Mesa Language Manual

by James G. Mitchell
William Maybury
Richard Sweet

Version 4.0
May 1978

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

XEROX

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

SYSTEMS DEVELOPMENT DEPARTMENT
3408 Hillview Avenue / Palo Alto / California 94304

CONTENTS

Preface

This document presents a tutorial approach to the Mesa Programming Language. It was written by William Maybury and edited by James Mitchell and Richard Sweet. Its tutorial approach means that it is intended to be read somewhat as a textbook. It is neither a user's guide nor a reference manual.

It is suggested that the *Elements of Mesa Style*, by James Morris, be read in conjunction with this manual. The style manual contains several additional examples of well-constructed Mesa programs, with commentary on using the language properly. Its purpose is to help one take advantage of the assistance Mesa can give in writing programs that work, are reliable, and are maintainable.

Programmers should also read the *Mesa System Documentation.* It describes the facilities available in the Alto implementation of the Mesa Programming system.

Acknowledgements

The principal participants in the development of the Mesa language were James Mitchell, Butler Lampson, Edwin Satterthwaite, and Charles Geschke. Early work on the language was done by William Paxton, with help from Peter Deutsch, Charles Irby, and Charles Dornbush. An implementation was done by Richard Sweet. James Frandeen was involved in thoroughly testing initial versions of the compiler. Vicki Parish, Gail Pilkington, and Janet Farness helped greatly with editing and formatting versions of this manual.

Forward to Version 1.1

This first revision contains numerous small changes to correct errors reported by the many helpful and careful readers of version 1.0. As well, it contains some major changes, and you are encouraged to read the following parts, even if you are an experienced Mesa programmer:

> Chapter 2 has been largely rewritten and contains a detailed discussion of expressions in general, and signed/unsigned arithmetic in particular. The first section of chapter 3 has been reorganized to attempt to better introduce user-defined types.
>
> Two new appendices have been added. Appendix C contains machine-dependent information for Mesa on an Alto. It is not intended to replace the (Preliminary) *Mesa System Documentation,* only to gather such information as may be needed while reading this manual. Appendix D contains a collected grammar for the language and fairly accurately represents the explanatory grammar spread throughout the manual.
>
> Lastly, there is an Index for the manual. This index was "hand-prepared", so no guarantee is made as to its completeness despite the best efforts of myself and Gail Pilkington.

As before, I urge you to notify me of any errors that you discover. Those who did so for the first issue have helped greatly in removing many important errors and in improving the exposition in a number of places. I would especially appreciate suggestions as to index entries that you think should be added, both new items and new page references for already present items. Thank you all.

Jim Mitchell
May 1977

Forward to Version 3.0

The main changes in this second revision of the manual are primarily concerned with modules, programs, and configurations (chapter 7). Other new language features included in Mesa 3.0 are packed arrays (chapter 3) and overlaid variant records (chapter 6). Lesser changes concern arrays, strings, BOOLEAN, and the UNWIND signal.

Please address all comments to SWEET@MAXC1 or by regular mail to

R. E. Sweet
Xerox Corporation
Systems Development Department
3408 Hillview Drive
Palo Alto, CA 94304

Your suggestions, corrections and criticism are encouraged.

Jim Mitchell
Dick Sweet
October 1977.

Forward to Version 4.0

This revision of the manual reflects several significant extensions to the language along with a few minor changes. The new major features are a process mechanism, enhanced arithmetic capabilities, long and base-relative pointers, and more general block structure. Much of chapter 10 (Processes) was written by David Redell. Edwin Satterthwaite, John Wick, and Richard Johnsson helped rewrite several chapters for this edition; Jim Sandman and Barbara Koalkin helped with proofreading and indexing. My sincere thanks to all of them, and to the readers who pointed out errors in version 3.0. Please direct comments and suggestions about this edition to your support group; failing that, send them to me at the address above.

Dick Sweet
May 1978.

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. However, issues concerning the actual developing, compiling, debugging, and loading of Mesa programs will not be treated here; other documents cover those aspects.

Each chapter of this manual discusses some aspect of the language, giving examples, semantics, and syntax. The chapters emphasize different language features at varying levels of detail, and more than one chapter may treat a single feature. Generally, earlier chapters introduce topics, and later ones supply finer detail. Chapter, section, or subsection titles indicate the language issues with which they deal.

Under a given title, discussion is presented at three levels:

(1) Ordinary usage is described (motives, forms, and semantics), frequently with examples.

(2) Syntax equations are shown (when appropriate).

(3) Fine points are covered (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 Mesa. Levels (2) and (3) provide information regarding the full power and details of Mesa.

