A program may only access non-interface elements using an interface type, but can access all elements (both interface and non-interface) when using an interface record.*

The names of interface records are declared in a PROGRAM module's IMPORTS list. The identifier preceding a "::" in the list names an interface record, while the name following that same "::" must name an interface type. In the following example, names ending with *Rec* specify interface records, and names ending in *Def* specify interface types:

```plaintext
DIRECTORY Deftl: FROM "defs1", Defs2: FROM "defs2";

Prog: PROGRAM IMPORTS IRec: Defs1, I2Rec: Defs2 =
BEGIN . . . END.
```

Within the body of *Prog*, references like *Deftl.x* are only valid if *x* is a non-interface element of *Deftl*. However, *IRec.x* can refer to any element *x* of *Deftl*, whether interface or non-interface. This distinction is necessary because a call on a procedure, *proc*, defined in *Deftl*, must refer to the actual descriptor in the interface record *IRec* at run time, not just to its compile-time definition.

Omitting the name of an interface record in an IMPORTS list and giving only the name of an interface type means that the record's name should be the same as the type's. For example, writing

```plaintext
Prog: PROGRAM IMPORTS IRec: Defs1, Defs2 = . . .
```

is the same as writing

```plaintext
Prog: PROGRAM IMPORTS IRec: Defs1, Defs2: Defs2 = . . .
```

Then, within the body of *Prog*, *Deft2* refers to an interface record. In fact, it is impossible thereafter to refer to the interface type *Deft2*, although one can still refer to the interface type *Deft1* because its name has not been reused.

Sometimes one needs to have access to more than one instance of an interface record at run time. For example, the Mesa compiler needs to access one instance of a symbol table package for the program that it is compiling, and at least one for the symbol tables for modules included by that program. This can be done by defining a number of interface records for a single interface type, as in the following:

```plaintext
DIRECTORY SymDefs: FROM "SymDefs";

PartOfCompiler: PROGRAM IMPORTS mainSym: SymDefs, auxSym: SymDefs =
BEGIN . . . END.
```

Within the body of *PartOfCompiler*, one would access an interface element of *SymDefs* named *LookUp* for the main symbol table as *mainSym.LookUp*, and for the auxiliary symbol table as *auxSym.LookUp*. 
7.4.2. **Importing program modules**

Any module can include a program module \( X \) by naming \( X \) in its directory. One can use \( X \) to declare program variables of type `POINTER TO FRAME[X]`. `FRAME[X]` is not a valid type because frames cannot be embedded in other structures. One may obtain a value for `POINTER TO FRAME[X]` is from a system procedure as described in the *Mesa System Documentation*, or by importing.

A module can import a program \( X \) by naming it in its IMPORTS list. For example,

```plaintext
DIRECTORY XProg1: FROM "XProg1", XProg2: FROM "XProg2";
Prog: PROGRAM IMPORTS frame1: XProg1, XProg2 =
    BEGIN . . . END.
```

This has an effect similar to declaring

```plaintext
frame1: POINTER TO FRAME[XProg1] = . . . ;
XProg2: POINTER TO FRAME[XProg2] = . . . ;
```

except that these constant frame pointers will be filled by the Mesa binder. (However, the declaration for `XProg2` could not actually be written as a valid Mesa statement because of the ambiguity inherent in the two occurrences of `XProg2` in it.)

Such imported program constants are the analogs of interface records. More can be done with them than with program types (just as one can access more with an interface record than with an interface type). In particular, one can access variables and procedures with a frame pointer (as well as the compile-time constants to which a program type provides access). Also, one can execute the module instance corresponding to a frame pointer using `START` and `RESTART` (sec. 7.8.2) and create additional instances of it using `NEW` (sec. 7.8.1).

Accessing values in a program frame as described above treats the frame as a record with its variables and its procedures as its components. The price paid for such close coupling with a program is that the importer must be recompiled whenever the program is.

7.4.3. **Exporting interfaces and program modules**

A module can *export* an interface if it provides `PUBLIC` procedures, signals, errors, or variables whose names and types match those of interface elements in a `DEFINITIONS` module. In addition, the program can export itself as part of an interface if its name appears there with an appropriate `PROGRAM` type. In all these cases, the compiler checks that the type of each exported element is assignment compatible (sec. 2.3) with the type of the corresponding interface element.

A single program module need not provide implementations for all the items in an interface. This allows two or more modules to cooperate in completely defining an interface. In such a case, it is common for each of the cooperating modules to use interfaces elements provided by the others. It can do so by importing and exporting the same interface.

7.4.4. **IMPORTS in DEFINITIONS modules**

Recall from section 7.2 that a definitions module can contain an `ImportsList`. One interface, say `Defs2`, must import another, `Defs1`, if `Defs2` requires access to a nonconstant component of the latter, e.g., to a procedure (inline or otherwise) or interface variable in `Defs1`. All interfaces mentioned in the `ImportsList` of a `DEFINITIONS` module must be unnamed. Any importer of `Defs2` must also import an unnamed instance of `Defs1`; in establishing the final binding, all unnamed instances are matched. For example:
Mesa Language Manual

Deft2: DEFINITIONS IMPORTS Deft1 =
BEGIN
...
Proc2: PROCEDURE [...] = INLINE'
BEGIN
... IF Deft1.var # v THEN Deft1.Proc[...]; ...
END;
...
END.

Prog: PROGRAM IMPORTS Deft1, Deft2, AnotherDeft1: Deft1 =
BEGIN
...
Deft2.Proc[...]; -- expansion references Deft1.Proc, not AnotherDeft1.Proc
...
END.

Note that Prog must import an instance of Deft1, even if it makes no other mention of it, and one instance of Deft1 must be unnamed.

7.5. Controlling module interfaces: PUBLIC and PRIVATE

Every name defined in a module possesses an Access attribute, either PUBLIC or PRIVATE (the module in which a name is defined is called its home module). These are used to determine whether a name may be referenced when its home module is included by some other module. A PUBLIC name can always be used; a PRIVATE name cannot generally be used, except by modules which specify that they SHARE the included (PROGRAM or DEFINITIONS) module. The former modules are called non-privileged modules, and the latter are called privileged modules. A variable's home module is, of course, privileged.

Generally speaking, an Access may be specified

(a) anywhere a name can be declared. This includes normal declarations, named field lists (for records or parameter lists), preceding SELECT in a record's variant part, and the declaration for an actual tag in a variant part.

(b) preceding the TypeSpecification in a type definition.

In addition, an Access may be specified

(c) at the beginning of a module (the GlobalAccess), to provide a default Access for any identifier in that module when one is not given explicitly for the identifier.

The syntax in the following section is intended to supersede earlier definitions of the same constructs only by showing where attributes may be inserted. Otherwise, the earlier versions are correct. Each syntax definition is followed by examples of its use.

7.5.1. Access attributes in declarations

The following three subsections deal with the placement of Access options in declarations, field lists, and in variant records.
7.5.1.1. Declared names

The form of Declaration specifying an Access for its declared names is as follows:

\[
\text{Declaration} ::= \text{IdList} : \text{Access TypeSpecification} \text{ Initialization} ;
\]

Examples:

\[q1, q2: \text{PUBLIC INTEGER} \leftarrow 0;\]
\[\text{Mine: PRIVATE TYPE = \{yes, no, maybe\};}\]

\text{Mine} can only be used in (i.e., seen from) privileged modules. To non-privileged modules it is not visible at all.

7.5.1.2. Names specified in field lists

The forms for specifying Access in a NamedFieldList (sec. 3.4.1) are as follows:

\[
\text{NamedFieldList} ::= \text{IdList} : \text{Access FieldDescription} | \text{NamedFieldList , IdList} : \text{Access FieldDescription}
\]

\[
\text{FieldDescription} ::= \text{TypeSpecification} | \text{TypeSpecification} + \text{Expression}
\]

Example:

\[\text{blk: PUBLIC RECORD}\]
\[
\begin{align*}
\text{[} & \\
\text{a: INTEGER,} & \\
\text{b: PRIVATE INTEGER} \leftarrow 1234, & \\
\text{c, d: BOOLEAN,} & \\
\text{e: PRIVATE BOOLEAN} & \\
\text{]};
\end{align*}
\]

A non-privileged module could only access components \(a, c,\) and \(d\) in this case, and then only using qualified references such as \(\text{blk.a}\). Within a non-privileged module, extractors and constructors cannot be employed for a record type with any PRIVATE components.

7.5.1.3. Names for variant parts and for tags in variant records

The forms for specifying Access in a VariantFieldList or Tag (sec. 6.3.1) are as follows:

\[
\text{VariantFieldList} ::= \text{CommonPart identifier : Access VariantPart} | \text{VariantPart} | \text{NamedFieldList} | \text{UnnamedFieldList}
\]

\[
\text{CommonPart} ::= \text{empty} | \text{NamedFieldList}
\]

\[
\text{VariantPart} ::= \text{SELECT Tag FROM VariantList} \quad \text{-- same as in sec. 6.3.1 END CASE}
\]

\[
\text{Tag} ::= \text{identifier : Access TagType} | \text{COMPUTED TagType}
\]

\[
\text{TagType} ::= \text{TypeSpecification} *
\]
Example:

VarRec: PUBLIC TYPE = RECORD
   [  
      link: POINTER TO VarRec,  -- public common component
      vpl: SELECT tgl1: PRIVATE Etype FROM  -- public variant, private tag
        adj1 =>
          [  
            youGet: ThisItem,
            iGet: PRIVATE SELECT tgl2: * FROM  -- a private variant part
              ...
            ENDCASE
          ]
        adj2 =>...
    ENDCASE
];

Suppose a non-privileged module has a record of type VarRec. Then it could access variant part vpl but neither tag tgl1 nor variant part iGet. This only prevents it from referring to tgl1 by qualification; it may still use a discriminating SELECT (which implicitly accesses tgl1) for records of type VarRec. Thus, an adj1 arm of such a WITH...SELECT could access component youGet. However, it would be unable to access component iGet in any case.

Notice that the only way that the tag of a variant can be changed is by writing a variant constructor (sec. 6.3.3).

7.5.2 Access attributes in TYPE definitions

The form for specifying a Typelentifier whose defined type has an explicit Access is as follows:

Declaration :: = ...|IdList : TYPE = Access TypeSpecification ;

Example:

OurType: PUBLIC TYPE = PRIVATE RECORD[comp1: INTEGER, comp2: BOOLEAN];

A non-privileged module could declare records of type OurType, but it could not access the record components. The module could, however, pass values of type OurType as parameters, receive them as results from procedures, and use them as operands of a fundamental operation (+, =, #).

The Access in this form could be specified as PUBLIC, but this would be pointless (if OurType is PUBLIC then its type would be PUBLIC by default; if OurType is PRIVATE then its type attribute is irrelevant). Note: Only names specified within the defined type are affected by this form of attribute specification. Consequently, it is intended for use only when defining record types and is just a factorization: the PRIVATE could have been written after each inner colon; also, specific fields can be made accessible by writing PUBLIC internally, as shown below:

AlmostPrivateType: PUBLIC TYPE = PRIVATE RECORD
   [  
      comp1: PUBLIC INTEGER,  -- overrides outer PRIVATE
      comp2: BOOLEAN
   ];

7.5.3. Default global access

If, as in section 7.4.1, a declaration specifies an Access for a name, then that unilaterally determines its Access. If not, the given item receives a default Access. The default may be specified by the programmer in the GlobalAccess for a module; otherwise one is assumed (for a program module, the normal default is PRIVATE, for a DEFINITIONS module, it is PUBLIC). For example,

\[ M1: \text{PROGRAM} = \text{PUBLIC} \quad \text{-- specified GlobalAccess} \]
\[ \quad \text{BEGIN} \]
\[ \quad \ldots \]
\[ \quad \text{END.} \]
\[ M3: \text{PROGRAM} = \quad \text{-- PRIVATE (by default)} \]
\[ \quad \text{BEGIN} \]
\[ \quad \ldots \]
\[ \quad \text{END.} \]

7.5.4. Accessing the PRIVATE predefined symbols of other modules

A module may be privileged to use PRIVATE items in an included module by using a Shares clause: this contains a list of the (included) module names whose PRIVATE symbols it needs to access. Consider the Friendly module below:

\[ \text{DIRECTORY} \]
\[ \quad \text{SpecialDefs: FROM "specialdefs";} \]
\[ \quad \text{StandardDefs: FROM "standarddefs";} \]
\[ \quad \text{PrivateDefs: FROM "private";} \]
\[ \text{Friendly: PROGRAM SHARES PrivateDefs, SpecialDefs =} \]
\[ \quad \text{BEGIN} \]
\[ \quad \ldots \]
\[ \quad \text{END.} \]

In this case, Friendly can use PRIVATE symbols defined by PrivateDefs and SpecialDefs but not the PRIVATE symbols of StandardDefs. There is no particular significance to the ordering of module names listed after SHARES. Any kind of module may use SHARES (but it ought to be one that is "friendly", to say the least).

7.6. The Mesa configuration language, an introductory example

This section discusses C/Mesa, the Mesa configuration language, first by example, and then more rigorously by syntactic definition and detailed semantics. It ends with a number of detailed examples which explore some of the more intricate parts of C/Mesa.

We first present an example consisting of three Mesa modules:

- An interface (a DEFINITIONS module),
- an implementor for it (a PROGRAM module),
- and a client for the implementation (also a PROGRAM module).

The example is presented here to show the relationships among definitions, implementors, and clients. Following it will be a sequence of example configurations for systems constructed from this implementor and client. The line numbers in the left margin are provided for ease of reference and are not part of the source code. First the interface:
7.6.1. Lexicon: a module implementing LexiconDef

The following module (Lexicon) implements the LexiconDefs interface. That is,

(a) Lexicon declares PUBLIC procedures FindString, AddString, and PrintLexicon, which have procedure types conforming to their counterparts in the DEFINITIONS module;

(b) Lexicon EXPORTS the interface LexiconDefs.

Lexicon IMPORTS three interfaces: SystemDefs, IODefs, and StringDefs. The USING clauses of the DIRECTORY note which procedures are defined in each.

Details on these and other Mesa system interfaces are contained in the Mesa System Documentation.

The code for Lexicon follows. For reading convenience, any references to procedures from imported interfaces are in boldface.

```
PROGRAM Lexicon;
IMPORTS SystemDefs, IODefs, StringDefs
EXPORTS LexiconDefs =

BEGIN
  Node: TYPE = RECORD [llink, rlink: NodePtr, string: STRING];
  NodePtr: TYPE = POINTER TO Node;
  Comparative: TYPE = {lessThan, equalTo, greaterThan};
  root: NodePtr = NIL;

  FindString: PUBLIC PROCEDURE [s: STRING] RETURNS [BOOLEAN] =
  BEGIN RETURN [SearchForString[root, s]]; END;

  SearchForString: PROCEDURE [n: NodePtr, s: STRING]
  RETURNS [found: BOOLEAN] =
  BEGIN
    IF n = NIL THEN RETURN [FALSE];
    SELECT LexicalCompare [s, n.string] FROM
    lessThan  => found + SearchForString[n.llink, s];
    equalTo   => found + TRUE;
    greaterThan => found + SearchForString[n.rlink, s];
    ENDCASE;
  RETURN [found];
  END;
```
AddString: PUBLIC PROCEDURE [s: STRING] =
BEGIN InsertString[root, s]; END;

InsertString: PROCEDURE [n: NodePtr, s: STRING] =
BEGIN
  NewNode: PROCEDURE RETURNS [n: NodePtr] =
  BEGIN
    OPEN SystemDe/s;
    n = AllocateHeapNode[SIZE [Node]];
    mt ← Node[string: AllocateHeapString[s.length], llink: NIL, rlink: NIL];
    StringDefs.AppendString[n.string, s];
    RETURN;
  END;
  RETURN;
END;

IF n = NIL THEN root ← NewNode[] -- then just return
ELSE
  SELECT LexicalCompare[s, n.string] FROM
  lessThan  => IF n.rlink # NIL THEN InsertString[n.rlink, s] ELSE n.rlink ← NewNode[] END;
  equalTo   => NULL; -- already there; just return
  greaterThan => IF n.rlink # NIL THEN InsertString[n.rlink, s] ELSE n.rlink ← NewNode[] END;
ENDCASE;

LexicalCompare: PROCEDURE [s1, s2: STRING] RETURNS [c: Comparative] =
BEGIN
  n: CARDINAL = MIN [s1.length, s2.length];
  i: CARDINAL;
  FOR i IN [0..n) DO
    SELECT LowerCase[s1[i]] FROM
    <LowerCase[s2[i]] => RETURN [lessThan];
    >LowerCase[s2[i]] => RETURN [greaterThan];
    ENDCASE;
    ENDLOOP;
  c ← SELECT s1.length FROM
  <s2.length  => lessThan, -- s1 is shorter than s2
  >s2.length  => greaterThan, -- s1 is longer than s2
  ENDCASE =⇒ equalTo; -- lengths are the same
  RETURN[c];
END;

lower: PACKED ARRAY CHARACTER[ 'A' .. 'Z' ] OF CHARACTER =
[ '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' ];

LowerCase: PROCEDURE [c: CHARACTER] RETURNS [CHARACTER] =
BEGIN RETURN [c IN ['A'..'Z'] THEN lower[c] ELSE c] END;

PrintLexicon: PUBLIC PROCEDURE =
BEGIN PrintNode[root] END;

PrintNode: PROCEDURE[n: NodePtr] =
BEGIN
  IF n = NIL THEN RETURN;
  PRINT Node[n.llink];
  IODefs.WriteLine[n.string];
  PrintNode[n.rlink];
  END;
END.
7.6.2. LexiconClient: a client module

The module, *LexiconClient*, below is a client for *Lexicon* and imports *LexiconDefs*. It is also a client for the interface *IODefs* (and also uses the constant *CR* defined in *IODefs* in section 7.1). The program provides a simple terminal interface to a user for testing *Lexicon*.

c1:  DIRECTORY
c2:   IOdefs: FROM "iodefs" USING [CR, ReadChar, ReadLine, WriteChar, WriteLine],
c3:   LexiconDefs: FROM "lexicondefs" USING [AddString, FindString, PrintLexicon];
c4:   c5:  LexiconClient: PROGRAM IMPORTS IOdefs, LexiconDefs =
c6:   c7:   BEGIN OPEN IOdefs, LexiconDefs;
c8:   s: STRING ← [80];
c9:   ch: CHARACTER;
c10:  DO -- loop until stopped by user typing q or Q (last case below).
c11:  c12:   WriteChar[CR]; WriteLine["Lexicon Command: "];
c13:   ch ← ReadChar[];
c14:   WriteChar[ch]; -- Echo the character (ReadChar doesn't).
c15:   SELECT ch FROM
c16:     'f, 'F =>
c17:       BEGIN
c18:         WriteLine["find: "]; -- terminal will read: "find: 

c19:         ReadLine[s]; -- s will contain the string read from the terminal

c20:         IF FindString[s] THEN WriteLine[" -- found"]
c21:         ELSE WriteLine[" -- not found"];
c22:       END;
c23:     'A, 'a =>
c24:       BEGIN
c25:         WriteLine["add: "];

c26:         ReadLine[s];

c27:         AddString[s];

c28:       END;
c29:     'P, 'p =>
c30:       BEGIN
c31:         WriteLine["print lexicon"]; -- terminal will read: "print lexicon"

c32:         WriteChar[CR]; PrintLexicon[];

c33:       END;
c34:     'Q, 'q =>
c35:       BEGIN

c36:         WriteLine["quit"]; WriteChar[CR]; -- terminal will read: "quit"

c37:         STOP;

c38:       END;

c39:       ENDCASE => WriteLine[" Commands are Find, Add, Print lexicon, and Quit"];

c40:     ENDLOOP;

c41:   c42:     END.

7.6.3. Binding, loading, and running a configuration: an overview

A *configuration description*, a "program" written in C/Mesa, describes how a set of Mesa modules are to be bound together to form a configuration. This binding is accomplished by "compiling" the configuration description (or, *configuration* for short) and results in a *binary configuration description* (a BCD).
The simplest (or atomic) BCD is the object module for a Mesa program module. Thus, the Mesa compiler produces the simplest BCDs, and the C/Mesa compiler (also called the Mesa Binder) produces complex BCDs from simpler ones. Indeed, a configuration may combine both atomic and non-atomic BCDs together into a single, new BCD. For these reasons, the object modules produced by the Mesa compiler have the same form of names as the output of the Binder, i.e., names of the form "BasicName.bcd".

Once a BCD has been created, it can be loaded and run.

Loading is a sequence of two actions. The first makes an instance of the configuration by allocating a frame for each atomic module in the BCD. Each frame has space for the module's static variables (those declared in the main body of the module) and some extra space for information used by the Mesa system. Imported procedures and variables are accessed via links. Space for these links is allocated either in the frame or in the code of the module.

The second part of loading completes the binding process by filling in the links for each module instance in the configuration instance. Some of these links will "point to" procedures and variables in the same configuration. Others will "point to" procedures and variables in the running system in which the configuration is being loaded.

Once a configuration is loaded, each module instance in it has all its interfaces bound. However, no code has been executed in the instances, so global variables are not initialized, and no mainline statements have executed. STARTing (sec. 7.8.2) an instance executes any code for initializing static variables and also executes its mainline code. For correct operation, this must occur before any of its procedures are used or before any of its global variables are referenced. If a module is not explicitly STARTed before one of its procedures is called, then a trap occurs, and it is automatically started. Once it STOPS (sec. 7.8.2), the procedure call is allowed to proceed. Subsequent procedure calls will not repeat this trap and auto-initialization sequence. Section 7.8 details how these mechanisms generalize for configurations.

### 7.6.4. A configuration description for running LexiconClient

The following configuration will bind Lexicon, LexiconClient, and other necessary modules and can be used to start the client program running. The comments to the right of each module name indicate which interfaces are imported and exported by that particular module; they are not part of Config1. This configuration is completely self-contained: all the needed imports are satisfied by interfaces exported from modules which are part of the configuration.

```
Config1: CONFIGURATION
  CONTROL LexiconClient =
BEGIN
  Fs;             -- IMPORTS SystemDefs
  IOPkg;          -- IMPORTS SystemDefs, IODefs
  Strings;        -- IMPORTS SystemDefs, IODefs, StringDefs
  Lexicon;        -- IMPORTS SystemDefs, IODefs, StringDefs
  LexiconClient;  -- IMPORTS IODefs, LexiconDefs
END;
```

To see that this configuration is completely self-contained, notice that LexiconClient imports IODefs, which is exported by IOPkg, and imports LexiconDefs, which is exported by the instance of Lexicon. Similarly, the other instances' import requirements are satisfied by some exported interface in Config1.
7.7 C/Mesa: syntax and semantics

The following is the complete syntax for C/Mesa. It bears strong resemblance to Mesa itself, but this grammar describes a completely separate language. A phrase class beginning with a C indicates a syntactic unit that is unique to C/Mesa. All the other units have the same syntax (but not necessarily exactly the same semantics) as they do in Mesa itself.

```
ConfigDescription ::= CDirectory -- optional
CPacking
Configuration. -- note the final period
CDirectory ::= -- same as in Mesa, only no USING clauses
Configuration ::= identifier: CHead =
CBody
CExports ::= empty | EXPORTS ItemList
CExpression ::= CPrimary | CExpression THEN CRightSide
CLeftSide ::= Item | [ItemList]
CBody ::= BEGIN CStatementSeries END
CHead ::= CONFIGURATION CLinks Imports CExports ControlClause
ControlClause ::= CONTROL identifier | empty
CLinks ::= empty | LINKS : CODE | LINKS : FRAME
CPacking ::= empty | CPackSeries ;
CPackList ::= PACK IdList
CPackSeries ::= CPackList | CPackSeries ; CPackList
CPrimary ::= CRightSide | CPrimary PLUS CRightSide
CRightSide ::= Item | Item [] CLinks | Item [IdList ] CLinks
CStatement ::= CLeftSide = CExpression |
CRightSide |
Configuration
CStatementSeries ::= CStatement |
CStatementSeries ; |
CStatementSeries ; CStatement
Imports ::= empty | IMPORTS ItemList
Item ::= identifier | identifier : identifier
ItemList ::= Item | ItemList , Item
```

We will use the term "component" to refer to the parts of a configuration; i.e., for both atomic modules and configurations containing several modules. When necessary, the kind of component will be expressly given.

Similarly, we will use the term "interface" to stand for an interface record or a module instance (if used in discussing imports or exports), and we will distinguish as necessary. However, "interface" will never include or imply the term "interface type" (sec. 7.4.1).

Lastly, we will need to distinguish between instances of components and their prototypes (the BCD files) from which such instances are made. Hence, a program prototype is the BCD file for a Mesa program module, and a configuration prototype is the analog for configurations. If the term prototype is used by itself, it includes both cases.

The CPacking and CLinks clauses in the syntax are directives to the Mesa Binder. CPacking identifies modules whose code should be packed together for swapping purposes. CLinks specifies
for a module or a configuration whether links to imported interfaces should be stored in the frame
or in the code. The use and implications of these optional clauses is described in Appendix D.

7.7.1. IMPORTS, EXPORTS, and DIRECTORY in C/Mesa

For completely self-contained, simple configurations like Config1, a configuration description is
primarily just a list of component names. An instance of each named component will be part of the
configuration, and if a component imports any interfaces, they will be supplied by those exported
from other components of the configuration.

Configurations need not be self-contained, however, and may themselves import interfaces to be
further imported by their components. In this way, subsystems can be constructed with some
imported interfaces unbound. Loading such a configuration or naming it as a component in another
configuration will supply the necessary interfaces. Furthermore, a configuration can make exported
interfaces available for importation by other modules and configurations. For example, the
interfaces SystemDefs, IODefs, and StringDefs needed by Config1 would normally be supplied by a
pre-existing Mesa system configuration. Therefore, it is really not necessary to include instances of
Fsp, IOPkg, and Strings in Config1. Instead, it can just import them:

```
c2.1: Config2: CONFIGURATION
c2.2: IMPORTS SystemDefs, IODefs, StringDefs
c2.3: CONTROL LexiconClient =
c2.4: BEGIN
 c2.5: Lexicon;
c2.6: LexiconClient;
c2.7: END
```

The imports clause in a configuration serves the same purpose as in a program module. The rule
for importing is: If some component named in a configuration imports SomeDefs, and SomeDefs
is not exported by a component in the configuration, then it must be imported. For example,
SystemDefs did not have to be imported into Config1, but it did have to be imported into Config2.

The rule for exports is simpler: If a component in a configuration exports an interface, that
interface may also be exported another level from the configuration. It is not required that it be
exported, however. This important feature enables one to control what is exported from a
configuration and what is to be hidden from external view.

None of the example configurations given so far have had a DIRECTORY section. This is because the
default association of a component named Prog is to a file named "Prog.bcd" in which the
ModuleName is also Prog. Since this is often the case, the programmer normally does not need to
supply one. A DIRECTORY part would be needed if the file did not have such a defaultable name.
For example:

```
DIRECTORY
 Prog: FROM "OldProgFile";
```

could not be omitted if the component named Prog is contained in the file "OldProgFile.bcd",
rather than in "Prog.bcd".

7.7.2 Explicit naming, IMPORTS, and EXPORTS *

In Mesa, names may be given to the interface records in an IMPORTS list (sec. 7.4.1); the same is true
in a configuration description. These names can then be used to supply the interfaces needed by
component instances in the configuration. The notation for explicitly supplying interfaces to a
component is similar to that for parameter lists in Mesa (except that there is no keyword notation for explicit imports parameter lists). For example, lines c2.1 through c2.5 above could have been written as:

   c2a.1: Config2A: CONFIGURATION
   c2a.2: IMPORTS alloc: SystemDefs, io: IODefs, str: StringDefs
   c2a.3: CONTROL LexiconClient =
   c2a.4: BEGIN
   c2a.5: Lexicon[alloc, io, str];

The interfaces listed after Lexicon must correspond in order and (interface) type with the IMPORTS list for Lexicon (look at Lexicon in sec. 7.6.1 to check this).

A name may also be given to each component instance in a configuration by preceding the instance with "identifier :". This facility is necessary to distinguish multiple instances of the same prototype from one another. For example, we could name the Lexicon instance in line c2a.5 as follows:

   alex: Lexicon[alloc, io, str];

Lexicon exports an interface whose type is LexiconDefs, and that interface record can also be named. The following further modification to line c2a.5 names it lexRec:

   lexRec: LexiconDefs + alex: Lexicon[alloc, io, str];

Here, as in Mesa, the type of lexRec follows the colon in the declaration, and lexRec is assigned the (single) interface exported by Lexicon. However, the type LexiconDefs is not actually necessary (it is inferred from Lexicon's EXPORTS list), and the line could have been shortened to

   lexRec + alex: Lexicon[alloc, io, str];

Using all these explicit naming capabilities, we can now write a new version of the configuration in which none of the C/Mesa default naming is used:

   c3.1: Config3: CONFIGURATION
   c3.2: IMPORTS alloc: SystemDefs, io: IODefs, str: StringDefs
   c3.3: CONTROL lexClient =
   c3.4: BEGIN
   c3.5: lexRec: LexiconDefs + alex: Lexicon[alloc, io, str];
   c3.6: lexClient: LexiconClient[io, lexRec];
   c3.7: END.

An exported interface like lexRec need not always be set as the result of including a component instance like alex in the configuration. One can also assign interface records to one another as in the following two (equivalent) lines:

   anotherLexRec: LexiconDefs + lexRec;
   anotherLexRec + lexRec;

The form of CRightSide in these two statements only copies lexRec, whereas ones like line c3.5 above involve a "call" on a component prototype. The result of that "call" is an instance of the component, and a set of results, the interface records exported by it.

7.7.3. Default names for interfaces and instances *

A component instance that is not explicitly given a name is given a default name equal to the name
of the component prototype. Thus, the body of Config2 is treated as if the programmer had written:

BEGIN
  Lexicon: Lexicon;
  LexiconClient: LexiconClient;
END.

Similarly, an unnamed interface is given a default name equal to the name of its interface type. So, another equivalent body for Config2 is

BEGIN
  LexiconDefs: LexiconDefs + Lexicon: Lexicon[
  LexiconClient: LexiconClient;
END.

The empty imports parameter list in "Lexicon[]" specifies that a new instance of the prototype Lexicon is to be created. If the empty imports list were not there, the binder would interpret the appearance of Lexicon (the one after the colon) as the name of an already existing interface (not of an already existing module instance). When no assignment is specified, the empty imports parameter list is not necessary, as shown in the earlier examples.

Normally, omitting an imports parameter list (or, equivalently, specifying an empty list) means that the binder should use the default-named interfaces needed by that component instance. Thus, we could rewrite a completely explicit (and very wordy, but equivalent) version of Config2:

```plaintext
  c2x.1: Config2X: CONFIGURATION
  c2x.2: IMPORTS SystemDefs: SystemDefs, IODefs: IODefs, StringDefs: StringDefs
  c2x.3: CONTROL LexiconClient =
  c2x.4: BEGIN
  c2x.5: LexiconDefs: LexiconDefs + Lexicon: Lexicon[SystemDefs, IODefs, StringDefs];
  c2x.6: LexiconClient: LexiconClient[IODefs, LexiconDefs];
  c2x.7: END.
```

Notice that the defaults greatly simplify a configuration, but that they also obscure a great deal of machinery concerned with naming things. It is important that the programmer not completely forget these details. Otherwise one could commit errors by not distinguishing between interface records and interface types, or between component instances and prototypes. For instance, this could be a problem if there are multiple component instances. Therefore, one is well advised to assign unique names to the instances.

### 7.7.4. Multiple exported interfaces from a single component

A component can export more than a single interface. Assigning these exported interfaces to interface records is done using a Mesa-like extractor (sec. 3.4.5). For example, if we had a program module StringsAndIO that exported both StringDefs and IODefs, we could use it in a modified Config2 as follows:

```plaintext
  c4.1: Config4: CONFIGURATION
  c4.2: IMPORTS alloc: SystemDefs
  c4.3: CONTROL LexiconClient =
  c4.4: BEGIN
  c4.5: [str: StringDefs, io: IODefs] + StringsAndIO[];
  c4.6: Lexicon[alloc, io, str];
  c4.7: LexiconClient[io, LexiconDefs];
  c4.8: END.
```

Line c4.5 assigns the exported interfaces obtained by instantiating StringsAndIO (that is why it has
an explicit, although empty imports parameter list following it) and declares their types as well. It
would be equally correct to write instead

\[
[str, io] \leftarrow StringsAndIO[];
\]

In this case the types for \textit{io} and \textit{str} would be inferred from the types of the interface records
exported by \textit{StringsAndIO}. However, if the programmer had written instead,

\[
[io, str] \leftarrow StringsAndIO[];
\]

with the positions of \textit{io} and \textit{str} reversed, that would have been accepted, but would have caused
errors in both lines c4.6 and c4.7 because their inferred types would not match those explicit imports
parameter lists. Be cautious when doing this.

Default names could also have been used for the exported interfaces in line c4.5, and \textit{Config4} could
simply have been written as

\[
c4a.1: \hspace{1em} Config4A: CONFIGURATION

c4a.2: \hspace{1em} \textbf{IMPORTS} SystemDefs

c4a.3: \hspace{1em} \textbf{CONTROL} LexiconClient =

c4a.4: \hspace{1em} \textbf{BEGIN}

c4a.5: \hspace{1em} StringsAndIO;

c4a.6: \hspace{1em} Lexicon;

c4a.7: \hspace{1em} LexiconClient;

c4a.8: \hspace{1em} \textbf{END}.
\]

This would assign the exported interfaces to the default-named records \textit{StringDefs} and \textit{IODefs} and
would use them in the defaulted import parameter lists for \textit{Lexicon} and \textit{LexiconClient}. Line c4a.5
could also show what \textit{StringsAndIO} exports using the default names for its exported records. This
would give rise to the statement:

\[
[StringDefs, IODefs] \leftarrow StringsAndIO[];
\]

Cases like this require that the user be aware of the distinction between interface records and
interface types: \textit{StringDefs} names an interface record here, but in line c4.5, it names an interface
type.

7.7.5. Multiple components implementing a single interface *

An exported interface can be the result of contributions by a number of components. Think of the
interface as a logical unit that may be implemented by a number of cooperating physical units (i.e.,
modules and configurations). For example, assume that \textit{Lexicon} is divided into two modules
\textit{LexiconFA} and \textit{LexiconP}, with \textit{LexiconFA} providing the procedures \textit{FindString} and \textit{AddString}, and
\textit{LexiconP} providing \textit{PrintLexicon}. Each exports \textit{LexiconDefs}, but neither fully implements that
interface. Still, \textit{LexiconClient} will see a single interface in the following:

\[
c5.1: \hspace{1em} Config5: CONFIGURATION

c5.2: \hspace{1em} \textbf{IMPORTS} SystemDefs, IODefs, StringDefs

c5.3: \hspace{1em} \textbf{CONTROL} LexiconClient =

c5.4: \hspace{1em} \textbf{BEGIN}

c5.5: \hspace{1em} lexRec: LexiconDefs \leftarrow LexiconFA[]; \hspace{1em} \text{\textendash\ use default imports}

c5.6: \hspace{1em} lexRec \leftarrow LexiconP[]; \hspace{1em} \text{\textendash\ merge interface contributions}

c5.7: \hspace{1em} LexiconClient[IODefs, lexRec];

c5.8: \hspace{1em} \textbf{END}.
\]

The two separate assignments to \textit{lexRec} above actually merge the interface elements exported by the
two modules. This merging does not allow any duplication of elements, and if both modules
exported \textit{PrintLexicon}, for example, an error would be generated during processing of \textit{Config5} by
the Binder.

The user may control the merging of interfaces himself using the PLUS operator. To obtain the same effect as above (but by explicit specification), one could write

\[
\begin{align*}
\text{lexRecFA} & \leftarrow \text{LexiconFA}[]; \\
\text{lexRecP} & \leftarrow \text{LexiconP}[]; \\
\text{lexRec} & \leftarrow \text{lexRecFA} \text{PLUS} \text{lexRecP}; \\
\text{LexiconClient}[\text{IODefs}; \text{lexRec}]; & \quad \text{-- one part} \\
\text{lexRecP} & \leftarrow \text{LexiconP}[]; \\
\text{lexRec} & \leftarrow \text{lexRecFA} \text{PLUS} \text{lexRecP}; \\
\text{LexiconClient}[\text{IODefs}; \text{lexRec}]; & \quad \text{-- the merge} \\
\text{lexRec} & \leftarrow \text{lexRecFA} \text{PLUS} \text{lexRecP}; \\
\text{LexiconClient}[\text{IODefs}; \text{lexRec}]; & \quad \text{-- same as line c5.7}
\end{align*}
\]

If the programmer wanted to use the original Lexicon, but use LexiconP's PrintLexicon in the interface instead of Lexicon's, he could use the THEN operator:

\[
\begin{align*}
\text{lexRec} & \leftarrow \text{Lexicon}[]; \\
\text{lexRecP} & \leftarrow \text{LexiconP}[]; \\
\text{lexRecNew} & \leftarrow \text{lexRecP} \text{THEN} \text{lexRec}; \\
\text{LexiconClient}[\text{IODefs}; \text{lexRecNew}]; & \quad \text{-- defines a complete interface} \\
\text{lexRecP} & \leftarrow \text{LexiconP}[]; \\
\text{lexRec} & \leftarrow \text{lexRecFA} \text{PLUS} \text{lexRecP}; \\
\text{LexiconClient}[\text{IODefs}; \text{lexRecNew}]; & \quad \text{-- defines one procedure} \\
\text{lexRec} & \leftarrow \text{lexRecFA} \text{PLUS} \text{lexRecP}; \\
\text{LexiconClient}[\text{IODefs}; \text{lexRecNew}]; & \quad \text{-- this order is important}
\end{align*}
\]

The THEN operator makes an interface that includes all the elements defined by lexRecP (the left operand) together with those from lexRec (the right operand) that do not duplicate any in lexRecP. This could be useful if one simply wanted to test a new version of PrintLexicon procedure without altering Lexicon itself during the debugging period. Also, one could use THEN to provide a number of alternative PrintLexicon procedures, with the standard one incorporated in Lexicon.

7.7.6. Nested (local) configurations

Configurations may be defined within configurations, much like local procedures (sec. 5.7) may be defined within other procedures in Mesa. They can then be instantiated and parametrized, and they can export interfaces (just like any configuration).

Nested configurations can be used to hide some of the interfaces exported by components in a configuration. For example, suppose that multiple instances of some component ProgMod were needed in a configuration, and further suppose that ProgMod exports the interface ProgDefs. Even if none of the exported ProgDefs interface records are needed in the configuration, they would each have to be given a unique name to avoid an interface merging error (sec. 7.7.5).

This could be avoided by defining the following nested configuration:

\[
\text{NonexportingPM: CONFIGURATION} = \begin{align*}
\text{BEGIN} & \text{ ProgMod } \text{ END.}
\end{align*}
\]

Using NonexportingPM in place of ProgMod avoids the duplicate interface problem because the local configuration does not export the interface ProgDefs produced by instantiating ProgMod within it.

Nested configurations can also be used to avoid writing sequences of C/Mesa statements more than once. By collecting such a sequence in a nested configuration, one can get the effect of writing the whole sequence simply by instantiating the configuration.

The scope rules for names in C/Mesa allows a nested configuration to access interfaces and other (also nested) configurations outside it. So, one configuration can make instances of others. However, in its IMPORTS list, a nested configuration must name any interfaces that its components import but which are not satisfied within it. That is, interfaces are never automatically imported into a nested configuration.
7.8. Loading modules and configurations

This section describes how configurations are loaded and run. Simple, atomic modules are discussed first, and then more general configurations.

Loading and running an atomic module is a sequence of four actions:

1. loading its object code (from the .BCD file),
2. allocating a frame for its static variables,
3. filling in procedure descriptors for imported procedures and frame pointers for imported modules,
4. initializing the module’s variables and executing its mainline code.

Actions (1), (2), and (3) are accomplished by system procedure, documented in the System Documentation. Action (4) can be accomplished by explicitly starting the instance or by means of a trap on the first call to any of its procedures (both of these methods are described below).

7.8.1. The NEW operation for making copies of modules

The syntax for the NEW operation is

\[
\text{Expression} ::= \ldots | \text{NEW Variable}
\]

The Variable may be the name of an imported frame pointer, a pointer to the frame of a program module, a program variable, or the module name of the module containing the NEW statement. For example:

\[
\text{progInst1}, \text{progInst2}: \text{POINTER TO FRAME[Prog]};
\]

\[
\text{progInst2} \leftarrow \text{NEW progInst1};
\]

The new instance is only a copy of the frame insofar as its interface records are concerned. In all other respects it is uninitialized, just like a new instance. In particular, it must be started to supply its program parameters (if any) and to initialize its global variables. At the time the copy is made, it will have exactly the same bindings as the original. If some of the globally available interface records maintained by the loader (sec. 7.8.2) later change, the copy may be bound differently than the original.

If a module imports a program Pimpl (sec. 7.4.2), the operation "NEW Pimpl" copies Pimpl.

A program module’s type may be declared in a definitions module in the same way as a procedure’s type is. Such a defined program is part of the interface defined by that definitions module and may, therefore, be imported by another module as part of that interface. Then, copies of that module can be made using the NEW operation. For example, assume that the following declaration appears in the definitions module, Defs:

\[
\text{ExportedProg: PROGRAM [i: INTEGER];}
\]

Any program that imports Defs will then have access to a value named ExportedProg which will have been bound (in step (3) of the loading process) to an instance of a program whose parameter types conform with those of ExportedProg. The only operation that a program can perform using this value is to START it, RESTART it, or make a copy of it using "NEW ExportedProg". In summary, a program imported as part of an interface behaves like a value that is a pointer to a frame.
If a program, say *Prog*, wishes to create a copy of itself, it can say:

```plaintext
copy: POINTER TO FRAME[Prog];
...
copy ← NEW Prog;
```

7.8.2. *How the loader binds interfaces*

Each instance of an atomic module or of a configuration may export some interfaces. To make these exported interfaces available for importation by other instances, the loader maintains a single, simple global table of all the exported interfaces. If any duplicates are created as the result of a `NEW`, they are merged into the already existing interface records as if a `THEN` (sec. 7.7.5) had been done.

The moral here is that complicated binding to hide interfaces, etc. must be done using the binder, and only the simplest, most straightforward forms should be used at loading time.

7.8.3. **STARTing, STOPping, and RESTARTing module instances**

The `START` operation suspends the execution of the program or procedure executing it and transfers control to a new, uninitialized instance of an atomic module. Additionally, if the program instance being started requires parameters, they are supplied as part of the `START`. Similarly, if the program being started is specified to return results (more details below), then the `START` operation may appear in a `RightSide` context, and the returned value is the value of the operation. Its syntax is

```plaintext
StartStmt ::= START Call|...
StartExpr ::= START Call|...
```

The variable following the word `START` must represent a global frame pointer or program variable; i.e., its type must conform to some `POINTER TO FRAME` type or `PROGRAM` type. Here are some examples of its use:

```plaintext
START progInst;
START ExportedProg[5+j];
x ← START progWithResult[firstArg: a, secondArg: b]; --keyword parameter list
```

When a program is started, it first executes code to initialize any static variables that were declared with initialization expressions. The initializations are done in the order in which the variables were declared in the program. Also, they may call both local and imported procedures (since descriptors for all imported procedures are filled in as part of the `NEW` operation - sec. 7.7.1).

After all initialization expressions are complete, the mainline statements of the program commence executing. Control can then return to the caller (the program or procedure which initiated the `START`) in one of two ways: the started program may `STOP` or it may `RETURN` with results (however, it cannot use both).