As a rule, these levels of discourse occur separately and in the indicated order. 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 word or phrase which is italicized is important. If a technical Mesa term is being introduced, it will be in italics; if a term is used before being defined, it will be in italics to warn the reader that it should not be taken lightly and that Mesa has a particular meaning for it. 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 surrounding explanations.

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

1.1. Syntax notation

Mesa's grammar is described using a variation of Backus-Naur Form (or *BNF*) syntax equations. For those unfamiliar with BNF, an explanation follows: reading and understanding it is imperative if you intend to use this manual fully; if you are interested in reading it only at level one, it can safely be skipped. Those familiar with it should still scan this section to see the particular modification of standard BNF used in this manual.

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 with the phrase name on the left, followed by the operator ::= (which should be pronounced "is defined to be"), and is in turn followed by *formation rules* for that phrase class. For example, an **OctalDigit**, which can be any of 0, 1, 2, . . ., 7, is defined by the equation:

```
OctalDigit      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

Phrase names are always printed in the syntax font when their use is meant to be technically accurate. Other names such as BEGIN may also appear in syntax equations. If entirely composed of upper-case characters in the font shown, they are special, or *key words* of Mesa. *Special characters* such as 0, 1, 2, ... above, and others such as =, +, or ←, for instance, stand for themselves in syntax equations; some special characters are formed from more than one single character, e.g., =>. Spaces in syntax equations are used only to separate the items in the rules and have no special significance.

The only other character with special meaning in syntax equations is the vertical bar, |, which separates alternate meanings for the phrase class being defined. It is read as "or": for example, the above equation for **OctalDigit** is read: "An **OctalDigit** is defined to be 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7." The parts of the rule separated by | are called *alternatives*. Each alternative may contain any intermixed sequence of phrase names, special characters, or keywords.

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 may or may not occur when applying the formation rule -- it is optional).

Commentary material imbedded in syntax rules is preceded by a double dash, --, and lies to the right. For example:

```
Digit           ::= OctalDigit | 8 | 9      -- 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 at a time. In order to alert the reader that there are other ways of forming phrases of that class, an ellipsis (...) is used as a final alternative in the rule at that point. The definition of **Statement** is distributed throughout much of the manual in this way. When a certain type of statement such as the **AssignmentStmt** is being discussed, the following partial rule appears:

```
Statement       ::= AssignmentStmt | ...  -- This is just an example.
```

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

The parts of a single alternative are strictly ordered; i.e., 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:

```
ReturnStmt      ::= RETURN |
                    RETURN Constructor
```

("A **ReturnStmt** is defined to be RETURN or RETURN followed by a **Constructor.**") The second alternative means that RETURN and some actual phrase defined by **Constructor** occur in exactly that order.

Syntax equations can indicate recursive substitution. For example:

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

An **identifier** is basically a name in a Mesa program. This equation defines an **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 **IdList** consists of two alternative rules:

Rule 1: (First alternative) "An **IdList** is defined to be an identifier.", i.e., you can substitute any one *name* for an **IdList**.

Rule 2: (Second alternative) "An **IdList** is defined to be an identifier followed by a comma followed by another **IdList**.", i.e., you can substitute *name,* **IdList** for an **IdList**.

To derive a single name, just use Rule 1 as shown for the equation below. (Note: The substitutions are emphasized by writing them in *italics*.)

IdList	::=	*name*	(by Rule 1)

To derive two names separated by a comma:

IdList	::=	*name,* **IdList**	(by Rule 2)
		name, *name*	(by Rule 1)

To derive three names separated by commas:

IdList	::=	*name,* **IdList**	(by Rule 2)
		name, *name,* **IdList**	(by Rule 2)
		name, name, *name*	(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:

```
StmtSeries      ::= empty |
                    Statement |
                    Statement ; StmtSeries
```

The equation is read as, "A **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 **StmtSeries** may take the form specified by the third alternative, "**Statement ; StmtSeries**".

2) Its reference to **StmtSeries**, by recursive substitution, may then take the "**empty**" form, i.e., "**Statement ; empty**"

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 carries over to phrase names such as **IdList** and **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 data types INTEGER, LONG INTEGER, CARDINAL, REAL, CHARACTER and BOOLEAN, 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; it is encouraged to choose or invent suitable types for each particular application. Every variable used in a *legal* 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 information implied by a type can include the following:

Value classification: integer, Boolean, character, etc.

Structure: scalar or an aggregate such as an array or record.

Symbolic representation for constants: numeric, character, string, etc.

Boundary indications: limits for subrange variables, indexing ranges for arrays, etc.