A program that executes a `STOP` can be `RESTART`ed later. `RESTART` is distinct from `START` primarily because it cannot pass parameters as `START` can. If a program does not return results, it either by an explicit use of `STOP` or by running off the end of the program.

If a program declares (in its `ModuleHeader`) that it returns results, it uses `RETURN` statements just as does a procedure (and it cannot use `STOP`). A `RETURN` from a program does *not* deallocate its global frame. The syntax for `RESTART` and `STOP` is
The Variable following RESTART must be a pointer to the frame for a program instance or a program variable, just as for START. A program that RETURNS results or has run off the end cannot be RESTARTed. Attempting to do so will result in a run time error.

A module instance can also be STARTed "automatically". If a call is made on a procedure in an instance that has not yet been started, a start trap occurs. If the module does not take parameters when started, then it is started by the Mesa start-trap handler. When it STOPS or RETURNS, the trap handler completes the procedure call that was in progress when the trap occurred. (See the next section for further discussion of the start trap for configurations.)

Warning: A module must be STARTed either explicitly or implicitly before any attempt is made to access its variables through a POINTER TO FRAME.

7.8.4. Loading and starting configurations

By using system routines, one can also make instances of configurations that are more than simple, atomic modules. A non-atomic configuration cannot be STARTed (what would it mean to start one?), but its CONTROL module can (if it has one). Basically, the CONTROL module acts as the representative for the whole configuration (since a C/Mesa configuration description does not contain executable Mesa statements). Thus, a program that STARTS the CONTROL module for a configuration has essentially STARTed the configuration. If the order of starting some of the instances in a configuration is important or if they take arguments when started, its CONTROL module should START them explicitly.

The start trap works for configurations as well as for atomic modules. If a start trap occurs for a module \( M \) in configuration \( C \) with control module \( cM \), then the trap handler automatically starts \( cM \) rather than \( M \). If the handler discovers, however, that \( cM \) has already been started, it will start \( M \) (since \( cM \) would have started \( M \) if it had intended to). In fact, if the handler starts \( cM \) but still finds \( M \) unstarted when \( cM \) STOPS, it will start \( M \) itself before finally returning from the trap. Then the procedure call that caused the trap will be allowed to go through.

Fine points:

If an attempt is made to RESTART a program which has not been started, a START trap will occur and then the RESTART will proceed.

Other forms of START and STOP statements are used to catch signals. This is discussed in Chapter 8, but the forms look roughly as follows:

\[
\begin{align*}
\text{START} & \ someInstance [ \ ComponentList ! \ CatchPhrase ] \\
\text{STOP} & \ [ ! \ CatchPhrase ]
\end{align*}
\]
CHAPTER 8.

SIGNALLING AND SIGNAL DATA TYPES

Signals are used to indicate when exceptional conditions arise in the course of execution, and they provide an orderly means of dealing with those conditions, at low cost if none are generated (and they almost never are). For example, it is common in most languages to write a storage allocator so that, if asked for a block whose size is too large, it returns a null (or otherwise invalid) pointer value. Any program which calls the allocator then embeds the call in an \texttt{if} statement, and checks the return value to make sure that the request was satisfied. What that procedure then does is a very local decision.

In Mesa, one would write the allocator as if it \texttt{always} returned a valid pointer to an allocated block, and calls to it would simply assign the returned value to a suitable pointer, without checking whether or not the allocation worked. If the caller needs to gain control when the allocator fails, the programmer attaches a \texttt{CatchPhrase} to the call; then if the allocator generates the signal \texttt{BlockTooLarge}, and the caller has indicated that it wants to catch that signal, it will.

This way of handling exceptions has two important properties, one for the human reader of the program, and one for its execution efficiency:

Anyone reading a program with a call on the allocator can see immediately that an exceptional condition can arise (by the catch phrase on the call or nearby); he then knows that this is an unusual event and can read on with the normal program flow: \texttt{if} statements do not have this characteristic of distinguishing one branch from the other.

When the program is executing, the code to check the value returned by the allocator on every call is not present and therefore takes no space or execution time. Instead, if a signal is generated, there is more overhead to get to the catch phrase than a simple transfer; but since it happens infrequently, the overall efficiency is much higher than checking each call with an \texttt{if} statement.

Signals work over many levels of procedure call, and it is possible for a signal to be generated by one procedure and be handled by another procedure much higher up in the call chain. We later discuss the mechanisms by which this is done; until then, examples show signals being caught by the caller of the procedure which generated the signal.

8.1. Declaring and generating SIGNALS and ERRORS

In its simplest form, a signal is just a name for some exceptional condition. Often, parameters are passed along with the signal to help a catch phrase which handles it in determining what went wrong. It is also possible to recover from a signal and allow the routine which generated it to continue on its merry way. This is done by a catch phrase returning a result; the program which generated the signal receives this result as if it had called a normal procedure instead of a signal.
Therefore, from the type viewpoint, signals correspond very closely to procedures; in fact, the type constructor for declaring signals is just a variation of the one for procedures:

\[
\text{SignalTC} ::= \text{SignalOrError ParameterList} \text{RETURNS ResultList} \\
\text{SignalOrError ParameterList} \\
\text{SignalOrError} \text{RETURNS ResultList} \\
\text{SignalOrError}
\]

\[
\text{SignalOrError} ::= \text{SIGNAL} | \text{ERROR}
\]

For example, the signal \textit{BlockTooLarge} might be defined to carry along with it two parameters, a \textit{Zone} within which the allocator was trying to get a block, and the number of words needed to fill the current request. The catch phrase that handles the signal is expected to send back (i.e., return) an array descriptor for a block of storage to be added to the zone. The declaration of \textit{BlockTooLarge} would look like:

\[
\text{BlockTooLarge: SIGNAL} [z: \text{Zone}, \text{needed: CARDINAL}] \\
\text{RETURNS [newStorage: DESCRIPTOR FOR ARRAY OF CARDINAL];}
\]

A signal variable contains a unique name at run time, which is a code identifying an \textit{actual signal}, just as a procedure variable must be assigned an \textit{actual procedure} before it can be used. If a procedure is imported from an interface (sec. 7.4), any signals that it generates directly are probably contained in the same interface. Imported signals are bound by the same mechanisms as procedures. In addition, one may have signal variables which can be assigned any signal value of a compatible type.

The signal analog of an actual procedure is obtained by initializing a signal variable using the syntax "\texttt{=} \text{CODE}" in place of "\texttt{=} \texttt{BEGIN...END}" for procedures. This causes the signal to be initialized to contain a unique value. The following syntax describes the initialization for an actual signal:

\[
\text{Initialization} ::= \text{=} \text{CODE} ...
\]

A signal is generated by using it in a \texttt{SignalCall} as shown in the syntax below:

\[
\text{Statement} ::= \text{SignalCall} ...
\]

\[
\text{SignalCall} ::= \text{SIGNAL Call | ErrorCall}
\]

\[
\text{ErrorCall} ::= \text{RETURN WITH ERROR Call} \\
\text{ERROR Call} \\
... \text{special error}
\]

\text{Call} is defined in section 5.4, and the called \texttt{Expression} must have some signal type in this case. A \texttt{SignalCall} can be used as an \texttt{Expression} as well as a \texttt{Statement}. For example,

\[
\text{newblock } \leftarrow \text{SIGNAL } \textit{BlockTooLarge}[\textit{zone, n}];
\]

Thus, generating a signal or error looks just like a procedure call, except for the additional word, \texttt{ERROR} or \texttt{SIGNAL}.

\text{Fine point:}

Although it is not recommended, the keywords \texttt{SIGNAL} and \texttt{ERROR} may be omitted (except in the \texttt{RETURN WITH ERROR} construct). This makes the signal look exactly like a procedure call.

Initialization by \texttt{SIGNAL = CODE} produces a unique value that contains, in part, the global frame index of the module containing the initialization. There are two points worth making. If one creates a copy of the module with the \texttt{NEW} statement, signals raised by the two copies will be different. If the signal is declared and initialized in a procedure, recursive calls of the procedure will not generate different signal values.
If a signal is declared as an ERROR, it must be generated by an ErrorCall. If, however, it is declared as a SIGNAL, it can be generated by any SignalCall, including an ErrorCall. The difference between the two is that a catch phrase may not RESUME a signal generated by an ErrorCall (sec. 8.2.5).

Except for a slight difference in the way the error is started (sec. 8.2.3), the RETURN WITH ERROR construct behaves like the ERROR statement. Its primary use is in monitor ENTRY procedures (chapter 10).

The "special error" in the above syntax is used to indicate that something has gone wrong, without giving any indication of the cause; the statement

ERROR;

generates a system-defined error. It is provided to cover those "impossible" cases which should never occur in correct programs but which it is always best to check for (such as falling out of a loop that should never terminate normally, or arriving at the ENDCASE of a SELECT statement that claims to handle all the cases). It can only be caught using the ANY option in a catch phrase (sec. 8.2.3). It is customarily handled by the debugger.

8.1.1 ERROR in expressions

When an ERROR is declared to return values, this is purely for syntactic convenience, since one of the principal features of an ERROR is that it does not "return". The reason for doing this is to allow the ERROR to stand in an expression context considered invalid or impossible. Such declarations of returned values are not necessary; if an expression has an ERROR type (or SIGNAL type raised as an error) and returns no value, then that expression can be used wherever an expression of any type is required. For example:

```vcl
Color: TYPE = {red, orange, yellow, green, blue, violet};
c: Color;
button: [0..2);

button ← SELECT c FROM
  red => 0,
  yellow => 1,
  blue => 2,
ENDCASE => ERROR;
```

In the example, the only valid colors for buttons are red, yellow, and blue. Any other value results in an error (in this case, the unnamed system error). Such constructs allow an inexpensive way to get to the debugger in those "impossible" cases that arise from program errors.

A fine point:

If the ERROR type is defined to return a value, any use of that expression must be type correct with respect to the "returned" value.

8.2. Control of generated signals

Any program which needs to handle signals must anticipate that need by providing catch phrases for the various signals that might be generated. During execution, certain of these catch phrases will be enabled at different times to handle signals. Loosely speaking, when a signal S is generated, the
procedures in the call hierarchy at that time will be given a chance to catch the signal, in a last-in-first-out order. Each such procedure \( P \), if it has an enabled catch phrase, is given the signal \( S \) in turn, until one of them stops the signal from propagating any further (by mechanisms which are explained below). \( P \) may decide to reject \( S \) (in which case the next procedure in the call hierarchy will be considered), or \( P \) may decide to handle \( S \) by taking control and attempting to recover from the signal.

8.2.1. Preparing to catch signals: catch phrases

A catch phrase has the following form:

\[
\begin{align*}
\text{CatchTail} &::= \text{Catch} \\
&\quad \text{ANY} \Rightarrow \text{Statement} \\
\text{Catch} &::= \text{ExpressionList} \Rightarrow \text{Statement} \\
\text{CatchSeries} &::= \text{CatchTail} | \\
&\quad \text{Catch} ; \text{CatchSeries}
\end{align*}
\]

The expressions in the ExpressionList (semantically restricted to a list of variables) must evaluate to the names of signals (unless otherwise stated, we use \textit{signal} to stand for both \textit{ERROR} and \textit{SIGNAL}). The special identifier \textit{ANY} will match any signal (sec. 8.2.3). Note that if \textit{ANY} occurs, it must be last.

A catch phrase is written as part of an argument list, just after the last argument and before the right bracket. Catch phrases may appear in a procedure call, \textit{SignalCall}, \textit{NEW}, \textit{START}, \textit{RESTART}, \textit{STOP}, \textit{JOIN}, \textit{FORK}, or \textit{WAIT} (but not in a \textit{RESUME} or \textit{RETURN}). A catch phrase may also be appended to the \texttt{BEGIN} of a block or the \texttt{DO} of a loop statement by means of an \texttt{EnableClause}. The applicable syntax for a call and for a block or loop statement is

\[
\begin{align*}
\text{Call} &::= \text{Variable} [ \text{ComponentList} ! \text{CatchSeries} ] | \\
&\quad \text{Variable} [ ! \text{CatchSeries} ] \\
&\quad \ldots \\
\text{Block} &::= \text{BEGIN} \quad \text{-- (from Section 4.4)} \\
&\quad \text{OpenClause} \\
&\quad \text{EnableClause} \\
&\quad \text{DeclarationSeries} \\
&\quad \text{StatementSeries} \\
&\quad \text{ExitsClause} \\
&\quad \text{END} \\
\text{EnableClause} &::= \text{empty} | \\
&\quad \text{ENABLE \ CatchItem} ; | \\
&\quad \text{ENABLE BEGIN \ CatchSeries END} ; | \\
&\quad \text{ENABLE BEGIN \ CatchSeries ; END} ;
\end{align*}
\]

Note that the \texttt{EnableClause} is always followed by a semi-colon, and \texttt{BEGIN...END} must be used if there is more than one \texttt{Catch} in an \texttt{EnableClause}.

The main difference between the two kinds of catch phrases (\texttt{ENABLE} and \texttt{!}) is the scope of their influence. A catch phrase on a \texttt{Call} is only enabled during that call. A catch phrase at the beginning of a compound or loop statement is enabled as long as control is in that block; it can catch a signal resulting from \texttt{any call} in the block (or generated in the block).

To clarify the scope of influence of \texttt{ENABLE} clauses, the following two diagrams are reproduced from chapter 4. The scope of each phrase extends over others with greater indentation.
8.2.2. The scope of variables in catch phrases

Catch phrases are called to handle signals (the exact mechanisms are discussed in the next section). The naming environment that exists when a catch phrase is called (in order of innermost to outermost scope) includes any parameters passed with that signal (these are declared as part of a signal's definition), and any variables to which the procedure or program activation containing the catch phrase has access.

If a Catch has more than one label (or the label ANY), where the types of those labels are not identical, then the signal’s arguments are not accessible in the Statement chosen by that Catch.

If, however, there is exactly one type for the signals named in a Catch's ExpressionList, then the signal’s arguments are accessible in the statement following "=>". The names used are the parameters given in the signal’s declaration, just as for procedures. For example, a catch phrase for signal BlockTooLarge (defined earlier) might be used in a section of code such as:

```plaintext
-- in StorageDefs
BlockTooLarge: SIGNAL [z: Zone, needed: CARDINAL]
    RETURNS[newStorage: DESCRIPTOR FOR ARRAY OF CARDINAL];
GetMoreStorage: PROCEDURE [z: Zone, n: CARDINAL]
    RETURNS [DESCRIPTOR FOR ARRAY OF CARDINAL];
-- in a user program
p: POINTER TO Account;
...
p <= Allocate[SIZE[Account]]!
    BlockTooLarge => RESUME[GetMoreStorage[z, needed]];
```

The names z and needed in the catch phrase refer to the parameters passed along with the signal from Allocate (see sec. 8.2.5 for a discussion of RESUME).
8.2.3. Catching signals

When a signal is generated, what really happens is that the signal code, and a descriptor for the actual arguments of the signal, are passed to a Mesa run-time procedure named \textit{Signaller}. \textit{Signaller}'s definition is

\begin{verbatim}
Signaller: PROCEDURE[s: SignalCode, m: Message];
\end{verbatim}

Here \(s\) identifies the signal being generated, and \(m\) contains its arguments. (Actually, different procedures are used to distinguish between \textit{SIGNAL}, \textit{ERROR}, and \textit{RETURN WITH ERROR}.)

\textit{Signaller} proceeds to pass the signal and its argument record from one enabled catch phrase to the next in an orderly fashion. The order, at the procedure level, follows the current call hierarchy, from the most recently called procedure to least recently called, \textit{beginning with the procedure that generated the signal itself}. If the caller of a procedure is the outermost block of code for a program, the \textit{Signaller} will follow its \textit{return link} to continue propagating the signal (the return link points to the frame that last \textit{STARTed} the module (sec. 7.8)).

If, in place of \textit{SIGNAL} or \textit{ERROR}, a \textit{RETURN WITH ERROR} is used, the procedure that generated the error is first deleted (after releasing the monitor lock, if it is an \textit{ENTRY} procedure), and propagation of the error begins with its caller.

As \textit{Signaller} considers each frame, it looks to see whether that frame has any enabled catch phrases; if so, \textit{Signaller calls} the innermost catch phrase as if it were a procedure, passing it the \textit{SignalCode} and \textit{Message}. The innermost catch phrase is defined to be

- either the one after "!" attached to the currently incomplete procedure call for that frame, or
- the one following an \textit{ENABLE} in the innermost enclosing block that contains that call.

Because signals can be propagated right through the call hierarchy, the programmer must consider catching not only signals generated \textit{directly} within any procedure that is called, but also any generated \textit{indirectly} as a result of calling that procedure. Indirect signals are those generated by procedures called from within a procedure that you call, unless they are stopped before reaching you.

When a catch phrase is called, it behaves like a \textit{SELECT} statement: it compares the signal code passed to it with each signal value in the \textit{ExpressionList} of each \	extit{Catch} in the catch phrase. If the signal code matches one of the signal values, control enters the statement following the "=>" for that \textit{Catch}; if not, the next \textit{Catch} is tried. A \textit{Catch} consisting of "ANY => Statement" automatically matches any signal code (and is the only way to catch the unnamed \textit{ERROR} generated by the standalone \textit{ERROR} statement discussed in section 8.1).

\textbf{Fine point:}

The \textit{ANY} catchall is intended primarily for use by the debugger, and should generally be avoided. It matches any signal, including \textit{UNWIND} and all system-defined signals that might indicate some catastrophic condition (a double memory parity error, for example).

When a match is found, that \textit{Catch} is said to have \textit{caught} or \textit{accepted} the signal. If no alternative in a catch phrase accepts the signal, there may be another enabled catch phrase in some surrounding block. If so, the first catch phrase sends control to the second one so that it can inspect the signal, and so on until the last enabled catch phrase in that routine has had a chance at the signal. If no catch phrase in the routine accepts the signal, control returns to \textit{Signaller} with a value indicating that the signal was rejected, and \textit{Signaller} propagates the signal to the next level in the call hierarchy. In fact, all catch phrases are called by \textit{Signaller} as if they were procedures of the following type:
Chapter 8: Signalling and SIGNAL Data Types

 CatchPhrase: PROCEDURE[s: SignalCode, m: Message]  
                  RETURNS{{Reject, Unwind, Resume}};

The SELECT-like statement associated with each Catch has an implicit Reject return as its ENDCASE; hence, if control simply falls out of the statement, the signal is rejected.

Fine point:

If the same signal, _foo_, is enabled in several nested catch phrases in a procedure, each is given a chance to handle _foo_ if the inner ones reject the signal.

_Signaller_ continues propagating the signal up the call chain until it is exhausted, i.e., until the root of the process has considered and rejected the signal. At that point, an uncaught signal has been generated, and drastic action must be taken.

_Mesa guarantees that all signals will ultimately be caught and reported by the Debugger to the user. This is helpful in debugging because all the control context which existed when the signal was generated is still around and can be inspected to investigate the problem._

The declaration of CatchPhrase above indicates three reasons for returning to Signaller. The first, Reject, has already been discussed. The third, Resume, is discussed in section 8.2.5.

The second reason, Unwind, is used when a catch phrase has accepted a signal and is about to do some form of unconditional jump into the body of the routine containing it (this is the only form of "non-local goto" in _Mesa_). The jump may be generated by a GOTO statement (sec. 4.4), an EXIT or LOOP (sec. 4.5), or a RETRY or CONTINUE (see below). As immediately preceding such a jump, the catch phrase returns to _Signaller_ with result Unwind; it also indicates the frame containing the catch phrase and the location for the jump. This causes _Signaller_ to perform the following sequence of actions:

1 (1) Beginning at the frame in which the original signal was generated (or its caller, if a RETURN WITH ERROR was executed), it passes the signal UNWIND to each frame. This signal tells that activation that it is about to be destroyed and gives it a chance to clean up before dying. _Signaller_ then deallocates the frame and follows the same path as it did for the original signal to continue unwinding control. When it comes to the frame containing the catch phrase, it stops.

2. _Signaller_ then arranges for the jump to take place, and simply does a return to that frame, destroying itself in the process.

Every _Mesa_ program contains the pre-declared value

UNWIND: ERROR = CODE;

Fine points:

One cannot say RETURN in a catch phrase to return from the enclosing procedure. This is an implementation restriction that may be removed in the future, caused by the way in which a catch phrase is "called" like a procedure itself.

The UNWIND sequence gives each activation that is to lose control a chance to make consistent any data structures for which it is responsible. There are no constraints on the kinds of statements that it can use to do this: procedure calls, loops, or whatever are all legal. If, however, a catch for the UNWIND signal, such as,

START NextPhase [ !UNWIND => GOTO BailOut];

decides itself to perform a control transfer that would also initiate an UNWIND, this will override the original UNWIND, and _Signaller_ will stop right there, as if the second UNWIND catch had been the originator of the UNWIND.
8.2.4. **RETRY and CONTINUE in catch phrases**

Besides GOTO, EXIT, and LOOP, there are two other statements, RETRY and CONTINUE, which initiate an UNWIND. These can only be used within catch phrases.

RETRY means "go back to the beginning of the statement to which this catch phrase belongs"; CONTINUE means "go to the statement following the one to which this catch phrase belongs" (what is called *Next-Statement* in chapter 4).

For a catch phrase in a Call, the catch phrase "belongs" to the statement containing that Call. Thus, if the signal NoAnswer is generated for the call below, the assignment statement is retried:

```plaintext
answer ← GetReply[Send["What next?"] ! NoAnswer => RETRY];
```

On the other hand, if CONTINUE had been used instead, the statement after the assignment would be executed next (and the assignment would not be performed). For example, suppose the procedure ReadLine reads characters from a file up to a carriage return and appends them onto the string buffer. If reading beyond the end of file raises the signal StreamError, the call

```plaintext
ReadLine[ ! StreamError => IF buffer.length > 0 THEN CONTINUE];
```

deals with the case of no carriage return after that last line in the file. If there is no such final line, other catch phrases higher on the call chain are given a chance to catch the signal.

For a catch phrase after ENABLE, there are two cases to consider, blocks and loops. In a block, the catch phrase "belongs" to that statement; the next section shows an example. In a loop, the catch phrase "belongs" to the body of the loop, and CONTINUE really means "go around the loop again."

The following two examples are equivalent:

```plaintext
UNTIL p = NIL
  DO ENABLE TryList2 => BEGIN p ← list2; CONTINUE END;
  ...
ENDLOOP;

UNTIL p = NIL
  DO
    BEGIN ENABLE TryList2 => BEGIN p ← list2; CONTINUE; END;
    ...
  END;
ENDLOOP;
```

In any case, recall that an Unwind is initiated prior to completion of a RETRY or CONTINUE.

If a procedure call in the Initialization clause of a declaration contains a catch phrase, this catch phrase cannot contain RETRY or CONTINUE since it is in no well defined statement.

8.2.5. **Resuming from a catch phrase: RESUME**

The third alternative available to a catch phrase, after Reject and Unwind, is Resume. This option is invoked by using the RESUME statement to return values (or perhaps just control) from a catch phrase to the routine which generated the signal. To that routine, it appears as if the signal call were a procedure call that returns some results. The syntax for RESUME is just like that for RETURN:

```plaintext
Statement ::= ResumeStmt | RETRY | CONTINUE |
ResumeStmt ::= RESUME | RESUME [ComponentList]
```
When Signaller receives a Resume from a catch phrase, it simply returns and passes the accompanying results to the routine that originally called it (i.e., that generated the signal). If the signal was generated by an ErrorCall and a catch phrase requests a Resume, Signaller simply generates a signal itself (which results in a recursive call on Signaller); its declaration is

\[ \text{ResumeError: PUBLIC ERROR;} \]

Since it is an ERROR, one cannot legally Resume it.

The ability to Resume and return values gives the ability to deal with exceptional conditions in a way that is quite inexpensive in the non-exceptional case. For example, consider the declaration

\[ \text{StringBoundsFault: SIGNAL [s: STRING] RETURNS [ns: STRING];} \]

This signal allows the user to deal with the situation where characters are to be added to a string that is already "full". Thus the call

\[ \text{AppendChar[str, c ! StringBoundsFault =>} \]
\[ \text{BEGIN} \]
\[ \text{ns := AllocateString[s.maxlength+10];} \]
\[ \text{AppendString[ns, s];} \]
\[ \text{FreeString[s];} \]
\[ \text{RESUME[str => ns];} \]
\[ \text{END;} \]

allocates a larger string and updates the local variable whenever the string is about to overflow. Of course, the procedure AppendChar has to be written in such a way as to deal with the signal being resumed with a new string value. This application of signals can cause errors if there are any procedures between the signaller and the catcher that have their own idea about the location of the string. One possible fix (if such situations are possible) is to have a second signal

\[ \text{StringMoved: SIGNAL [old, new: STRING] = CODE;} \]

that is raised by AppendChar after StringBoundsFault is resumed.

The presence or absence of the ComponentList depends on whether the signal caught is declared to return values. In a Catch whose ExpressionList contains more than one signal, one can Resume only if all signals have equivalent types. For example:

\[ \text{ASig: TYPE = SIGNAL RETURNS [CARDINAL];} \]
\[ \text{sig1: ASig;} \]
\[ \text{sig2: ASig;} \]
\[ \text{sig3: SIGNAL RETURNS [CARDINAL];} \]
\[ \text{sig4: SIGNAL;} \]

\[ \text{ENABLE BEGIN} \]
\[ \text{sig1, sig2 => RESUME[3]; -- legal} \]
\[ \text{sig1, sig3 => RESUME[0]; -- legal} \]
\[ \text{sig1, sig4 => RESUME[1]; -- illegal} \]
\[ \text{END;} \]

8.3. Signals within signals *

What happens if, in the course of handling a signal, firstSignal, a catch phrase (or some procedure called by it) generates another signal, secondSignal? Handling nested signal generation is almost exactly like non-nested signal propagation. Generating the signal will call Signaller (recursively, since the instance of Signaller responsible for the first signal is still around), and it propagates the
new signal back through the call hierarchy by calling a second activation of *Signaller*, say "*Signaller2". When in the course of doing this it encounters the previous activation of *Signaller* ("*Signaller1"), then something different must be done.

If *firstSignal* is not the same as *secondSignal*, *Signaller2* propagates it right through *Signaller1*, and all the activations beyond it are also given a chance to catch *secondSignal*.

On the other hand, if *secondSignal = firstSignal*, then all of the routines whose frames lie beyond *Signaller1*, up to the frame containing the catch phrase called by *Signaller1*, have already had a chance to handle *firstSignal*, so they are not given it again. In order to skip around that section of the call hierarchy, *Signaller2* simply copies the appropriate state variables from *Signaller1*. Next, *Signaller2* skips over the frame containing the catch phrase (by following its return link), and continues propagating *secondSignal* normally.

For the programmer, the main import of nested signals is that one needs to consider, when writing a routine, not only what signals can be generated, directly or indirectly, by the called procedures, but also those which can be generated by catch phrases in that procedure or even the catch phrases of any calling procedures, also both directly or indirectly.
Mesa has, in addition to procedures, another mechanism by which programs may transfer control. This mechanism is called a PORT; PORTS allow separate modules or procedures to act as coroutines. When one calls a procedure and it returns, the procedure is finished; if the same operation is needed again, another call will create a new activation of it to perform that action. However, when a coroutine returns control, it does not finish and disappear. Calling it again only resumes it from where it left off. The advantage of this scheme is that the coroutine may keep some of its state from call to call encoded in its program counter: i.e., if it is at a certain place in its code, then that place does not need to be encoded somehow and saved as a variable in order to decide how to proceed when next called.

Actually, as described later, PORTS are normally used in pairs, just like electrical plugs and sockets, one for each side of the connection. If two coroutines $A$ and $B$ are connected, what is seen by $A$ as a call to $B$ appears to $B$ as a return from $A$, and vice-versa. Thus, both $A$ and $B$ regard the other as a facility to be called to accomplish some processing task. For instance, if $ReadFile$ is a coroutine for reading characters from a file which are then given, one at a time, to another coroutine, its view is that it reads characters from the file and calls the other coroutine to process them (in some unspecified way). $WriteFile$, on the other hand, a coroutine for writing characters into a file, would call a coroutine to get the next character to be written. Together these two coroutines could make a file copying program.

A coroutine needs to be able to send arguments and to receive results. The language facilities for doing this closely mirror procedure parameter and result lists. For example, a PORT over which $ReadFile$ could send a character would be declared by $ReadFile$ as

\[ \text{Out: PORT[ch: CHARACTER];} \]

The port over which $WriteFile$ receives a character, and which could be connected to $ReadFile$'s Out PORT, is declared as

\[ \text{In: PORT RETURNS[CHARACTER];} \]

There is only one other consequential difference between procedures and coroutines. A procedure can be called at any time because a new activation is created, which will always consume the arguments sent to it as soon as it begins. However, if two coroutines like $ReadFile$ and $WriteFile$ communicate, in order for the transfer of control and arguments to go smoothly, $WriteFile$ must be prepared to receive a character when $ReadFile$ sends it. Coroutines are not parallel processes, and one has to be started before the other, so it is guaranteed that the first attempt at transferring control between $ReadFile$ and $WriteFile$ will not work smoothly. Fortunately, Mesa provides a mechanism for starting a whole set of interconnected coroutines to get them past this start-up transient (sec. 9.2). The most important property of the mechanism is that the coroutines themselves need never be concerned about the startup transient — they are written as if it never happens.
9.1. Syntax and an example of PORTs

The syntax for declaring a port is the following:

\[
\text{PortTC ::= PORT ParameterList ReturnsClause | RESPONDING PORT ParameterList ReturnsClause}
\]

The ParameterList and ReturnsClause may both be empty, just as for procedures. RESPONDING PORTS are covered in section 9.3. The syntax for making a call on a port is exactly the same as for calls on procedures (both as statements and functions).

The following pair of program modules implement the coroutines ReadFile and WriteFile described earlier; they use the ports Out and In, respectively:

DIRECTORY

FileDeft: FROM "filedefs" USING [NUL, FileHandle, FileAccess, OpenFile, ReadChar, EndOfFile, CloseFile];

ReadFile: PROGRAM[name: STRING] IMPORTS FileDeft = BEGIN OPEN FileDeft;
Out: PORT[ch: CHARACTER];
input: FileHandle;
input + OpenFile[name: name, access: FileAccess[Read]];
STOP;
UNTIL EndOfFile[input]
DO
Out[ReadChar[input]];  -- PORT call: send a character from the file
ENDLOOP;
CloseFile[input];
Out[NUL];  -- send a null character to indicate end-of-file
END.

DIRECTORY

FileDefts: FROM "filedefs" USING [NUL, FileHandle, FileAccess, OpenFile, WriteChar, CloseFile];

WriteFile: PROGRAM[name: STRING] IMPORTS FileDefts = BEGIN OPEN FileDefts;
In: PORT RETURNS[ch: CHARACTER];
char: CHARACTER;
output: FileHandle;
output + OpenFile[name: name, access: FileAccess[New]];
STOP;
DO -- until In sends a NUL
char ← In[];  -- PORT call: get a character
if char = NUL THEN EXIT;  -- check for end of stream
WriteChar[output, char];  -- write the character into the file
ENDLOOP;
CloseFile[output];
END.

ReadFile first initializes its variables and opens the input file (with Read access). When it is restarted, it loops, reading characters from the file and sending them over its Out PORT until it reaches the end of the input file; then it sends a single NUL character. If it regains control, it simply returns.
WriteFile, after creating and opening a new output file, loops, reading characters from the port In and writing them to the output. If it receives a NUL character, it closes the output file and returns. Thus, if ReadFile and WriteFile's ports were connected so that they were working together as coroutines, ReadFile would never regain control after sending the NUL character.

9.2. Creating and starting coroutines

To set up the above two programs as coroutines, their respective ports must be connected, and then they must be started individually, with the start-up transient handled. This is usually done by another, controlling program like the following:

```
DIRECTORY
  TrapDefs: FROM "trapdefs" USING [PortFault],
  IODefs: FROM "iodefs" USING [ReadLine, WriteString],
  ReadFile: FROM "readfile",
  WriteFile: FROM "writefile";

CopyMaker: PROGRAM IMPORTS IODefs, reader: ReadFile, writer: WriteFile =
BEGIN OPEN IODefs;
  input: STRING ← [256];
  output: STRING ← [256];
  -- first ask the user for the names of the input and output files
  WriteString["Name of input file: "; ReadLine[input];
  WriteString["Name of output file: "; ReadLine[output];
  -- create and initialize instances of ReadFile and WriteFile;
  START reader[input];
  START writer[output];
  -- connect their ports and then restart them to get them synchronized
  CONNECT reader.In TO writer.Out;
  CONNECT writer.Out TO reader.In;
  RESTART writer[ ! TrapDefs.PortFault => CONTINUE];
  RESTART reader[ ! TrapDefs.PortFault => ERROR];
END.
```

Logically, CopyMaker is a very simple program. However, it must know how to start ReadFile and WriteFile and how to connect their ports (and it must handle the signal PortFault -- see below). This is typical of the use of PORTS: the coroutines themselves do not know (nor should they care) exactly which other program(s) they are connected to; each PORT is viewed as a virtual facility to be called to perform some task, such as providing the next input or taking an output.

CopyMaker first requests the names for the input file to be copied and the output file to which it should be copied. The names are read into the string variables input and output. Then an instance of ReadFile is made and initialized. Similarly, an instance of WriteFile is created and STARTed. When the NEWS are performed, pointers to the instances are stored (into reader and writer above).

After both instances have been created and initialized, CopyMaker performs the operations to get them past the startup transient. First it connects writer.In (i.e., WriteFile's In PORT) to reader.Out; this simply amounts to storing a pointer in writer.In to the PORT reader.Out. Then it connects reader.Out to writer.In.

Fine point:

The STARTs must be performed before the ports are connected. In general, it is not legal to access a module's variables before it has been started (and the variables have been initialized). Calls to procedures are allowed, however; they are handled by the start trap mechanism (sec. 7.8.3).
Once the CONNECTs are done, all that remains is to get the two coroutines synchronized. First, _WriteFile_ is **RESTARTed**; it makes a port call on _In_ to get the first character to be written into the file.

The port call almost works because _In_ is connected to another port. But, since _ReadFile_ is not waiting for control to return over its _Out_ port, it doesn’t quite work. This fact is detected because a part of the underlying representation of _Out_ indicates that no instance is **pending** on it (i.e., waiting to receive control via _Out_). This results in a **trap**, which is quickly converted into the **ERROR PortFault**. _CopyMaker_ clearly anticipated this as part of the normal startup transient (as evidenced by the presence of the catch phrase on the **START** statement). The **CONTINUE** in that catch phrase means: “forget about this signal and continue execution at the next statement in _CopyMaker_.”

The next action taken by _CopyMaker_ is to **RESTART** _ReadFile_. _ReadFile_ reads the first character from the input file and attempts a port call on _Out_, passing the character as its argument. **This is the end of startup transients**: this port call works. It works because _WriteFile_ was left pending on _In_ when it attempted to call it, even though that call did not go through completely. Since _WriteFile_ is pending on _In_, it resumes, stores the argument in _char_, and proceeds. From now on, port calls between _ReadFile_ and _WriteFile_ will go smoothly, with no further intervention by _CopyMaker_. (Moreover, a port call is more efficient than a procedure call because no frames are allocated and deallocated in the process).

When there are no more characters in the input file, _ReadFile_ sends a final NUL character which causes _WriteFile_ to close the output file and to return. This returns control to _CopyMaker_, who, in this example, also returns.

The above description skipped one or two important details of the startup process and port calls. The next section corrects those omissions and discusses the underlying representation of ports.

### 9.2.1. The CONNECT statement

The first CONNECT statement in _CopyMaker_ is equivalent to the following (illegal) assignment:

```plaintext
writer.In.link += @reader.Out;
```

This assignment is illegal because, at the language level, a **PORT** does not look like a record with a **link** component. Nevertheless, the code produced by the compiler for the CONNECT statement in _CopyMaker_ performs exactly this assignment (the compiler is allowed to treat PORTS in terms of their underlying representations, without regard to type - it implements type checking). Note that CONNECT is not a symmetric operation: it only connects in one direction.

The syntax for CONNECT is the following:

```plaintext
ConnectStmt ::= CONNECT expression TO expression
```

These expressions must both be valid **leftSides**. The first expression must conform to some **PORT** type, and the second may conform to either a **PORT** or a **PROCEDURE** type (see sec. 9.2.2 for a discussion of ports connected to procedures).

The types of the two expressions must be **port-compatible**. To be port-compatible, the result list of one must be compatible (see definition in sec. 5.2) with the parameter list of the other, **and** vice versa. This basically says that the first port sends what the second expects to receive, and the second sends what the first expects to receive.

Fine point:

In the present compiler, the CONNECT statement is not implemented.
9.2.2. Low-level actions during a PORT call

A PORT is represented as a record with two components, one of which is a pointer to another PORT, and one of which points to a frame (the frame which is pending on that PORT). Its definition is:

```pascal
Port: TYPE = MACHINE DEPENDENT RECORD
    [frame: POINTER TO Frame, -- internal view of a frame
        link: SELECT OVERLAID * FROM
            null => [value: NullControlLink],
            port => [portDesc: POINTER TO Port],
            procedure => [procDesc: ProcedureDescriptor],
    ]
ENDCASE
```

We will not discuss the internal format of the types Frame, ProcedureDescriptor, or NullControlLink here. The first two are the underlying representations for a frame and a procedure value, respectively. The last is just a special value which is used to initiate a trap if the port is used without having been connected first.

The variant part of a Port distinguishes three cases (how these cases are identified is a function of the underlying implementation). The null case is how a Port which has not been connected is represented; it is what causes a trap if a call on the port is made before it is connected (this is called a linkage fault). If the Port is connected to another Port (the normal case), then the port variant holds.

Procedure calls, port calls, and returns are all examples of control transfers: each suspends the execution of one activation and transfers control to another. They also perform other actions, such as creating or destroying frames, etc. Every control transfer from one activation to another has a source control link and a destination control link. By control link we mean a procedure value, a pointer to a port, or a pointer to a frame.

All the high level control transfers in Mesa are built from one common, low-level mechanism called XFER, which effects the transfer from a source to a destination. In fact, it is possible to bind any form of control link to any other; thus, if the program uses a port, it could be bound to a procedure, and calls on the port would actually result in calls on the procedure. A RETURN from the procedure would cause control to come back in through the port. Similarly, a procedure value could contain a pointer to a port, in which case calls on that "procedure" would actually result in a port transfer via the destination port to the coroutine pending on it.

The common part of a Port record is used when control is returning over a PORT. When a coroutine does a port call and is suspended, a pointer to its frame is assigned to the frame component of that port. Then, when control returns over that port (usually because of a port call on the port to which it is connected), the frame field is used to locate the instance which is to be resumed.

The value contained in the frame component may indicate that it is null. If so, a control fault trap will be generated should a transfer using that port ever occur. This condition can arise for two different reasons:

1. Due to startup transients, the instance which would normally be pending on that port is not.
2. There is a genuine error in the way that a configuration of coroutines has been constructed, and control is attempting to "loop back" into a coroutine. The simplest example of this situation is the following: consider a coroutine A with two ports, p1 and p2. If p1 were connected to p2, then a port call on p1 would clearly result in a control fault when p2 was reached in the call, since A cannot be pending on both p1 and p2 simultaneously.
The action taken on a control fault during a port call is described in the next section.

There is one last important detail about a port call: as part of the action of returning to a port, its link is set to point at the source port if the return is actually part of a port call. This constitutes an indirect return link. However, if the return is from a procedure to which the port is bound, then the link field is not changed. This is so that the procedure value in the port is not destroyed; thus, future calls on that port will always result in new activations of that procedure.

Storing an indirect return link in the link field of a destination port means that the next port call on it will cause control to return via the port from which control most recently arrived. Using this, one can write coroutines that may be invoked by more than one coroutine connected to a given port: control will always return to the last coroutine which sent control over that port. For instance, the coroutine WriteFile above could be given its input stream of characters from many sources. If the system procedures ReadLine and WriteString both had ports connected to the port In in an instance of WriteFile, then everything typed to the user and typed by him would be recorded in a typescript of his interactions with the system.

9.2.3. Control faults and linkage faults

When a control or a linkage fault occurs, Mesa changes the trap into the ERROR PortFault or LinkageFault, respectively. These signals are part of a Mesa system interface TrapDefs and should be imported from there by any program, such as CopyMaker, which configures coroutines. In TrapDefs they are defined as follows:

PortFault, LinkageFault: ERROR;

Generally, programs should not handle the LinkageFault signal; ports should be properly connected before they are used. We include it here only for completeness (the fine point at the end of this section discusses LinkageFaults further).

These signals, unlike most other signals, are not passed initially to the instance which caused the fault (call it the culprit), but rather are given first to its owner: the frame to which the culprit’s return link points. This is so that the owner may catch the signal and cause an UNWIND without the culprit’s frame being destroyed as it would normally be. In the previous example, CopyMaker is the owner and ReadFile and WriteFile are possible culprits.

Note: if the owner does not catch the PortFault or the LinkageFault signal, it may possibly be unwound itself. This would leave the culprit’s return link pointing to an invalid address, because the owner’s frame would have been freed.

The standard action taken by the owner when receiving a PortFault while starting a coroutine is to press on and start the other members of the configuration. CopyMaker follows this pattern; when it starts the instance of WriteFile and a control fault is generated, it simply exits the catch phrase for PortFault and starts the instance of ReadFile. This is the recommended way to start configurations of coroutines.

Fine point:

If the source port in a port call is unbound (i.e., not connected), a LinkageFault ERROR is generated. This cannot be handled in the same manner as a control fault. If the catcher of this signal causes an UNWIND, there will be no way to restart the activation which caused the linkage fault; it will be pending on a port, and RESTARTing it will cause an error. This difficulty makes starting coroutines before connecting their ports an ill-advised thing to do. It is much better to do the CONNECTs first, and then start each activation.
9.2.4. Saving arguments during faults

When a port call faults, the instance which attempted the call is left pending on the source port before the trap is changed into the PortFault or LinkageFault signal. This is done by a Mesa procedure called the FaultHandler, which is called in response to the trap. In the case of starting writer above, this procedure did the following:

(1) It set the instance of WriteFile to be pending on its In port (the trap process provides information about which instance caused the trap, and what the source port was);

(2) By some low-level control mechanisms, it invoked the Signaller (sec. 8.2) as if from the owner of writer and simultaneously did a RETURN. Thus, that activation of FaultHandler disappeared and the Signaller was invoked as a single action.

Later, when reader called Out, control returned to writer via In, which continued normally because it was pending on In. To writer it appeared as if the first port call worked correctly.

Reader's call on Out passed an argument along with control. If CopyMaker had started reader first, what would have happened to that argument? Given the above description of FaultHandler, the argument would have been lost: there were no provisions for buffering or saving arguments.

To handle this, the FaultHandler buffers any arguments passed over a port on which a fault occurs. Instead of performing action (2) above, it actually does the following:

(2') It buffers the arguments for the port call, makes it appear that it (the FaultHandler itself) is pending on the source port, then calls Signaller, but without destroying itself in doing so.

For the following discussion, assume that the startup sequence in CopyMaker had been written as follows (the order of starting reader and writer has been inverted):

```plaintext
... -- connect their ports and then restart them to get them synchronized
CONNECT reader.Out TO writer.In;
CONNECT writer.In TO reader.Out;
RESTART reader[ ! TrapDefs.PortFault => CONTINUE];
RESTART writer[ ! TrapDefs.PortFault => ERROR];
END.
```

The revised version of FaultHandler would then do the following when writer was RESTARTed and tried its first call on In:

The instance of FaultHandler which had left itself pending on Out would have been resumed instead of reader. FaultHandler would then have set reader.Out.frame so that reader was again pending on it. Finally, it would have transferred control back through writer.In along with the arguments it had saved from the original call, destroying itself in the process.