Storage occupancy: number of words, bytes, or bits needed for a value.

Access method: direct, indirect, indexed, etc.

2.1. A slice of Mesa code

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

If either or both of *m* or *n* are zero, then the GCD is 1.

Otherwise, find the remainder of dividing *m* by *n*; set *m* to the value of *n*, and then set *n* to the remainder. If *n* is zero, we are finished, and the value in *m* is the GCD of the original *m* and *n*, except that it may be negative. Taking the absolute value of the last *m* gives us the GCD.

Example. Slice of Mesa Code Using the Euclidean Algorithm

```
-- Given are integers m and n, which can be altered.          (1)
```

```
gcd: INTEGER;                                              (2)
gcd ← 1;            -- gcd=1 if m=0 or n=0                 (3)
IF m#0 THEN                                                (4)
    UNTIL n=0                                              (5)
        DO                                                 (6)
        gcd ← n;                                           (7)
        n ← m MOD n;  -- n gets remainder of m/n           (8)
        m ← gcd;                                           (9)
        ENDLOOP;                                           (10)
gcd ← -- in case one of m or n was negative -- ABS[gcd];   (11)
```

The example contains eleven 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) declares a variable, *gcd*, of type INTEGER. A semicolon separates the declaration from the following statements. Line (3) contains an assignment statement (the character "←" is Mesa's assignment operator). This presets *gcd* in case one of the given values is zero.

Line (4) begins an IF statement, testing for *m* not equal to zero (the character "#" is Mesa's "unequal" operator). If *m* is not zero, the compound statement following THEN (lines (5) through (10) inclusive) is executed; otherwise, that statement is skipped.

Iteration for the Euclidean Algorithm is performed in the loop (UNTIL *n*=0 DO...ENDLOOP), 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.

Notice that statements are separated by semicolons. A semicolon at the end of a statement series that is embedded in another statement (such as the series in the loop) is harmless and optional; you can save thought by writing a semicolon after *every* statement in the series.

Within the loop, line (7) saves the value of *n* in *gcd*. Line (8) then updates *n* for the next iteration, if any. It assigns to *n* the value of the expression "*m* MOD *n*," which gives the remainder of dividing *m* by *n*. Line (9) updates *m* to contain the value that *n* had before line (8) and which was saved in *gcd*. Control then reaches the end of the loop, line (10), and transfers back to line (5), where *n* is again tested against zero. If it is zero, the loop is not entered again, and execution continues with the first statement past the loop, line (11).

In all cases, control eventually reaches the assignment statement in line (11). Either *gcd* has its initial value (i.e., 1), or it contains the value *n* had just before it became zero. The expression "ABS[*gcd*]" is the form for calling a function and passing it one or more arguments; square brackets are used for this purpose. Normal parentheses, "(" and ")", are used only for nested expressions, e.g., "*a**(*b*+*c*/(*d*-*e*)**f*)." The assignment places the absolute value of *gcd* 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*, and *n* 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 are keywords, and others name built-in functions, such as ABS. All such words that have special meaning and 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 hitting 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.

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 2.4.4). As a separator, any sequence of contiguous blanks is equivalent to a single blank. A TAB character also behaves exactly like a blank when used as a separator.

A carriage-return character (hereafter called a CR in this manual) 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, CR acts as the terminator of that comment. Thus, multiline comments (those containing CRs) 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, and the user is warned to be careful 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:

gcd: INTEGER;

This declares *gcd* 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, we use "list" 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 ← 1;
gcd ← n;
n ← m MOD n;
m ← gcd;
gcd ← ABS[gcd];
```

An assignment statement has the following syntax:

```
AssignmentStmt    ::= LeftSide ← RightSide | . . .
LeftSide          ::= identifier | . . .    -- plus forms for array indexing, etc.
RightSide         ::= 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 2.4.5 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 (←), 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 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, LONG INTEGER *and* CARDINAL

Normally a computer does not provide operations on the mathematical integers (denoted by $\underline{Z}$), but rather on a finite set of values restricted to a range determined by the properties of a particular machine. Programmers often (usefully) ignore the distinction and write programs as if they manipulate elements of $\underline{Z}$. On a machine with a short word length (e.g., 16 bits or less), the distinction is difficult to ignore. For this reason, Mesa provides three built-in numeric types, with values that are restricted to different subranges of $\underline{Z}$. The range of the first type, INTEGER, is (approximately) symmetric about zero, and values of type INTEGER are represented as *signed* numbers. The range of the second type, LONG INTEGER, is also symmetric about zero but larger. The range of the final type, CARDINAL, is some finite interval of the natural numbers $\underline{N}$ = {0, 1, 2, ...} 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.

In terms of an implementation, values of types INTEGER and CARDINAL are expected to be represented by single machine words, while a value of type LONG INTEGER is expected to occupy two words. For this reason, INTEGER and CARDINAL will be referred to as *short numeric types*; LONG INTEGER, as a *long numeric type*. 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 (Mesa uses ".." in place of the mathematician's comma in its interval notation):

INTEGER	$[-2^{N-1} .. 2^{N-1})$
CARDINAL	$[0 .. 2^{N})$
LONG INTEGER	$[-2^{2N-1} .. 2^{2N-1})$

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

The programmer must choose an appropriate type for each numeric variable. LONG INTEGERS offer a greater range but occupy more storage and are generally more time-consuming to manipulate. 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, CARDINAL and LONG INTEGER are distinct and not interchangable. On the other hand, it is clear that they are closely related. Mesa allows most combinations of these types to occur within assignments, arithmetic expressions and relational expressions, but care is necessary to avoid failures of representation, especially when values with different representations are mixed. This is discussed further in sections 2.4.5 and 2.5.1.

2.4.1.1. Numeric literals

A *numeric literal* always denotes an element of $\underline{N}$ (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, a numeric literal must denote an acceptable (positive) LONG INTEGER value. If its value lies in the range of CARDINAL $\cap$ LONG INTEGER, i.e., in $[0 .. 2^{N})$, it may also be considered a CARDINAL value in any context requiring a CARDINAL (section 2.5.1). Similarly, if the value lies in the range of CARDINAL $\cap$ INTEGER, i.e., in $[0 .. 2^{N-1})$, it may also be considered an INTEGER.

Numeric literals are instances of the phrase class **number**:

> 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 representing 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

2.4.2. Type REAL *(interim)*

The values of Mesa's type REAL are approximations of mathematical real numbers. These approximations are sometimes called *floating-point* numbers. The limitations of such discrete approximations are many and well known; they are not discussed here. 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. Users of real arithmetic must provide an appropriate set of procedures for performing the arithmetic and relational operations.

The source language provides no denotations of REAL literals, since the compiler does not know the internal representation expected by the user-supplied procedures. Mesa does provide automatic conversion from INTEGER, LONG INTEGER or CARDINAL to REAL (section 2.4.5). Thus **numbers** (numeric literals) can appear in REAL expressions and provide denotations of certain REAL constants.

2.4.3. 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 (like IF) and in certain loop constructs. For instance, the following statement appears in Example 1:

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

The expression "*m*#0" is a BOOLEAN expression; its value is FALSE if the value of *m* is zero and TRUE otherwise. The expression "*n*=0" is just the opposite; its value is TRUE if *n* is zero and is FALSE otherwise. The relational operators discussed in section 2.5.2 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 instance, the above program outline could validly be replaced by the following:

```
mNotZero, nIsZero: BOOLEAN;
. . .
mNotZero ← (m#0);
```

```
nIsZero ← n=0;                  -- compute whether n is zero
IF mNotZero=TRUE THEN           -- equivalent to just mNotZero by itself
    UNTIL nIsZero
        DO
        . . .
        nIsZero ← n=0;          -- recompute whether n is zero just before testing
        ENDLOOP;
```

2.4.4. *Type* CHARACTER

A value of type CHARACTER represents a *single* character of text. CHARACTER values are considered to be 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 2.5.1.4).

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:

```
lowerCaseA ← 'a;
mark ← ' ;              -- mark is set to be a blank. Here a blank is significant
endMarker ← '; ;        -- endMarker is set to be a semicolon
asciiCR ← 15C;          -- an Ascii Carriage Return character;
```

2.4.5. *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 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 2.4.1.1). The conformance relations can be summarized as follows:

INTEGER and CARDINAL conform to CARDINAL.

INTEGER and CARDINAL conform to INTEGER.

INTEGER, LONG INTEGER and CARDINAL conform to LONG INTEGER.

INTEGER, LONG INTEGER, 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 short numeric types with different representations are intermixed. Mathematically, $\underline{Z} \supset \underline{N}$; however, it is not necessarily true that INTEGER $\supset$ 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.

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 CARDINAL $\cap$ INTEGER. In the current version of Mesa, the conversion function is always compiled as the identity function (i.e., it generates no code), and no check is actually made. *It is the responsibility of the programmer to determine 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 LONG INTEGER $\supset$ INTEGER and that LONG INTEGER $\supset$ CARDINAL; thus it is always valid to assign a short numeric value to a LONG INTEGER variable. The properties of conversion to type REAL are not specified by the language.