The only remaining question is: "How does the FaultHandler know whether or not arguments should be buffered?" This question is not trivial: for example, if every instance of FaultHandler buffered arguments for every trapped port call, including those for ports like In, extra "ghost" port calls would occur during startup. FaultHandler determines whether or not to save arguments by inspecting information left by the compiler in the object code of every port call. This decision is made by the compiler on the following basis:

Arguments should only be buffered for a port which is not a Responding Port and which does have a non-empty ParameterList.

The next section discusses Responding Ports.
9.3. **RESPONDING PORTS** *

The normal analogy between a port and a procedure in terms of passing arguments and receiving results breaks down in one case. If a port is used both for sending arguments and for receiving results, it might do so for either of the following two reasons:

- It sends arguments to be processed, and the returned results of the port call indicate how they were handled (this closely mirrors procedures).
- It receives data to be processed, and, having done so responds by sending results of the processing back over the same port (there is no procedure analog of this).

The second case can not be distinguished from the first by usage in a program because the actions of sending and receiving over a port are intrinsically intertwined with the notation for a Call. Thus, it would *not* be possible to determine whether *BothWays* was a normal or a responding port by looking at the following (partial) module:

```mesa
BothWays: PORT[S: STRING] RETURNS[t: STRING];
aString: STRING;
bString: STRING;

aString + BothWays[bString];
```

To resolve this difficulty, the programmer may declare a port to be **RESPONDING**. For example,

```mesa
InOut: RESPONDING PORT[response: {okay, error}] RETURNS[input: STRING];
```

The module using *InOut* responds with either *okay* or *notOkay* to each string it has received.

If *InOut* faults the first time it is used, the *FaultHandler* will not buffer the *response* value for that call. Since *InOut* must, for type conformance, be connected to a port such as

```mesa
OutIn: PORT[output: STRING] RETURNS[response: {okay, error}],
```

both initial argument lists (the *response* for the first call on *InOut*, and the *output* of the first call on *OutIn*) cannot be buffered. The keyword **RESPONDING** indicates which initial argument list should be discarded (*InOut*’s initial *response*, in this case). For similar reasons, a responding port may not be connected to a procedure, and two responding ports may not be connected together.

**Fine point:**

In the current compiler, **RESPONDING PORTs** are not implemented
Mesa provides language support for concurrent execution of multiple processes. This allows programs that are inherently parallel in nature to be clearly expressed. The language also provides facilities for synchronizing such processes by means of entry to monitors and waiting on condition variables.

The next section discusses the forking and joining of concurrent process. Later sections deal with monitors, how their locks are specified, and how they are entered and exited. Condition variables are discussed, along with their associated operations.

10.1. Concurrent execution, FORK and JOIN.

The FORK and JOIN statements allow parallel execution of two procedures. Their use also requires the new data type PROCESS. Since the Mesa process facilities provide considerable flexibility, it is easiest to understand them by first looking at a simple example.

10.1.1. A process example

Consider an application with a front-end routine providing interactive composition and editing of input lines:

\[
\text{ReadLine: PROCEDURE [s: STRING] RETURNS [CARDINAL] =}
\]
\[
\begin{align*}
\text{BEGIN} \\
\text{c: CHARACTER;}
\text{s.length} + 0;
\text{DO}
\text{c} \leftarrow \text{ReadChar[];}
\text{IF ControlCharacter[c] THEN DoAction[c]}
\text{ELSE AppendChar[s,c];}
\text{IF c = CR THEN RETURN [s.length];}
\text{ENDLOOP;}
\text{END;}
\end{align*}
\]

The call

\[
\text{n + ReadLine[buffer];}
\]

will collect a line of user type-in up to a CR and put it in some string named buffer. Of course, the caller cannot get anything else accomplished during the type-in of the line. If there is anything else that needs doing, it can be done concurrently with the type-in by forking to ReadLine instead of calling it:
\[ p \leftarrow \text{FORK } \text{ReadLine(buffer)}; \]
\[ \ldots \]
\[ \langle \text{concurrent computation} \rangle \]
\[ \ldots \]
\[ n \leftarrow \text{JOIN } p; \]

This allows the statements labeled \( \langle \text{concurrent computation} \rangle \) to proceed in parallel with user typing (clearly, the \( \langle \text{concurrent computation} \rangle \) should not reference the string \text{buffer}). The \text{FORK} construct spawns a new process whose result type matches that of \text{ReadLine}. (\text{ReadLine} is referred to as the "root procedure" of the new process.)

\[ p: \text{PROCESS} \text{RETURNS [CARDINAL]}; \]

Later, the results are retrieved by the \text{JOIN} statement, which also deletes the spawned process. Obviously, this must not occur until both processes are ready (i.e. have reached the \text{JOIN} and the \text{RETURN}, respectively); this rendezvous is synchronized automatically by the process facility.

Note that the types of the arguments and results of \text{ReadLine} are always checked at compile time, whether it is called or forked.

The one major difference between calling a procedure and forking to it is in the handling of signals; see section 10.5.1 for details.

10.1.2. Process language constructs

The declaration of a \text{PROCESS} is similar to the declaration of a \text{PROCEDURE}, except that only the return record is specified. The syntax is formally specified as follows:

\[
\begin{align*}
\text{TypeConstructor} &::= \ldots | \text{ProcessTC} \\
\text{ProcessTC} &::= \text{PROCESS ReturnsClause} \\
\text{ReturnsClause} &::= \text{empty} | \text{RETURNS ResultList} \quad \text{-- from sec. 5.1.} \\
\text{ResultList} &::= \text{FieldList} \quad \text{-- from sec. 5.1.}
\end{align*}
\]

Suppose that \( f \) is a procedure and \( p \) a process. In order to fork \( f \) and assign the resulting process to \( p \), the \text{ReturnClause} of \( f \) and that of \( p \) must be compatible, as described in sec 5.2.

The syntax for the \text{FORK} and \text{JOIN} statements is straightforward:

\[
\begin{align*}
\text{Statement} &::= \ldots | \text{JoinCall} \\
\text{Expression} &::= \ldots | \text{ForkCall} | \text{JoinCall} \\
\text{ForkCall} &::= \text{FORK Call} \\
\text{JoinCall} &::= \text{JOIN Call} \\
\text{Call} &::= \text{(see sections 5.4 and 8.2.1)}
\end{align*}
\]

The \text{ForkCall} always returns a value (of type \text{PROCESS}) and thus a \text{FORK} cannot stand alone as a statement. Unlike a procedure call, which returns a \text{RECORD}, the value of the \text{FORK} cannot be discarded by writing an empty extractor. The action specified by the \text{FORK} is to spawn a process parallel to the current one, and to begin it executing the named procedure.
The **JoinCall** appears as either a statement or an expression, depending upon whether or not the process being joined has an empty **Returns Clause**. It has the following meaning: When the forked procedure has executed a `RETURN` and the **JOIN** is executed (in either order),

the returning process is deleted, and

the joining process receives the results, and continues execution.

A catch phrase can be attached to either a **FORK** or **JOIN** by specifying it in the Call. Note, however, that such a catch phrase does not catch signals incurred during the execution of the procedure; see section 10.5.1 for further details.

There are several other important similarities with normal procedure calls which are worth noting:

The types of all arguments and results are checked at compile time.

There is no **intrinsic** rule against multiple activations (calls and/or forks) of the same procedure coexisting at once. Of course, it is always possible to write procedures which will work incorrectly if used in this way, but the mechanism itself does not prohibit such use.

One expected pattern of usage of the above mechanism is to place a matching FORK/JOIN pair at the beginning and end of a single textual unit (i.e. procedure, compound statement, etc.) so that the computation within the textual unit occurs in parallel with that of the spawned process. This style is encouraged, but is **not** mandatory; in fact, the matching FORK and JOIN need not even be done by the same process. Care must be taken, of course, to insure that each spawned process is joined only once, since the result of joining an already deleted process is undefined. Note that the spawned process always begins and ends its life in the same textual unit (i.e. the target procedure of the FORK).

While many processes will tend to follow the FORK/JOIN paradigm, there will be others whose role is better cast as continuing provision of services, rather than one-time calculation of results. Such a "detached" process is never joined. If its lifetime is bounded at all, its deletion is a private matter, since it involves neither synchronization nor delivery of results. No language features are required for this operation; see the runtime documentation for the description of the system procedure provided for detaching a process.

### 10.2. Monitors

Generally, when two or more processes are cooperating, they need to interact in more complicated ways than simply forking and joining. Some more general mechanism is needed to allow orderly, synchronized interaction among processes. The interprocess synchronization mechanism provided in Mesa is a variant of **monitors** adapted from the work of Hoare, Brinch Hansen, and Dijkstra. The underlying view is that interaction among processes always reduces to carefully synchronized access to shared data, and that a proper vehicle for this interaction is one which unifies:

- the synchronization
- the shared data
- the body of code which performs the accesses

The Mesa monitor facility allows considerable flexibility in its use. Before getting into the details, let us first look at a slightly over-simplified description of the mechanism and a simple example. The remainder of this section deals with the basics of monitors (more complex uses are described in section 10.4); **WAIT** and **NOTIFY** are described in section 10.3.
10.2.1. An overview of monitors

A monitor is a module instance. It thus has its own data in its global frame, and its own procedures for accessing that data. Some of the procedures are public, allowing calls into the monitor from outside. Obviously, conflicts could arise if two processes were executing in the same monitor at the same time. To prevent this, a monitor lock is used for mutual exclusion (i.e. to insure that only one process may be in each monitor at any one time). A call into a monitor (to an entry procedure) implicitly acquires its lock (waiting if necessary), and returning from the monitor releases it. The monitor lock serves to guarantee the integrity of the global data, which is expressed as the monitor invariant -- i.e an assertion defining what constitutes a "good state" of the data for that particular monitor. It is the responsibility of every entry procedure to restore the monitor invariant before returning, for the benefit of the next process entering the monitor.

Things are complicated slightly by the possibility that one process may enter the monitor and find that the monitor data, while in a good state, nevertheless indicates that that process cannot continue until some other process enters the monitor and improves the situation. The WAIT operation allows the first process to release the monitor lock and await the desired condition. The WAIT is performed on a condition variable, which is associated by agreement with the actual condition needed. When another process makes that condition true, it will perform a NOTIFY on the condition variable, and the waiting process will continue from where it left off (after reacquiring the lock, of course.)

For example, consider a fixed block storage allocator providing two entry procedures: Allocate and Free. A caller of Allocate may find the free storage exhausted and be obliged to wait until some caller of Free returns a block of storage.

```
StorageAllocator: MONITOR =
BEGIN
    StorageAvailable: CONDITION;
    FreeList: POINTER;

Allocate: ENTRY PROCEDURE RETURNS [p: POINTER] =
BEGIN
    WHILE FreeList = NIL DO
        WAIT StorageAvailable
    ENDLOOP;
    p = FreeList; FreeList = p.next;
END;

Free: ENTRY PROCEDURE [p: POINTER] =
BEGIN
    p.next = FreeList; FreeList = p;
    NOTIFY StorageAvailable
END;
END.
```

Note that it is clearly undesirable for two asynchronous processes to be executing in the StorageAllocator at the same time. The use of entry procedures for Allocate and Free assures mutual exclusion. The monitor lock is released while waiting in Allocate in order to allow Free to be called (this also allows other processes to call Allocate as well, leading to several processes waiting on the queue for StorageAvailable).
10.2.2. **Monitor locks**

The most basic component of a monitor is its *monitor lock*. A monitor lock is a predefined type, which can be thought of as a small record:

\[
\text{MONITORLOCK: TYPE } = \text{PRIVATE RECORD [locked: BOOLEAN, queue: Queue];}
\]

The monitor lock is private; its fields are never accessed explicitly by the Mesa programmer. Instead, it is used implicitly to synchronize entry into the monitor code, thereby authorizing access to the monitor data (and in some cases, other resources, such as I/O devices, etc.). The next section describes several kinds of monitors which can be constructed from this basic mechanism. In all of these, the idea is the same: during entry to a monitor, it is necessary to acquire the monitor lock by:

1. waiting (in the queue) until: \( \text{locked} = \text{FALSE} \),
2. setting: \( \text{locked} \leftrightarrow \text{TRUE} \).

10.2.3. **Declaring monitor modules, ENTRY and INTERNAL procedures**

In addition to a collection of data and an associated lock, a monitor contains a set of procedures that do operations on the data. Monitor modules are declared much like program or definitions modules; for example:

\[
M: \text{MONITOR \{arguments\} = }
\begin{align*}
\text{BEGIN} \\
\ldots
\text{END.}
\end{align*}
\]

The procedures in a monitor module are of three kinds:

- Entry procedures
- Internal procedures
- External procedures

Every monitor has one or more entry procedures; these acquire the monitor lock when called, and are declared as:

\[
P: \text{ENTRY PROCEDURE \{arguments\} = \ldots}
\]

The entry procedures will usually comprise the set of public procedures visible to clients of the monitor module. (There are some situations in which this is not the case; see external procedures, below). The usual Mesa default rules for public and private procedures apply.

Many monitors will also have internal procedures: common routines shared among the several entry procedures. These execute with the monitor lock held, and may thus freely access the monitor data (including condition variables) as necessary. Internal procedures should be private, since direct calls to them from outside the monitor would bypass the acquisition of the lock (for monitors implemented as multiple modules, this is not quite right; see section 10.4, below). Internal procedures can be called only from an entry procedure or another internal procedure. They are declared as follows:

\[
Q: \text{INTERNAL PROCEDURE \{arguments\} = \ldots}
\]
The attributes ENTRY or INTERNAL may be specified on a procedure only in a monitor module. Section 10.2.4 describes how one declares an interface for a monitor.

Some monitor modules may wish to have external procedures. These are declared as normal non-monitor procedures:

\[ R: \text{PROCEDURE} \ [\text{arguments}] = \ldots \]

Such procedures are logically outside the monitor, but are declared within the same module for reasons of logical packaging. For example, a public external procedure might do some preliminary processing and then make repeated calls into the monitor proper (via a private entry procedure) before returning to its client. Being outside the monitor, an external procedure must not reference any monitor data (including condition variables), nor call any internal procedures. The compiler checks for calls to internal procedures and usage of the condition variable operations (WAIT, NOTIFY, etc.) within external procedures, but does not check for accesses to monitor data.

A fine point:
Actually, unchanging read-only global variables may be accessed by external procedures; it is changeable monitor data that is strictly off-limits.

Generally speaking, a chain of procedure calls involving a monitor module has the general form:

Client procedure -- outside module
\[ \downarrow \]
External procedure(s) -- inside module but outside monitor
\[ \downarrow \]
Entry procedure -- inside monitor
\[ \downarrow \]
Internal procedure(s) -- inside monitor

Any deviation from this pattern is likely to be a mistake. A useful technique to avoid bugs and increase the readability of a monitor module is to structure the source text in the corresponding order:

\[ M: \text{MONITOR} = \]
\begin{verbatim}
BEGIN
<External procedures>
<Entry procedures>
<Internal procedures>
<Initialization (main-body) code>
END.
\end{verbatim}

10.2.4. Interfaces to monitors

In Mesa, the attributes ENTRY and INTERNAL are associated with a procedure's body, not with its type. Thus they cannot be specified in a DEFINITIONS module. Typically, internal procedures are not exported anyway, although they may be for a multi-module monitor (see section 10.4.4). In fact, the compiler will issue a warning when the combination PUBLIC INTERNAL occurs.

From the client side of an interface, a monitor appears to be a normal program module, hence the keywords MONITOR and ENTRY do not appear. For example, a monitor \( M \) with entry procedures \( P \) and \( Q \) might appear as:
10.2.5. Interactions of processes and monitors

One interaction should be noted between the process spawning and monitor mechanisms as defined so far. If a process executing within a monitor forked to an internal procedure of the same monitor, the result would be two processes inside the monitor at the same time, which is the exact situation that monitors are supposed to avoid. The following rule is therefore enforced:

A FORK may have as its target any procedure except an internal procedure of a monitor.

A fine point:

In the case of a multi-module monitor (see section 10.4.4) calls to other monitor procedures through an interface cannot be checked for the INTERNAL attribute, since this information is not available in the interface (see section 10.2.4).

10.3. Condition Variables

Condition variables are declared as:

c: CONDITION;

The content of a condition variable is private to the process mechanism; condition variables may be accessed only via the operations defined below. It is important to note that it is the condition variable which is the basic construct; a condition (i.e. the contents of a condition variable) should not itself be thought of as a meaningful object; it may not be assigned to a condition variable, passed as a parameter, etc.

10.3.1. Wait, notify, and broadcast

A process executing in a monitor may find some condition of the monitor data which forces it to wait until another process enters the monitor and improves the situation. This can be accomplished using a condition variable, and the three basic operations: WAIT, NOTIFY, and BROADCAST, defined by the following syntax:

\[
\begin{align*}
\text{Statement} & \quad ::= \quad \ldots \mid \text{WaitStmt} \mid \text{NotifyStmt} \\
\text{WaitStmt} & \quad ::= \quad \text{WAIT} \ \text{Variable} \ \text{OptCatchPhrase} \\
\text{NotifyStmt} & \quad ::= \quad \text{NOTIFY} \ \text{Variable} \mid \text{BROADCAST} \ \text{Variable}
\end{align*}
\]

A condition variable \(c\) is always associated with some Boolean expression describing a desired state of the monitor data, yielding the general pattern:

Process waiting for condition:

\[
\begin{align*}
\text{WHILE} \ \sim \text{BooleanExpression} \ \text{DO} \\
\text{WAIT} \ c \\
\text{ENDLOOP};
\end{align*}
\]
Process making condition true:

make BooleanExpression true; -- i.e. as side effect of modifying global data
NOTIFY c;

Consider the storage allocator example from section 10.2.1. In this case, the desired BooleanExpression is "FreeList ≠ NIL". There are several important points regarding **wait** and **notify**, some of which are illustrated by that example:

**wait** always releases the lock while waiting, in order to allow entry by other processes, including the process which will do the **notify** (e.g. **Allocate** must not lock out the caller of **Free** while waiting, or a deadlock will result). Thus, the programmer is always obliged to restore the monitor invariant (return the monitor data to a "good state") before doing a **wait**.

**notify**, on the other hand, retains the lock, and may thus be invoked without restoring the invariant; the monitor data may be left in an arbitrary state, so long as the invariant is restored before the next time the lock is released (by exiting an entry procedure, for example).

A **notify** directed to a condition variable on which no one is waiting is simply discarded. Moreover, the built-in test for this case is more efficient than any explicit test that the programmer could make to avoid doing the extra **notify**. (Thus, in the example above, **Free** always does a **notify**, without attempting to determine if it was actually needed.)

Each **wait** must be embedded in a loop checking the corresponding condition. (E.g. **Allocate**, upon being notified of the **StorageAvailable** condition, still loops back and tests again to insure that the freelist is actually non-empty.) This rechecking is necessary because the condition, even if true when the **notify** is done, may become false again by the time the awakened process gets to run. (Even though the freelist is always non-empty when **Free** does its **notify**, a third process could have called **Allocate** and emptied the freelist before the waiting process got a chance to inspect it.)

Given that a process awakening from a **wait** must be careful to recheck its desired condition, the process doing the **notify** can be somewhat more casual about insuring that the condition is actually true when it does the **notify**. This leads to the notion of a covering condition variable, which is notified whenever the condition desired by the waiting process is likely to be true; this approach is useful if the expected cost of false alarms (i.e. extra wakeups that test the condition and wait again) is lower than the cost of having the notifier always know precisely what the waiter is waiting for.

The last two points are somewhat subtle, but quite important; condition variables in Mesa act as suggestions that their associated Boolean expressions are likely to be true and should therefore be rechecked. They do not guarantee that a process, upon awakening from a **wait**, will necessarily find the condition it expects. The programmer should never write code which implicitly assumes the truth of some condition simply because a **notify** has occurred.

It is often the case that the user will wish to notify all processes waiting on a condition variable. This can be done using:

```java
BROADCAST c;
```

This operation can be used when several of the waiting processes should run, or when some waiting process should run, but not necessarily the head of the queue.
Consider a variation of the StorageAllocator example:

\begin{verbatim}
StorageAllocator: MONITOR =
    BEGIN
        StorageAvailable: CONDITION;
    ...

    BEGIN
    UNIL <storage chunk of size words is available> DO
        WAIT StorageAvailable
    ENDLOOP;
    p + <remove chunk of size words>;
    END;

Free: ENTRY PROCEDURE [p: POINTER, size: CARDINAL] =
    BEGIN
        <put back storage chunk of size words>
        BROADCAST StorageAvailable
    ...
    END;
END.
\end{verbatim}

In this example, there may be several processes waiting on the queue of StorageAvailable, each with a different size requirement. It is not sufficient to simply NOTIFY the head of the queue, since that process may not be satisfied with the newly available storage while another waiting process might be. This is a case in which BROADCAST is needed instead of NOTIFY.

An important rule of thumb: it is always correct to use a BROADCAST. NOTIFY should be used instead of BROADCAST if both of the following conditions hold:

It is expected that there will typically be several processes waiting in the condition variable queue (making it expensive to notify all of them with a BROADCAST), and

It is known that the process at the head of the condition variable queue will always be the right one to respond to the situation (making the multiple notification unnecessary);

If both of these conditions are met, a NOTIFY is sufficient, and may represent a significant efficiency improvement over a BROADCAST. The allocator example in section 10.2.1 is a situation in which NOTIFY is preferrable to BROADCAST.

As described above, the condition variable mechanism, and the programs using it, are intended to be robust in the face of "extra" NOTIFYS. The next section explores the opposite problem: "missing" NOTIFYS.

A fine point:

When a program WAITs, it releases the monitor lock. When it returns from the wait, it reacquires the lock. The address of the condition variable has to be calculated twice. If this address is obtained by a complicated expression, there is a subtle restriction. The address calculation cannot do a WAIT in the same process. In other words, consider the procedure
10.3.2. Timeouts

One potential problem with waiting on a condition variable is the possibility that one may wait "too long." There are several ways this could happen, including:

- Hardware error (e.g. "lost interrupt")
- Software error (e.g. failure to do a NOTIFY)
- Communication error (e.g. lost packet)

To handle such situations, waits on condition variables are allowed to time out. This is done by associating a timeout interval with each condition variable, which limits the delay that a process can experience on a given WAIT operation. If no NOTIFY has arrived within this time interval, one will be generated automatically. The Mesa language does not currently have a facility for setting the timeout field of a CONDITION variable. See the runtime documentation for the description of the system procedure provided for this operation.

The waiting process will perceive this event as a normal NOTIFY. (Some programs may wish to distinguish timeouts from normal NOTIFYS; this requires checking the time as well as the desired condition on each iteration of the loop.)

No facility is provided to time out waits for monitor locks. This is because there would be, in general, no way to recover from such a timeout.

10.4. More about Monitors

The next few sections deal with the full generality of monitor locks and monitors.

10.4.1. The LOCKS clause

Normally, a monitor's data comprises its global variables, protected by the special global variable LOCK:

\[ \text{LOCK: monitorlock}; \]

This implicit variable is declared automatically in the global frame of any module whose heading is of the form:

\[ M: \text{monitor } [\text{arguments}] \]
\[ \quad \text{IMPORTS} \ldots \]
\[ \quad \text{EXPORTS} \ldots = \]

In such a monitor it is generally not necessary to mention \textit{LOCK} explicitly at all. For more general use of the monitor mechanism, it is necessary to declare at the beginning of the monitor module exactly which \text{MONITORLOCK} is to be acquired by entry procedures. This declaration appears as part of the program type constructor that is at the head of the module. The syntax is as follows:
If the LocksClause is empty, entry to the monitor is controlled by the distinguished variable LOCK (automatically supplied by the compiler). Otherwise, the LocksClause must designate a variable of type MONITORLOCK, a record containing a distinguished lock field (see section 10.4.2), or a pointer that can be dereferenced (perhaps several times) to yield one of the preceding. If a LocksClause is present, the compiler does not generate the variable LOCK.

If the USING clause is absent, the lock is located by evaluating the LOCKS expression in the context of the monitor’s main body; i.e., the monitor’s parameters, imports, and global variables are visible, as are any identifiers made accessible by a global OPEN. Evaluation occurs upon entry to, and again upon exit from, the entry procedures (and for any WAITS in entry or internal procedures). The location of the designated lock can thus be affected by assignments within the procedure to variables in the LOCKS expression. To avoid disaster, it is essential that each reevaluation yield a designator of the same MONITORLOCK. This case is described further in section 10.4.4.

If the USING clause is present, the lock is located in the following way: every entry or internal procedure must have a parameter with the same identifier and a compatible type as that specified in the USING clause. The occurrences of that identifier in the LOCKS clause are bound to that procedure parameter in every entry procedure (and internal procedure doing a WAIT). The same care is necessary with respect to reevaluation; to emphasize this, the distinguished argument is treated as a read-only value within the body of the procedure. See section 10.4.5 for further details.

10.4.2. Monitored records

For situations in which the monitor data cannot simply be the global variables of the monitor module, a monitored record can be used:

\[ r: \text{MONITORED RECORD} [x: \text{INTEGER}, \ldots ]; \]

A monitored record is a normal Mesa record, except that it contains an automatically declared field of type MONITORLOCK. As usual, the monitor lock is used implicitly to synchronize entry into the monitor code, which may then access the other fields in the monitored record. The fields of the monitored record must not be accessed except from within a monitor which first acquires its lock. In analogy with the global variable case, the monitor lock field in a monitored record is given the special name LOCK; generally, it need not be referred to explicitly (except during initialization; see section 10.6).

A fine point:

A more general form of monitor lock declaration is discussed in section 10.4.6

CAUTION: If a monitored record is to be passed around (e.g. as an argument to a procedure) this should always be done by reference using a POINTER TO MONITORED RECORD. Copying a monitored record (e.g. passing it by value) will generally lead to chaos.

10.4.3. Monitors and module instances

Even when all the procedures of a monitor are in one module, it is not quite correct to think of the module and the monitor as identical. For one thing, a monitor module, like an ordinary program module, may have several instances. In the most straightforward case, each instance constitutes a
separate monitor. More generally, through the use of monitored records, the number of monitors may be larger or smaller than the number of instances of the corresponding module(s). The crucial observation is that in all cases:

*There is a one-to-one correspondence between monitors and monitor locks.*

The generalization of monitors through the use of monitored records tends to follow one of two patterns:

*Multi-module monitors*, in which several module instances implement a single monitor.

*Object monitors*, in which a single module instance implements several monitors.

A fine point:

These two patterns are *not* mutually exclusive; multi-module object monitors are possible, and may occasionally prove necessary.

### 10.4.4. Multi-module monitors

In implementing a monitor, the most obvious approach is to package all the data and procedures of the monitor within a single module instance (if there are multiple instances of such a module, they constitute separate monitors and share nothing except code.) While this will doubtless be the most common technique, the monitor may grow too large to be treated as a single module.

Typically, this leads to multiple modules. In this case the mechanics of constructing the monitor are changed somewhat. There must be a central location that contains the monitor lock for the monitor implemented by the multiple modules. This can be done either by using a `MONITORED RECORD` or by choosing one of the modules to be the "root" of the monitor. Consider the following example:

```mesa
BigMonRoot: MONITOR IMPORTS ••• EXPORTS ••• =
BEGIN
  monitorDatum1: ...
  monitorDatum2: ...
  p1: PUBLIC ENTRY PROCEDURE ...
  ...
END.

BigMonA: MONITOR
  LOCKS root -- could equivalently say root.LOCK
  IMPORTS root: BigMonRoot ... EXPORTS ... SHARES BigMonRoot =
  BEGIN
  ...
  p2: PUBLIC ENTRY PROCEDURE ...
  x ← root.monitorDatum1; -- access the protected data of the monitor
  ...
  END.

BigMonB: MONITOR
  LOCKS root
  IMPORTS root: BigMonRoot ... EXPORTS ... SHARES BigMonRoot =
  BEGIN OPEN root;
  ...
  p3: PUBLIC ENTRY PROCEDURE ...
```
monitorDatum2 + ...; -- access the protected data via an OPEN
...
END.

The monitor \textit{BigMon} is implemented by three modules. The modules \textit{BigMonA} and \textit{BigMonB} have a \texttt{LOCKS} clause to specify the location of the monitor lock: in this case, the distinguished variable \texttt{LOCK} in \texttt{BigMonRoot}. When any of the entry procedures \texttt{p1}, \texttt{p2}, or \texttt{p3} is called, this lock is acquired (waiting if necessary), and is released upon returning. The reader can verify that no two independent processes can be in the monitor at the same time.

Note that since the \texttt{LOCK} field is private in \texttt{BigMonRoot}, the modules \texttt{BigMonA} and \texttt{BigMonB} must \texttt{SHARE} \texttt{BigMonRoot}. Another way to accomplish access to the lock would be to specify a \texttt{PUBLIC GlobalAccess} (sec. 7.5) for \texttt{BigMonRoot}.

Another means of implementing multi-module monitors is by means of a \texttt{MONITORED RECORD}. Use of \texttt{OPEN} allows the fields of the record to be referenced without qualification. Such a monitor is written as:

\begin{verbatim}
MonitorData: TYPE = MONITORED RECORD [x: INTEGER, ...];

MonA: MONITOR [pm: POINTER TO MonitorData]
    LOCKS pm
    IMPORTS ...;
    EXPORTS ... =
    BEGIN OPEN pm;
    P: ENTRY PROCEDURE [...] =
    BEGIN
    ...
    x + x+1; -- access to a monitor variable
    ...
    END;
    ...
    END.
\end{verbatim}

The \texttt{LOCKS} clause in the heading of this module (and each other module of this monitor) leads to a \texttt{MONITORED RECORD}. Of course, in all such multi-module monitors, the \texttt{LOCKS} clause will involve one or more levels of indirection (\texttt{POINTER TO MONITORED RECORD}, etc.) since passing a monitor lock by value is not meaningful. As usual, Mesa will provide one or more levels of automatic dereferencing as needed.

More generally, the target of the \texttt{LOCKS} clause can evaluate to a \texttt{MONITORLOCK} (i.e. the example above is equivalent to writing "\texttt{LOCKS pm.LOCK}"").

\textbf{CAUTION:} The meaning of the target expression of the \texttt{LOCKS} clause \textit{must not change} between the call to the entry procedure and the subsequent return (i.e. in the above example, changing \texttt{pm} would invariably be an error) since this would lead to a different monitor lock being released than was acquired, resulting in total chaos.

There are a few other issues regarding multi-module monitors which arise any time a tightly coupled piece of Mesa code must be split into multiple module instances and then spliced back together. For example:

If the lock is in a \texttt{MONITORED RECORD}, the monitor data will probably need to be in the record also. While the global variables of such a multi-module monitor are covered by the
monitor lock, they do not constitute monitor data in the normal sense of the term, since they are not uniformly visible to all the module instances.

Making the internal procedures of a multi-module monitor private will not work if one module wishes to call an internal procedure in another module. (Such a call is perfectly acceptable so long as the caller already holds the monitor lock). Instead, a second interface (hidden from the clients) is needed as part of the "glue" holding the interface together. Note however, that Mesa cannot currently check that the procedure being called through the interface is an internal one (see section 10.2.4).

A fine point:

The compiler will complain about the public internal procedures, but this is just a warning.

10.4.5. Object monitors

Some applications deal with objects, implemented, say, as records named by pointers. Often it is necessary to ensure that operations on these objects are atomic, i.e., once the operation has begun, the object will not be otherwise referenced until the operation is finished. If a module instance provides operations on some class of objects, the simplest way of guaranteeing such atomicity is to make the module instance a monitor. This is logically correct, but if a high degree of concurrency is expected, it may create a bottleneck; it will serialize the operations on all objects in the class, rather than on each of them individually. If this problem is deemed serious, it can be solved by implementing the objects as monitored records, thus effectively creating a separate monitor for each object. A single module instance can implement the operations on all the objects as entry procedures, each taking as a parameter the object to be locked. The locking of the parameter is specified in the module heading via a LocksClause with a using clause. For example:

ObjectRecord: type = monitored record [ ... ];

ObjectHandle: type = pointer to ObjectRecord;

ObjectManager: monitor [arguments]
    Locks object using object: ObjectHandle
    imports . . .
    exports . . . =
    begin
    Operation: public entry procedure [object: ObjectHandle, . . . ] =
        begin
        . . .
        end;
        . . .
        end.

Note that the argument of using is evaluated in the scope of the arguments to the entry procedures, rather than the global scope of the module. In order for this to make sense, each entry procedure, and each internal procedure that does a wait, must have an argument which matches exactly the name and type specified in the using subclause. All other components of the argument of locks are evaluated in the global scope, as usual.

As with the simpler form of locks clause, the target may be a more complicated expression and/or may evaluate to a monitor lock rather than a monitored record. For example:

Locks p.q.lock using p: pointer to ComplexRecord . . .
CAUTION: Again, the meaning of the target expression of the LOCKS clause must not change between the call to the entry procedure and the subsequent return. (I.e. in the above example, changing \( p \) or \( p.q \) would almost surely be an error.)

CAUTION: It is important to note that global variables of object monitors are very dangerous; they are not covered by a monitor lock, and thus do not constitute monitor data. If used at all, they must be set only at module initialization time and must be read-only thereafter.

10.4.6. Explicit declaration of monitor locks

It is possible to declare monitor locks explicitly:

```plaintext
myLock: MONITORLOCK;
```

The normal cases of monitors and monitored records are essentially stylized uses of this facility via the automatic declaration of \( LOCK \), and should cover all but the most obscure situations. For example, explicit declarations are useful in defining MACHINE DEPENDENT monitored records. (Note that the LOCKS clause becomes mandatory when an explicitly declared monitor lock is used.) More generally, explicit declarations allow the programmer to declare records with several monitor locks, declare locks in local frames, and so on; this flexibility can lead to a wide variety of subtle bugs, hence use of the standard constructs whenever possible is strongly advised.

10.4.7 Inline ENTRY procedures

The syntax for definitions modules allows the specification of a LOCKS clause. This is to allow inline ENTRY PROCEDURES to be declared in the interface. In order for this to make sense, the monitor lock must be an interface variable, or the procedures must deal with an object style monitor. No special restrictions (other than those that apply to all INLINE bodies) need be met when declaring inline ENTRY PROCEDURES within the program module of a monitor.

10.5. Signals

10.5.1. Signals and processes

Each process has its own call stack, down which signals propagate. If the signaller scans to the bottom of the stack and finds no catch phrase, the signal is propagated to the debugger. The important point to note is that forking to a procedure is different from calling it, in that the forking creates a gap across which signals cannot propagate. This implies that in practice, one cannot casually fork to any arbitrary procedure. The only suitable targets for forks are procedures which catch any signals they incur, and which never generate any signals of their own.

10.5.2. Signals and monitors

Signals require special attention within the body of an entry procedure. A signal raised with the monitor lock held will propagate without releasing the lock and possibly invoke arbitrary computations. For errors, this can be avoided by using the RETURN WITH ERROR construct:

```plaintext
RETURN WITH ERROR NoSuchObject;
```

Recall from Chapter 8 that this statement has the effect of removing the currently executing frame from the call chain before issuing the ERROR. If the statement appears within an entry procedure,
the monitor lock is released before the error is started as well. Naturally, the monitor invariant must be restored before this operation is performed.

For example, consider the following program segment:

\[
\begin{align*}
\text{Failure: } & \text{ ERROR } [\text{kind: CARDINAL}] = \text{ CODE;} \\
\text{Proc: } & \text{ ENTRY PROCEDURE } [. . .] \text{ RETURNS } [c1, c2: \text{CHARACTER}] = \\
& \text{ BEGIN } \\
& \hspace{1cm} \text{ ENABLE UNWIND } \Rightarrow . . . \\
& \hspace{1cm} \text{ IF cond1 THEN ERROR Failure}[1]; \\
& \hspace{1cm} \text{ IF cond2 THEN RETURN WITH ERROR Failure}[2]; \\
& \hspace{1cm} \text{ END;}
\end{align*}
\]

Execution of the construct \text{ERROR Failure}[1] raises a signal that propagates until some catch phrase specifies an exit. At that time, unwinding begins; the catch phrase for \text{UNWIND} in \text{Proc} is executed and then \text{Proc}'s frame is destroyed. Within an entry procedure such as \text{Proc}, the lock is held until the unwind (and thus through unpredictable computation performed by catch phrases).

Execution of the construct \text{RETURN WITH ERROR Failure}[2] releases the monitor lock and destroys the frame of \text{Proc} before propagation of the signal begins. Note that the argument list in this construct is determined by the declaration of \text{Failure} (not by \text{Proc}'s \text{RETURNS} clause). The catch phrase for \text{UNWIND} is not executed in this case. The signal \text{Failure} is actually raised by the system, after which \text{Failure} propagates as an ordinary error (beginning with \text{Proc}'s caller).

When the \text{RETURN WITH ERROR} construct is used from within an internal procedure, the monitor lock is \text{not} released; \text{RETURN WITH ERROR} will release the monitor lock in precisely those cases that \text{RETURN} will.

Another important issue regarding signals is the handling of \text{UNWINDS}; any entry procedure that may experience an \text{UNWIND} must catch it and clean up the monitor data (restore the monitor invariant):

\[
\begin{align*}
P: & \text{ ENTRY PROCEDURE } [. . .] = \\
& \text{ BEGIN } \text{ ENABLE UNWIND } \Rightarrow \text{ BEGIN } \langle \text{restore invariant}\rangle \text{ END;} \\
& \hspace{1cm} \ldots \\
& \text{ END;}
\end{align*}
\]

At the end of the \text{UNWIND} catchphrase, the compiler will append code to release the monitor lock before the frame is unwound. It is important to note that a monitor always has at least one cleanup task to perform when catching an \text{UNWIND} signal: the monitor lock must be released. To this end, the programmer should be sure to place an enable-clause on the body of every entry procedure that might evoke an \text{UNWIND} (directly or indirectly). If the monitor invariant is already satisfied, no further cleanup need be specified, but \text{the null catch-phrase must be written} so that the compiler will generate the code to unlock the monitor:

\[
\begin{align*}
& \text{ BEGIN } \text{ ENABLE UNWIND } \Rightarrow \text{ NULL;}
\end{align*}
\]

This should be omitted only when it is certain that no \text{UNWINDS} can occur.

Another point is that signals caught by the \text{OptCatchPhrase} of a \text{WAIT} operation should be thought of as occurring after reacquisition of the monitor lock. Thus, like all other monitor code,
catch phrases within a monitor are always executed with the monitor lock held.

10.6. Initialization

When a new monitor comes into existence, its monitor data will generally need to be set to some appropriate initial values; in particular, the monitor lock and any condition variables must be initialized. As usual, Mesa takes responsibility for initializing the simple common cases; for the cases not handled automatically, it is the responsibility of the programmer to provide appropriate initialization code, and to arrange that it be executed at the proper time. The two types of initialization apply in the following situations:

Monitor data in global variables can be initialized using the normal Mesa initial value constructs in declarations. Monitor locks and condition variables in the global frame will also be initialized automatically (although in this case, the programmer does not write any explicit initial value in the declaration).

Monitor data in records must be initialized by the programmer. System procedures must be used to initialize the monitor lock and condition variables. See the runtime documentation for the descriptions of appropriate procedures.

A fine point:

If a variable containing a record is declared in a frame, it is normally possible to initialize it in the declaration (i.e. using a constructor as the initial value); however, this does not apply if the record contains monitor locks or condition variables, which must be initialized via calls to system procedures.

Since initialization code modifies the monitor data, it must have exclusive access to it. The programmer should insure this by arranging that the monitor not be called by its client processes until it is ready for use.
APPENDIX A. Pronouncing Mesa

The following suggestions may be helpful in reading Mesa programs:

For Read
=> chooses
+ gets
n: T n is a T
m.field m’s field
pt p’s referent
@x address of x
[a..b] (the interval) a through b
[a..b] (the interval) a up to b
(a..b] (the interval) above a through b
(a..b) (the interval) above a up to b
FOR i += j, k ... for i getting first j, thereafter k ...
/\x, y, z\ /\ f of x, y and z
! enabling

We leave as an exercise for the reader the following statement, attributed to Oscar Hammerstein II.

i += weary AND Sick[trying];
APPENDIX B. Programming Conventions

The Mesa compiler only uses blanks, tabs, and carriage returns as separators for basic lexical units such as identifiers; extra ones do not hurt. Furthermore, it allows you to write identifiers in any combination of upper and lower case letters: the identifiers Alpha, ALPHA, alpha and AlphA are all legal (but different) Mesa identifiers. It is recommended that you adhere to a standard set of conventions for constructing identifiers and laying out programs. The recommended conventions are summarized below.

B.1. Names

Most identifiers should be written in lower case, except that the first letter of each new "word" in the identifier should be capitalized. Thus,

```
line
firstLine
firstLinePos
```

This convention makes it easy to read identifiers which are made up of several words. (Note that Mesa does not allow spaces in identifiers.)

Capitalize the first letter of type identifiers, procedure names, signal names, and module names.

The following convention for constructing names has been used successfully to reflect their types:

Choose a short (2-3 character) tag for each "basic type" you use: e.g., In for Line and co for Coordinate. You can use the tag as the type name, or not, as you prefer. If you do, capitalize it.

Use the following prefixes to construct tags for "derived" types (most of them reflect the intended use of some underlying type):

```
p - pointer; pLn = pointer to a line
i - index; iLn = index in an array of lines.
l - length
n - number of items (total or count)
```

Whether to use a prefix or to invent a new type tag, is a matter of judgment: depending on whether it is better to emphasize the relationship of this type to another, or to emphasize its individuality.

If you need only one name of a given type in a scope, use the tag as its name:

```
ln : Ln;
pLn : POINTER TO Ln.
```

If you need several names, append modifiers to the tag (avoid simple numbers like 1, 2, etc.):

```
inOld, inNew, inBuffer: Ln.
```

The advantages of this scheme are three-fold:

- the reader spends less time looking up the types of identifiers;
- the writer spends less time thinking up names;
- if you have forgotten a name, there is a good chance you will be able to guess it correctly if you know the tag vocabulary.

B.2. Layout

Write statements one per line, unless several simple statements which together perform a single function will fit on one line.

Indent the labels of a SELECT (including the ENDCASE) one level, and the statements a second level (unless a statement will fit on the same line with the label).
Indent one level for the statement following a THEN or ELSE (unless it fits on the same line). Put THEN on the same line with IF, and don't indent ELSE with respect to IF. If the statement following ELSE is another IF, write both on the same line.

Indent one level for each compound BEGIN-END, DO-ENDLOOP, or bracket pair in a record declaration.

When the rules for IF and SELECT call for indenting a statement, do not indent an extra level for a BEGIN.

It is fine to put a compound statement or loop on a single line if it will fit.

If a statement won't fit on a single line, indent the continuation line(s) by two spaces.

Among other things, these rules have the property that they allow a program to be easily converted to a form in which the bracketing is implied by the indentation.

B.3. Spaces

The following rules for spaces should be broken when necessary, but are a good general guide:

A space after a comma, semicolon, or colon, and none before

No spaces inside brackets or parentheses

No spaces around single-character operations: *, -, etc., except for +.
APPENDIX C. Alto/Mesa Machine Dependencies

This appendix contains a number of machine-dependent constants and definitions for the Alto implementation of Mesa.

C.1. Numeric limits

On the Alto, the numeric limits are the following:

- \text{FIRST}[\text{INTEGER}] = 32768 = 2^{15}\text{ and has internal representation } 100000B
- \text{LAST}[\text{INTEGER}] = 32767 = 2^{15}1\text{ and has internal representation } 077777B
- \text{LAST}[\text{CARDINAL}] = 65535 = 2^{16}1\text{ and has internal representation } 177777B
- \text{FIRST}[\text{LONG INTEGER}] = -2147483648 = 2^{31}
- \text{LAST}[\text{LONG INTEGER}] = 2147483647 = 2^{31}1
- \text{LAST}[\text{LONG CARDINAL}] = 4294967295 = 2^{32}1

C.2. AltoDefs

A module similar to the one below is a part of the Alto/Mesa system and defines several useful constants.

```
AltoDefs: DEFINITIONS =
BEGIN
  wordlength: INTEGER = 16; -- Alto word length (bits)
  maxword: CARDINAL = 177777B; -- N.B. negative as 16 bit integer
  maxinteger: INTEGER = 077777B; -- maximum positive number
  charlength: INTEGER = 8; -- Alto character size (bits)
  maxcharcode: INTEGER = 377B;
  BYTE: TYPE = [0..maxcharcode];
  BytesPerWord, CharsPerWord: INTEGER = wordlength/charlength;
  LogBytesPerWord, LogCharsPerWord: INTEGER = 1;
  PageSize: INTEGER = 256; -- Alto page size (words)
  LogPageSize: INTEGER = 8;
  BytesPerPage,CharsPerPage: INTEGER = PageSize*CharsPerWord;
  LogBytesPerPage, LogCharsPerPage: INTEGER = LogPageSize + LogCharsPerWord;
  VMLimit: CARDINAL = 177777B; -- maximum Alto VM address
  Address: TYPE = [0..VMLimit];
  MaxVMPage: INTEGER = 255; -- maximum Alto VM page number
  MaxFilePage: CARDINAL = 077777B;
  PageNumber: TYPE = [0..MaxFilePage];
  pageCount: TYPE = [0..MaxVMPage + 1];
END.
```
## Appendix C: Alto/Mesa Machine Dependencies

### C.3. ASCII character set and ordering of character values

The following list gives the characters of the ASCII character set in increasing order, accompanied by their literal representations. Control characters are represented as \( \text{tao} \). In addition, a number of special characters such as SP (space), DEL (rubout) are denoted by their generally accepted names.

<table>
<thead>
<tr>
<th>Octal Value</th>
<th>Character Name(s)</th>
<th>Octal Value</th>
<th>Character Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000C</td>
<td>NUL</td>
<td>100C</td>
<td>@</td>
</tr>
<tr>
<td>001C</td>
<td>*A</td>
<td>101C</td>
<td>A</td>
</tr>
<tr>
<td>002C</td>
<td>*B</td>
<td>102C</td>
<td>B</td>
</tr>
<tr>
<td>003C</td>
<td>*C</td>
<td>103C</td>
<td>C</td>
</tr>
<tr>
<td>004C</td>
<td>*D</td>
<td>104C</td>
<td>D</td>
</tr>
<tr>
<td>005C</td>
<td>*E</td>
<td>105C</td>
<td>E</td>
</tr>
<tr>
<td>006C</td>
<td>*F</td>
<td>106C</td>
<td>F</td>
</tr>
<tr>
<td>007C</td>
<td>*G, BELL</td>
<td>107C</td>
<td>G</td>
</tr>
<tr>
<td>010C</td>
<td>*H, BS</td>
<td>110C</td>
<td>H</td>
</tr>
<tr>
<td>011C</td>
<td>*I</td>
<td>111C</td>
<td>I</td>
</tr>
<tr>
<td>012C</td>
<td>*J, LF</td>
<td>112C</td>
<td>J</td>
</tr>
<tr>
<td>013C</td>
<td>*K</td>
<td>113C</td>
<td>K</td>
</tr>
<tr>
<td>014C</td>
<td>*L</td>
<td>114C</td>
<td>L</td>
</tr>
<tr>
<td>015C</td>
<td>*M, CR</td>
<td>115C</td>
<td>M</td>
</tr>
<tr>
<td>016C</td>
<td>*N</td>
<td>116C</td>
<td>N</td>
</tr>
<tr>
<td>017C</td>
<td>*O</td>
<td>117C</td>
<td>O</td>
</tr>
<tr>
<td>020C</td>
<td>*P</td>
<td>120C</td>
<td>P</td>
</tr>
<tr>
<td>021C</td>
<td>*Q</td>
<td>121C</td>
<td>Q</td>
</tr>
<tr>
<td>022C</td>
<td>*R</td>
<td>122C</td>
<td>R</td>
</tr>
<tr>
<td>023C</td>
<td>*S</td>
<td>123C</td>
<td>S</td>
</tr>
<tr>
<td>024C</td>
<td>*T</td>
<td>124C</td>
<td>T</td>
</tr>
<tr>
<td>025C</td>
<td>*U</td>
<td>125C</td>
<td>U</td>
</tr>
<tr>
<td>026C</td>
<td>*V</td>
<td>126C</td>
<td>V</td>
</tr>
<tr>
<td>027C</td>
<td>*W</td>
<td>127C</td>
<td>W</td>
</tr>
<tr>
<td>030C</td>
<td>*X</td>
<td>130C</td>
<td>X</td>
</tr>
<tr>
<td>031C</td>
<td>*Y</td>
<td>131C</td>
<td>Y</td>
</tr>
<tr>
<td>032C</td>
<td>*Z</td>
<td>132C</td>
<td>Z</td>
</tr>
<tr>
<td>033C</td>
<td>ESC</td>
<td>133C</td>
<td></td>
</tr>
<tr>
<td>034C</td>
<td></td>
<td>134C</td>
<td></td>
</tr>
<tr>
<td>035C</td>
<td></td>
<td>135C</td>
<td></td>
</tr>
<tr>
<td>036C</td>
<td></td>
<td>136C</td>
<td></td>
</tr>
<tr>
<td>037C</td>
<td></td>
<td>137C</td>
<td></td>
</tr>
<tr>
<td>040C</td>
<td>SPace</td>
<td>140C</td>
<td></td>
</tr>
<tr>
<td>041C</td>
<td></td>
<td>141C</td>
<td></td>
</tr>
<tr>
<td>042C</td>
<td></td>
<td>142C</td>
<td></td>
</tr>
<tr>
<td>043C</td>
<td></td>
<td>143C</td>
<td></td>
</tr>
<tr>
<td>044C</td>
<td></td>
<td>144C</td>
<td></td>
</tr>
<tr>
<td>045C</td>
<td></td>
<td>145C</td>
<td></td>
</tr>
<tr>
<td>046C</td>
<td></td>
<td>146C</td>
<td></td>
</tr>
<tr>
<td>047C</td>
<td></td>
<td>147C</td>
<td></td>
</tr>
<tr>
<td>050C</td>
<td></td>
<td>150C</td>
<td></td>
</tr>
<tr>
<td>051C</td>
<td></td>
<td>151C</td>
<td></td>
</tr>
<tr>
<td>052C</td>
<td></td>
<td>152C</td>
<td></td>
</tr>
<tr>
<td>053C</td>
<td></td>
<td>153C</td>
<td></td>
</tr>
<tr>
<td>054C</td>
<td></td>
<td>154C</td>
<td></td>
</tr>
<tr>
<td>055C</td>
<td></td>
<td>155C</td>
<td></td>
</tr>
<tr>
<td>056C</td>
<td></td>
<td>156C</td>
<td></td>
</tr>
<tr>
<td>057C</td>
<td></td>
<td>157C</td>
<td></td>
</tr>
<tr>
<td>060C</td>
<td></td>
<td>160C</td>
<td></td>
</tr>
<tr>
<td>061C</td>
<td></td>
<td>161C</td>
<td></td>
</tr>
<tr>
<td>062C</td>
<td></td>
<td>162C</td>
<td></td>
</tr>
<tr>
<td>063C</td>
<td></td>
<td>163C</td>
<td></td>
</tr>
<tr>
<td>064C</td>
<td></td>
<td>164C</td>
<td></td>
</tr>
<tr>
<td>065C</td>
<td></td>
<td>165C</td>
<td></td>
</tr>
<tr>
<td>066C</td>
<td></td>
<td>166C</td>
<td></td>
</tr>
<tr>
<td>067C</td>
<td></td>
<td>167C</td>
<td></td>
</tr>
<tr>
<td>070C</td>
<td></td>
<td>170C</td>
<td></td>
</tr>
<tr>
<td>071C</td>
<td></td>
<td>171C</td>
<td></td>
</tr>
<tr>
<td>072C</td>
<td></td>
<td>172C</td>
<td></td>
</tr>
<tr>
<td>073C</td>
<td></td>
<td>173C</td>
<td></td>
</tr>
<tr>
<td>074C</td>
<td></td>
<td>174C</td>
<td></td>
</tr>
<tr>
<td>075C</td>
<td></td>
<td>175C</td>
<td></td>
</tr>
<tr>
<td>076C</td>
<td></td>
<td>176C</td>
<td></td>
</tr>
<tr>
<td>077C</td>
<td></td>
<td>177C</td>
<td>DEL</td>
</tr>
</tbody>
</table>
C.4. **Alto/Mesa STRING procedures**

A module similar to the one below is a part of the Alto/Mesa system and defines useful procedures provided by the system for operating on strings. See the system documentation for its exact form.

```
DIRECTORY AltoDefs: FROM "altodefs";

STRINGDefs: DEFINITIONS =
BEGIN
  -- COMPIL•TIME CONSTANTS AND TYPES
  SubStringDescriptor: TYPE = RECORD [ 
    base: STRING,
    offset, length: CARDINAL];
  SubString: TYPE = POINTER TO SubStringDescriptor;

  -- INTERFACE ITEMS
  Overflow: SIGNAL;
  InvalidNumber: SIGNAL;
  StringBoundsFault: SIGNAL [s: STRING] RETURNS [ns: STRING];
  WordsForString: PROCEDURE [nchars: CARDINAL] RETURNS [CARDINAL];
  AppendChar: PROCEDURE [s: STRING, c: CHARACTER];
  AppendString: PROCEDURE [to, from: STRING];
  Equa/String, Equa/Strings: PROCEDURE [s1, s2: STRING] RETURNS [BOOLEAN];
  EquivalentString, EquivalentStrings: PROCEDURE [s1, s2: STRING] RETURNS [BOOLEAN];
  AppendSubString: PROCEDURE [to: STRING, from: SubString];
  Equa/SubString, Equa/SubStrings: PROCEDURE [s1, s2: SubString] RETURNS [BOOLEAN];
  EquivalentSubString, EquivalentSubStrings: PROCEDURE [s1, s2: SubString] RETURNS [BOOLEAN];
  DeleteSubString: PROCEDURE [s: SubString];
  UpperCase, LowerCase: PROCEDURE [CHARACTER] RETURNS [CHARACTER];
  StringToDecimal: PROCEDURE [STRING] RETURNS [INTEGER];
  StringToOctal: PROCEDURE [STRING] RETURNS [UNSPECIFIED];
  StringToLongNumber: PROCEDURE [s: STRING, radix: CARDINAL] RETURNS [LONG UNSPECIFIED];
  AppendDecimal: PROCEDURE [s: STRING, n: INTEGER];
  AppendOctal: PROCEDURE [s: STRING, n: UNSPECIFIED];
  AppendNumber: PROCEDURE [s: STRING, n: UNSPECIFIED, radix: CARDINAL];
  AppendLongDecimal: PROCEDURE [s: STRING, n: LONG INTEGER];
  AppendLongNumber: PROCEDURE [s: STRING, n: LONG UNSPECIFIED, radix: CARDINAL];
END.
```
APPENDIX D. Binder Extensions

The Alto implementation of the Mesa binder provides two extensions for controlling the space occupied by Mesa programs at runtime. These are specified with the \texttt{CPacking} and \texttt{CLinks} clauses (section 7.7).

\textbf{D.1. Code packing}

It is possible to pack together the code for several modules into a single segment. This is useful for two reasons:

- Since the code is allocated an integral number of pages, there is some wasted space in the last page ("breakage"). If several modules are combined into a single segment, the breakage is amortized over all the modules, and there is less waste on the average.
- All the modules will be brought into and out of memory together, as a unit; a reference to any module in the pack will cause all the code to be brought in. Modules which are tightly coupled dynamically are good candidates for packing (for example, resident code should probably always be packed).

Of course, it is possible to "over pack" a configuration; the segments might become so large that there will never be room in memory for more than one of them at a time (this should remind you of an overlay system). \textit{Packing is a tradeoff, and should be used with caution.}

\textbf{D.1.1. Syntax}

The segments are specified at the beginning of the configuration by giving a list of the modules which comprise each one. Any number of \texttt{PACK} statements may appear. The scope of the packing specification is the whole configuration, and not subconfigurations or individual module instances, because there is at most one copy of a module's code in any configuration.

\[
\text{ConfigDescription} ::= \text{Directory CPacking Configuration} .
\]
\[
\text{CPacking} ::= \text{empty} | \text{CPackSeries} ;
\]
\[
\text{CPackList} ::= \text{PACK IdList}
\]
\[
\text{CPackSeries} ::= \text{CPackList} | \text{CPackSeries} ; \text{CPackList}
\]

Each \texttt{PackList} defines a single segment; the code for all the modules in the \texttt{IdList} will be packed into it. The identifiers in the \texttt{IdList} must refer to modules in the configuration, and not to module instances; it is the code and not the global frames that are being packed (the frames are always packed when they are allocated by the loader).

It is illegal to specify the same module in more than one \texttt{PackList}. Even though there may be multiple instances of the module (i.e., multiple global frames) in the configuration, the code is shared by all of them, and therefore can only appear in one pack.

Finally, it is perfectly fine to reach inside a previously bound configuration that is being instantiated and single out some or all of its modules for packing. Of course, you must know something about the structure of that configuration in order to do this.
D.1.2. Restrictions

Obviously, the PACK statements apply only if the code is being moved to the output file; otherwise, the pack lists are ignored (and no warning message is given). This allows the programmer to debug the configuration without shuffling the code from file to file, thereby saving time. When making the final version, the packing can be effected with a binder switch, without having to modify the source of the configuration description.

Once some modules have been packed together, they cannot be taken apart and repacked with other modules later on, when they are bound into some other configuration.

Fine point:

If a previously bound configuration contains a pack, referencing any module of the pack gets the whole thing. So it is possible to pack a module and a pack together, or even to pack two packs. It is never possible to unpack a pack.

In general, code packing should be specified only to the extent that no unpacking will ever be desired. Once the packing is done, it can't be undone, unless you start over with the individual modules.

D.2. External links

In previous Mesa systems, links to the externals referenced by a program (imported procedures, signals, errors, frames, and programs) were always stored in the module’s global frame. This allows each instance of a module to be bound differently, and it allows binding to be done at runtime without modification of the module’s code segment. However, it has two drawbacks:

The links are only referenced by the module’s code, and are therefore not needed when the code is swapped out. Hence, the links logically belong in the code segment.

If two instances of a module are bound identically (the usual case), the links must be stored twice.

Fine Point:

To determine the amount of space required for external links, see the compiler’s typescript file. Each link occupies one word.

The Mesa binder optionally places links in the code segment. This option is enabled by constructs in the configuration language, and is further controlled by binder and loader switches.

D.2.1. Syntax

For each component of a configuration, the link location is specified using the LINKS construct defined below. The default is frame links.

\[ \text{CLinks} ::= \text{empty} \mid \text{LINKS} : \text{CODE} \mid \text{LINKS} : \text{FRAME} \]

A link specification can optionally be attached to each instantiation of a module, overriding the current default, so that the link location can be different for each instance.

\[ \text{CRightSide} ::= \text{Item} \mid \text{Item} [ \text{CLinks} ] \mid \text{Item} [ \text{IdList} ] \text{CLinks} \]
Alternately, the link option can be specified in the configuration header. This merely changes the default option for the configuration; it will apply to all components (including nested configurations) unless it is explicitly overridden.

\[
\text{CHead} \quad ::= \quad \text{CONFIGURATION} \ \text{CLinks} \ \text{Imports} \ \text{CExports} \ \text{ControlClause}
\]

This construction works much like the \text{PUBLIC} / \text{PRIVATE} options in Mesa, and it nests in the same way. A link option attached to a configuration changes the default for all components within it, but that default can be overridden for a particular module (or nested configuration) by specifying a different link option.

\subsection{Restrictions}

This scheme has the consequence that, \textit{if a module with code links has multiple instances, each instance must be bound the same.}

As with code packing, the code links option takes effect only when the code is being moved to the output file. At this point, the binder will make room for the links as it copies the code if any module sharing that code has requested code links. Again, this allows a programmer to debug without the expense of moving the code (using frame links), and then to effect the code links option with a binder switch, without changing the source of the configuration description.

\textbf{Fine point:}

Once space for code links has been added to a configuration, it cannot be undone by a later binding. On the other hand, space for code links can always be added to a (previously bound) configuration, even if it did not specify code links in its description.

Using code links has one drawback: it slows down the binding and loading process, as the code must be swapped in and rewritten. The binder must make room in the code segment for the links, as described above. And because the loader resolves imports of previously loaded modules, as well as the imports of the module being loaded, it may have to swap in (and perhaps update and swapout) the code segment for every module in the system.

Because of the overhead involved, the loader will not automatically attempt to use code links, even if the space is available in the code segment. A loader switch must be used to effect this option.

Documentation of binder and loader switches in in the \textit{Mesa User's Handbook}. 

APPENDIX E. Mesa Reserved Words

Listed below are all of the Mesa reserved words. Words marked with an astrisk are predeclared rather than reserved. Predeclared identifiers can be redefined (but seldom should be).

<table>
<thead>
<tr>
<th>Predeclared</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>MOD</td>
</tr>
<tr>
<td>ALL</td>
<td>MONITOR</td>
</tr>
<tr>
<td>AND</td>
<td>MONITORED</td>
</tr>
<tr>
<td>ANY</td>
<td>MONITORLOCK*</td>
</tr>
<tr>
<td>ARRAY</td>
<td>NEW</td>
</tr>
<tr>
<td>BASE</td>
<td>NIL*</td>
</tr>
<tr>
<td>BEGIN</td>
<td>NOT</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>NOTIFY</td>
</tr>
<tr>
<td>BROADCAST</td>
<td>NULL</td>
</tr>
<tr>
<td>CARDINAL</td>
<td>OF</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>OPEN</td>
</tr>
<tr>
<td>CODE</td>
<td>OR</td>
</tr>
<tr>
<td>COMPUTED</td>
<td>ORDERED</td>
</tr>
<tr>
<td>CONDITION*</td>
<td>OVERLAID</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>PACKED</td>
</tr>
<tr>
<td>DECREASING</td>
<td>POINTER</td>
</tr>
<tr>
<td>DEFINITIONS</td>
<td>PORT</td>
</tr>
<tr>
<td>DEPENDENT</td>
<td>PRIVATE</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>PROCEDURE</td>
</tr>
<tr>
<td>DIRECTORY</td>
<td>PROCESS</td>
</tr>
<tr>
<td>DO</td>
<td>DO</td>
</tr>
<tr>
<td>ELSE</td>
<td>PUBLIC</td>
</tr>
<tr>
<td>ENABLE</td>
<td>READONLY</td>
</tr>
<tr>
<td>END</td>
<td>REAL*</td>
</tr>
<tr>
<td>ENDCASE</td>
<td>RECORD</td>
</tr>
<tr>
<td>ENDLOOP</td>
<td>RELATIVE</td>
</tr>
<tr>
<td>ENTRY</td>
<td>REPEAT</td>
</tr>
<tr>
<td>ERROR</td>
<td>RESTART</td>
</tr>
<tr>
<td>EXIT</td>
<td>RESUME</td>
</tr>
<tr>
<td>EXITS</td>
<td>RETRY</td>
</tr>
<tr>
<td>EXPORTS</td>
<td>RETURN</td>
</tr>
<tr>
<td>FALSE*</td>
<td>RETURNS</td>
</tr>
<tr>
<td>FINISHED.</td>
<td>SELECT</td>
</tr>
<tr>
<td>FIRST</td>
<td>SHARES</td>
</tr>
<tr>
<td>FOR</td>
<td>SIGNAL</td>
</tr>
<tr>
<td>FORK</td>
<td>SIZE</td>
</tr>
<tr>
<td>FRAME</td>
<td>START</td>
</tr>
<tr>
<td>FROM</td>
<td>STATE</td>
</tr>
<tr>
<td>GO</td>
<td>STOP</td>
</tr>
<tr>
<td>GOTO</td>
<td>STRING</td>
</tr>
<tr>
<td>IF</td>
<td>StringBody*</td>
</tr>
<tr>
<td>IMPORTS</td>
<td>THEN</td>
</tr>
<tr>
<td>IN</td>
<td>THROUGH</td>
</tr>
<tr>
<td>INLINE</td>
<td>TO</td>
</tr>
<tr>
<td>INTEGER</td>
<td>TRANSFER</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>TRUE*</td>
</tr>
<tr>
<td>JOIN</td>
<td>TYPE</td>
</tr>
<tr>
<td>LAST</td>
<td>UNSPECIFIED*</td>
</tr>
<tr>
<td>LENGTH</td>
<td>UNTIL</td>
</tr>
<tr>
<td>LOCKS</td>
<td>UNWIND*</td>
</tr>
<tr>
<td>LONG</td>
<td>USING</td>
</tr>
<tr>
<td>LOOP</td>
<td>WAIT</td>
</tr>
<tr>
<td>LOOPHOLE</td>
<td>WHILE</td>
</tr>
<tr>
<td>MACHINE</td>
<td>WITH</td>
</tr>
<tr>
<td>MAX</td>
<td>WORD*</td>
</tr>
<tr>
<td>MIN</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX F. Collected Grammar

The Mesa grammar in this section is a collected version of the grammar distributed throughout the body of the Manual. There are some differences, primarily due to the Manual's grammar being distorted for purposes of exposition. This one is intended to be internally consistent.

The grammar is divided into four parts, corresponding to the syntax for CompilationUnit, TypeSpecification, Statement, and Expression. These four parts refer to each other and occasionally use syntax rules from other parts (such as LeftSide, which is used in an assignment statement but defined under Expression). Where such cross references occur, a comment has been added to indicate which part to refer to. Other than this, each part is self-contained, and the productions within a part have been ordered alphabetically by their names, except that the productions for CompilationUnit, TypeSpecification, etc. head their respective sections.

**CompilationUnit ::=:**

```
Directory
Directory ::= empty | DIRECTORY IncludeList ;
ExportsList ::= empty | EXPORTS IdList
FileName ::= stringLiteral
GlobalAccess ::= Access -- in TypeSpecification
IdList ::= identifier | IdList , identifier
ImportsList ::= empty | IMPORTS InterfaceList
IncludeList ::= identifier : FROM FileName UsingClause |
               IncludeList , identifier : FROM FileName UsingClause
InterfaceItem ::= identifier | identifier : identifier
InterfaceList ::= InterfaceItem | InterfaceList , InterfaceItem
ModuleBody ::= Block . -- in Statement
ModuleHead ::= ProgramTC ImportsList ExportsList ShareList |
               DefinitionTC ImportsList ShareList
ModuleParams ::= empty | [ NamedFieldList ] -- in TypeSpecification
ShareList ::= empty | SHARES IdList
UsingClause ::= empty | USING [ IdList ]
```

**TypeSpecification ::=:**

```
Access ::= empty | PUBLIC | PRIVATE
Adjective ::= identifier
ArrayTC ::= PackingOption ARRAY IndexType OF TypeSpecification
BaseOption ::= empty | BASE
ByteList ::= Expression | ByteList , Expression
CommonPart ::= empty | NamedFieldList ,
ConstantList ::= Expression | ConstantList , Expression
DefaultOption ::= empty | + DefaultSpecification
DefaultSpecification ::= empty |
                        Expression |
                        NULL |
                        Expression | NULL
DefinitionTC ::= DEFINITIONS |
               DEFINITIONS LocksClause
```
Appendix F: Collected Grammar

DescriptorTC ::=
  DESCRIPTOR FOR ReadOnlyOption TypeSpecification |
  DESCRIPTOR FOR ReadWriteOption TypeSpecification

elementType ::=
  INTEGER | CARDINAL | CHARACTER | BOOLEAN |
  EnumerationTC | SubrangeTC

enumerationTC ::=
  { IdList }

fieldList ::=
  [ UnnamedFieldList ] [[ NamedFieldList ]

indexType ::=
  ElementType | Typeldentifier

inlineOption ::=
  empty | INLINE

instructionSeries ::=
  empty | ByteList | ByteList ; InstructionSeries

interval ::=
  [ Expression . Expression ]
  [ Expression . Expression ]
  ( Expression . Expression )
  ( Expression . Expression )

LocksClause ::=
  empty |
  LOCKS Expression |
  LOCKS Expression USING identifier : TypeSpecification

longTC ::=
  LONG TypeSpecification

MachineCode ::=
  MACHINE CODE BEGIN InstructionSeries END -- not described in this manual

MachineDependent ::=
  empty | MACHINE DEPENDENT

MonitoredOption ::=
  empty | MONITORED

NamedFieldList ::=
  IdList : Access TypeSpecification DefaultOption |
  IdList : Access TypeSpecification DefaultOption

OptionalInterval ::=
  empty | Interval

Ordered ::=
  empty | ORDERED

PackingOption ::=
  empty | PACKED

ParameterList ::=
  empty | FieldList

PointerTail ::=
  empty |
  TO ReadOnlyOption TypeSpecification |
  TO FRAME [ identifier ]

PortTC ::=
  Ordered BaseOption POINTER OptionalInterval PointerTail

PredefinedType ::=
  INTEGER | CARDINAL | LONG INTEGER |
  REAL | BOOLEAN | CHARACTER | STRING |
  UNSPECIFIED | WORD

ProcedureBody ::=
  InlineOption Block -- Block in Statement

ProcedureTC ::=
  PROCEDURE ParameterList ReturnsClause |
  PROCEDURE ParameterList ReturnsClause

ProcessTC ::=
  PROCESS ReturnsClause |
  PROCESS ReturnsClause

ProgramTC ::=
  PROGRAM ParameterList ReturnsClause |
  PROGRAM ParameterList ReturnsClause

ReadOnlyOption ::=
  empty | READONLY

RecordTC ::=
  MonitoredOption MachineDependent RECORD [ VariantFieldList ]

RelativeTC ::=
  Typeldentifier RELATIVE TypeSpecification |
  Typeldentifier RELATIVE LONG TypeSpecification

ReturnsClause ::=
  empty | RETURNS FieldList

SignalOrError ::=
  SIGNAL | ERROR

SignalTC ::=
  SignalOrError ParameterList ReturnsClause

SubrangeTC ::=
  Interval | Typeldentifier Interval

Tag ::=
  identifier : Access TagType |
  COMPUTED TagType |
  OVERLAID TagType

TagType ::=
  TypeSpecification *

TypeConstructor ::=
  DescriptorTC | ArrayTC | EnumerationTC | LongTC |
  PointerTC | PortTC | ProcedureTC | ProcessTC |
  RecordTC | RelativeTC | SignalTC | SubrangeTC

Typeldentifier ::=
  identifier |
  identifier . identifier
Appendix F: Collected Grammar

UnnamedFieldList ::= Adjective TypeIdentifier
UnnamedFieldList , TypeSpecification

Variant ::= IdList => [ VariantFieldList ], |
IdList => NULL ,

VariantFieldList ::= CommonPart identifier : Access VariantPart |
VariantPart |
NamedFieldList |
UnnamedFieldList |
empty

VariantList ::= Variant | VariantList Variant

VariantPart ::= SELECT Tag FROM
VariantList
ENDCASE

Statement ::= AssignmentStmt | Block | Call |
ContinueStmt | ExitStmt | GotoStmt | IfStmt |
JoinCall | LoopCloseStmt | -- JoinCall in Expression
LoopStmt | Notify | [ NullStmt |
ResumeStmt | RetryStmt | ReturnStmt | SelectStmt |
SignalCall | StartCall | RestartStmt |
StopStmt | WaitStmt

AdjectiveList ::= Adjective | AdjectiveList , Adjective -- in TypeSpecification

Assignment ::= FOR identifier = Expression , Expression

AssignmentStmt ::= LeftSide + RightSide |
 -- LeftSide, RightSide in Expression
Extractor + RightSide

Block ::= BEGIN
OpenClause
EnableClause
DeclarationSeries
StatementSeries
ExitsClause
END

Call ::= Variable | -- in Expression
Variable [ ComponentList ] | -- ComponentList in Expression
Variable [ ComponentList | CatchSeries ]
Variable [ CatchSeries ]

Catch ::= ExpressionList => Statement | -- ExpressionList in Expression
CatchItem ::= Catch | ANY => Statement
CatchSeries ::= CatchItem | Catch ; CatchSeries
ChoiceSeries ::= AdjectiveList => Statement ; |
ChoiceSeries AdjectiveList => Statement ; |

CompoundStmt ::= BEGIN
Body
ExitsClause
END

ConditionTest ::= empty | WHILE Expression | UNTIL Expression

ContinueStmt ::= CONTINUE

Declaration ::= IdList ;
 -- Access in TypeSpecification
ReadOnlyOption EntryOption
 -- ReadOnlyOption in TypeSpecification
TypeSpecification
Initialization ; |
IdList : Access TYPE = Access TypeSpecification ;

DeclarationSeries ::= empty | DeclarationSeries Declaration

Direction ::= empty | DECREASING

ElseClause ::= empty | ELSE Statement

EnableClause ::= enable | ENABLE CatchItem ; |
ENABLE BEGIN CatchSeries END ; |
Appendix F: Collected Grammar

ENABLE BEGIN CatchSeries ; END ;
empty
EntryOption ::= empty | ENTRY
ErrorCall ::= ERROR Call | ERROR
ExitsClause ::= empty | EXITS | EXITS ExitSeries | EXITS ExitSeries ;
ExitSeries ::= LabelList => Statement |
ExitSeries ; LabelList => Statement
ExitStmt ::= EXIT
ExtractItem ::= empty | LeftSide
Extractor ::= [ KeywordExtractList ]
[ PositionalExtractList ]
FinalStmtChoice ::= empty | => Statement
FinishedExit ::= FINISHED => Statement |
FINISHED => Statement ;
GotoStmt ::= GOTO Label | GO TO Label
IfStmt ::= IF Expression THEN Statement ElseClause
InitExpr ::= Expression |
ProcedureBody | in TypeSpecification
MachineCode | in TypeSpecification |
[ Expression ] | for STRING initialization |
CODE | for SIGNAL initialization
Initialization ::= empty | + InitExpr | InitExpr
Iteration ::= FOR identifier Direction IN LoopRange
IterativeControl ::= empty | Repetition | Iteration | Assignment
KeywordExtract ::= identifier : ExtractItem
KeywordExtractList ::= KeywordExtract |
KeywordExtractList , KeywordExtract
Label ::= identifier
LabelList ::= Label | LabelList , Label
LeftItem ::= Expression
LoopCloseStmt ::= LOOP
LoopControl ::= IterativeControl ConditionTest
LoopExits ::= ExitSeries | ExitSeries ; | FinishedExit | ExitSeries ; FinishedExit
LoopExitsClause ::= empty | REPEAT LoopExits
LoopRange ::= SubrangeTC | Typeldentifier BOOLEAN | CHARACTER
LoopStmt ::= LoopControl
DO
OpenClause
DeclarationSeries
EnableClause
StatementSeries
LoopExitsClause
ENDLOOP
NotifyStmt ::= NOTIFY Variable | BROADCAST Variable
NullStmt ::= NULL
OpenClause ::= empty | OPEN OpenList ;
OpenItem ::= Expression | identifier : Expression
OpenList ::= OpenItem | OpenItem , OpenItem
OptCatchPhrase ::= empty | [ ! CatchSeries ]
PositionalExtractList ::= ExtractItem |
PositionalExtractList , ExtractItem
Repetition ::= THROUGH Subrange -- in Expression
RestartStmt ::= RESTART Variable OptCatchPhrase -- Variable in Expression
ResumeStmt ::= RESUME |
RESUME [ ComponentList ] -- ComponentList in Expression
RetryStmt ::= RETRY
ReturnStmt ::= RETURN |
Appendix F: Collected Grammar

RETURN [ ComponentList ]  -- ComponentList in Expression
RETURN WITH ERROR Call

SelectStmt ::= Select | SelectVariant
Select ::= SELECT LeftItem FROM StmtChoiceSeries
          ENDCASE FinalStmtChoice
SelectVariant ::= WITH OpenItem SELECT TagItem FROM ChoiceSeries
                ENDCASE FinalStmtChoice
SignalCall ::= SIGNAL Call | ErrorCall
StartCall ::= START Call
StatementSeries ::= empty | Statement |
                Statement ; StatementSeries
StmtChoiceSeries ::= TestList => Statement ;
                    StmtChoiceSeries TestList => Statement ;
StopStmt ::= STOP OptCatchPhrase
TagItem ::= empty | Expression
Test ::= Expression | RelationTail -- RelationTail in Expression
TestList ::= Test | TestList , Test
WaitStmt ::= WAIT Variable OptCatchPhrase

Expression ::= AssignmentExpr | Disjunction | ForkCall | IfExpr |
              JoinCall | NewExpr | SelectExpr |
              SignalCall | --SignalCall in Statement
              StartCall | --StartCall in Statement
AddingOp ::= + | -
AssignmentExpr ::= LeftSide + RightSide
BuiltinCall ::= MIN [ ExpressionList ] | MAX [ ExpressionList ] |
               ABS [ Expression ] | LENGTH [ Expression ] |
               BASE [ Expression ] |
               TypeOp [ TypeSpecification ] |
               DESCRIPTOR [ Expression ] |
               DESCRIPTOR [ Expression , Expression ] |
               DESCRIPTOR [ Expression , Expression , TypeSpecification ] |
ChoiceList ::= AdjectiveList => Expression , |
              -- AdjectiveList in Statement
              ChoiceList AdjectiveList => Expression ,
Component ::= empty | Expression | NULL
ComponentList ::= KeywordComponentList | PositionalComponentList
Conjunction ::= Negation | Conjunction AND Negation
Constructor ::= OptionalTypeId [ ComponentList ]
Disjunction ::= Conjunction | Disjunction OR Conjunction
ExprChoiceList ::= TestList => Expression , |
                  -- TestList in Statement
ExprChoiceList TestList => Expression ,
ExpressionList ::= Expression => Expression , |
                 -- TestList in Statement
ExpressionList Expression |
Factor ::= Primary | Primary
ForkCall ::= FORK Call
FunctionCall ::= BuiltinCall | Call -- Call in Statement
IfExpr ::= IF Expression THEN Expression ELSE Expression
IndexedAccess ::= ( Expression ) | Expression |
                Variable [ Expression ] |
IndirectAccess ::= ( Expression ) | Variable |
JoinCall ::= JOIN Call
KeywordComponent ::= identifier : Component
KeywordComponentList ::= KeywordComponent |
                        KeywordComponentList , KeywordComponent
Appendix F: Collected Grammar

LeftSide ::= identifier | Call | -- Call in Statement
IndexedAccess | QualifiedAccess | IndirectAccess |
LOOPHOLE [ Expression ]
LOOPHOLE [ Expression, TypeSpecification ]

Literal ::= numericLiteral | -- all defined outside the grammar
stringLiteral |
characterLiteral

MultiplyingOp ::= * | / | MOD

Negation ::= Relation | Not Relation

NewExpr ::= NEW Variable OptCatchPhrase

Not ::= ~ | NOT

OptionalTypeid ::= empty | TypeIdentifier -- in TypeSpecification

PositionalComponentList ::= Component |
                           PositionalComponentList , Component

Primary ::= Variable | Literal | ( Expression ) | FunctionCall |
            Constructor | ALL [ Expression ] | @ LeftSide | identifier [ Expression ]

Product ::= Factor | Product MultiplyingOp Factor

QualifiedAccess ::= ( Expression ) . identifier | Variable . identifier

Relation ::= Sum | Sum RelationTail

RelationalOp ::= # | = | < | <= | > | >=

RelationTail ::= RelationalOp Sum | Not RelationalOp Sum |
                IN SubRange | Not IN Subrange

RightSide ::= Expression

SelectExpr ::= SelectExprSimple | SelectExprVariant

SelectExprSimple ::= SELECT LeftItem FROM -- LeftItem in Statement
                  ExprChoiceList
                  END_CASE = > Expression

SelectExprVariant ::= WITH OpenItem SELECT TagItem FROM -- OpenItem, TagItem in Statement
                  ExprChoiceList
                  ENDCASE = > Expression

Subrange ::= SubrangeTC | -- in TypeSpecification
            TypeIdentifier | -- in TypeSpecification

Sum ::= Product | Sum AddingOp Product

TypeOp ::= SIZE | FIRST | LAST

Variable ::= LeftSide
INDEX

In this index, bold face page numbers indicate where the primary, defining information can be found; plain page numbers designate further examples.

137, 138, 140
"95
* 12, 94, 95
+ 12
- 12
-- 2, 5, 6
. 34, 104, 106-7
/ 12
::= 2
; 5-6, 53, 57, 136, 137
=> 54, 56, 94, 96, 136, 138, 140
@ 29, 42, 44, 147
[] 125, 130
+ 42
+ 5, 17, 63, 76, 130
ABS 5, 12
Access 94, 104, 117-20
activation 72, 137
actual procedure 68, 74, 78
actual tag 94, 97, 99
AddingOperator 12
adjective 96
adjectives 92, 97
aggregate type 22
aligned 29, 42, 69
ALL 30
AlternateName 60, 98
ANY 136, 138-9
argument 72, 72
arguments 67, 144
arguments buffered 150
array 22, 27-8
constructor 28, 30
descriptor 85, 87
ARRAY 27-8, 29
Assignation 63, 63
assignment 6, 51
expression 52
AssignmentExpr 11, 17
AssignmentStmt 7, 51
automatic dereferencing 44
B 8
balancing 45, 47
BASE 29, 41, 43-4, 86, 87, 88, 90-1
base type 25
BCD 103, 123
BEGIN 57, 136
Binary Configuration Description 103, 123
binding 101, 103, 123-4, 132
blank 6
block 57
Block 57, 57, 73, 137, 140
BNF 2
BOOLEAN 7-8, 16, 22, 24, 53
bound variant 97
bound variant type 93, 95, 96
bounds 64
BROADCAST 159, 160-1
BuildinCall 12, 12
built-in type 7
C 9
C/Mesa 103, 120, 125
call 67
Call 12, 71, 137, 140
CallStmt 71
CARDINAL 7, 7, 10, 22, 26-7, 49
Catch 134, 136, 137-9
catch phrase 136, 139, 150, 154
CHARACTER 7, 9, 15, 22
characterLiteral 9
client 120
  module 123
CODE 125, 135, 141
coercion 46, 73
colon 6
comma 56, 94
comment 5, 6
common part 92
CommonPart 118
compatible 70
compilation order 110
CompilationUnit 104
compile-time 84, 104, 105
  constant 18, 30, 53
completely bound variant 100
component 27, 31, 125
component type 27
components 97
ComponentType 29
COMPUTED 94, 100
computed tag 93, 95, 97, 100
CONDITION 155, 158, 161
condition variable 155
ConditionTest 62, 63
configuration 123
Configuration 125
CONFIGURATION 125, 126
Configuration Description 103, 123-4
Index

GOTO 57, 58-9, 64, 66, 139
home module 117
identifier 21
constant 23, 94
list 6
identifier 5, 127
IdList 3, 6
if 5, 52, 76
expression 53
statement 52
IfStmt 52
implementing 121
implementor 120
implicit qualification 99
import 101
imports 126
IMPORTS 78, 102-4, 115, 124-5, 126, 127
IN 15, 47, 55, 63
include 105
IncludeList 104
indefinite index type 86
index type 27
indexed reference 27
IndexedReference 29
IndexType 29, 85
inequality 6
inherent
representation 49
type 45
initialization 18, 68, 84, 135
Initialization 19, 33
inline 69, 73, 80
INLINE 80, 113-4
InlineOption 73
instance 115, 124
INTEGER 7, 7, 10, 22, 26-7, 49
interface 101, 120, 125, 135, 158
element 110
record 115, 116, 127
type 115, 116
variable 110, 111-2, 113-4
INTERNAL 157, 165
internal procedure 158
interval 15, 63
Interval 15
iteration 63, 63
IterativeControl 63
JOIN 136, 152-3, 153, 154
jump table 55
keyword 39
constructor 31, 35, 35, 74
extractor 74
name 35
Label 58
Labels 58
LAST 24, 26
LeftSide 7, 29, 38, 42, 45, 51
length 82
LENGTH 29, 86, 87
lengthening 43
lexical units 6
link 124
LinkageFault 148, 149, 150
LINKS 125, 125
literal 18
Literal 12
loader 132
loading 123
local string literal 84
local variable 68
LOCK 161
LOCKS 162, 164, 166
LocksClause 162, 165
LONG 14, 43, 87, 92
CARDINAL 7, 9-10
INTEGER 7, 9-10
POINTER 43
STRING 85
long numeric type 9
LOOP 65, 139
loop control 61
loop statement 61
LoopCloseStatement 65
LoopControl 62
LoopExitsClause 64
LOOPHOLE 47
LoopRange 63, 64
lower bound 16
lower-case 5
MACHINE DEPENDENT 33, 94, 95, 166
MACHINE DEPENDENT RECORD 29
MAX 12, 47
maxlength 82, 83
MIN 12, 47
MOD 5
module 61, 68, 101
ModuleBody 104
monitor 154, 155
initialization 168
lock 155
MONITOR 156, 162
MONITORED RECORD 162, 163-4
MONITORLOCK 156, 162, 166
Multi-module monitor 163
multiple statements 6
MultiplyingOperator 12
name reference 108
result 67, 135, 144
record 73
Resume 139
RESUME 136, 136, 138, 140, 141
ResumeError 141
RETRY 140, 141
RETURN 69, 72, 74, 132, 136, 148
return link 138, 149
RETURN WITH ERROR 135, 138, 139, 166-7
RETURNS 69, 76
RightSide 7, 51
scalar type 22
scale factor 8
scope 74, 79, 99, 108, 137
SELECT 54, 54, 56, 77, 93-6, 98, 99-100, 141, 148
SELECT expressions 56
SelectExpr 56
SelectStmt 54
self-contained 124
Series 3
SHARES 104, 117, 120
short numeric type 9
SIGNAL 59, 135, 136, 141
SignalCall 135
Signaller 138, 139, 142, 150
signals 134
actual 135
catching 138
nested 142
signed number 7
single-component record 73
SIZE 34, 76
SP 141
space 6
START 83, 116, 124, 132, 133, 136, 138, 144, 146-7
start trap 111, 133, 147
startup transient 144, 147
statement 51
Statement 51
StatementSeries 57
static variable 124
StmtSeries 3
STOP 124, 132-3, 136
STRING 7, 82, 83
string literal 83
StringBody 82, 87
strongly typed 4
 subrange 21
type 22, 24
Sum 12
syntax notation 2
TAB 6, 141
tag 93
Tag 118
target
representation 49
type 45
TC 21
terminate
conditionally 62
forcibly 65
normally 63
text 82
THEN 5, 52, 76, 125, 130
THROUGH 63
time stamp 109
timeout 161
TRUE 8, 24
TYPE 20, 21
type
constructor 7
conversion 46
determination 45
type-correct 45
TypeConstructor 21
TypeDeclaration 20
Typeldentifier 21, 96, 107
TypeSpecification 21, 90
unbound variant 93
unique type 91
UnnamedFieldList 31
unqualified 61
unsigned number 7, 43
UNSPECIFIED 19, 43
UNTIL 5, 62, 140
Unwind 139
UNWIND 139, 140, 149-50, 167
upper bound 16
upper-case 5
user-defined type 7
USING 78, 106, 109, 162, 165-6
Variable 12
variant
part 92, 93, 97, 148
record 92
VariantFieldList 118
VariantPart 118
virtual interface record 102
voided component 35
voids 36
WAIT 136, 155, 157, 159, 160-1, 166, 168
WHILE 62
WITH 98
WORD 19
XFER 148