Some fine points:

A user supplied procedure *FLOAT* is automatically applied to convert a value from type LONG INTEGER to REAL. Short numeric values are converted first to LONG INTEGER and then to REAL.

Conversion from a short numeric value to a LONG INTEGER (and thus to a REAL) is substantially more efficient when the value has an unsigned representation.

The conversion of constants to type REAL is not done by the compiler but occurs every time the containing expression is evaluated at run-time.

Neither BOOLEAN nor CHARACTER conforms to any other basic type.

Examples:

```
i: INTEGER;   n: CARDINAL;   ii: LONG INTEGER;  x: REAL;
(valid)          i ← 0;
                 ii ← 0;
                 x ← n;
                 x ← ii;
(invalid)        i ← x;
                 n ← 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

Expression ::= Disjunction | AssignmentExpr | IfExpr | SelectExpr | . . .

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         -- same as what can receive values
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.

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 $-(i/j)=-i/j=i/-j$. 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 $(i/j)*j+(i \text{ MOD } j) = i$, 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 MIN computes the minimum value in a list of expressions; similarly, the MAX function, the maximum value. The built-in function 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 left 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. *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*-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*
multiply the result by *b*
add that result to the value of *a*

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 +, -, *, /, MIN, MAX and ABS for all the numeric types. The operation MOD is defined for all numeric types except REAL; the operation of unary negation, for all but 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 INTEGER values yields an INTEGER result, and dividing one CARDINAL by another yields a 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 INTEGERs is fast; multiplication and division are done by software and are relatively slow.

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

```
FADD, FSUB, FMUL, FDIV: PROCEDURE [REAL, REAL] RETURNS [REAL];
FCOMP: PROCEDURE [REAL, REAL] RETURNS [INTEGER];
    -- returns a value that is: 0 if equal, negative if the first is less, positive otherwise
FLOAT: PROCEDURE [LONG INTEGER] RETURNS [REAL];
```

All other REAL arithmetic operations are fabricated from these primitives.

Although the mathematical integers ($\underline{Z}$) and real numbers are closed under all these operations (except division by zero), the subranges defining the types INTEGER, LONG INTEGER and 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. The current version of Mesa does not detect these failures; *it is the programmer's responsibility to guard against overflow and underflow conditions.*

The effects of Mesa's conventions with respect to subtraction are worth emphasizing. If both operands have valid signed representations, the result is an INTEGER. If both have only unsigned representations, the result is a CARDINAL and is considered to overflow if the first operand is less than the second.

Example:

```
    i: INTEGER;   m, n: CARDINAL;
    i ← m-n;                            -- should be used only if it is known that m >= n
    i ← IF m >= n THEN m-n ELSE -(n-m);        -- a safer form (section 3.6)
```

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 rules for expressions involving types REAL and LONG INTEGER are .easy to state:

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

Otherwise, if any operand is LONG INTEGER, the LONG INTEGER operation is used.

The rules governing combination of short 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 ∩ CARDINAL have whatever representation is required by context.)

Examples:

```
   i, j, k: INTEGER;   m, n: CARDINAL;
   Factors:     n
                15
                (i+j+k)
                -15
                MIN[i, j, k, -15]
   Products:    m*n
                i/-15
                n MOD 8
                m/n*10               -- same as (m/n)*10 because of left-associativity
                -k*(i+1)/2 MOD 3     -- same as (((-k)*(i+1))/2) MOD 3
   Sums:        i+1
                -i+j
                j-i
                n-n MOD 8            -- same as n-(n MOD 8) because of precedence
                m - m/n*n            -- same as m MOD n
```

2.5.1.1. The operator LONG

The built-in function LONG converts any value with a short numeric type to a LONG INTEGER. The syntax is as follows:

```
    BuiltinCall  ::=  ... | 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:

```
LONG[m*n]                          -- "short" multiplication, overflow lost
LONG[m]*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*].

2.5.1.2. 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:

```
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, 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] [-1 .. -1) (-2 .. -1) (-2 .. -2]

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 short numeric operands with different representations; an attempt to compare two such values is an error. The precise rules can be found in section 3.5.

Examples:

Relations:	*n* = 15	
	m # *n*	-- or *m* ~= *n*
	i<=*j*	
	(*i*<*j*) = (*j*<*k*)	-- = with two BOOLEAN operands
	n IN [1 .. 5)	-- *n*>=1 and *n*<5
	i NOT IN [-1 .. 5]	-- only legal if *i* is signed (because -1 is)

2.5.3. BOOLEAN *operators*

The next lowest precedence operators apply only to BOOLEAN values. They are NOT (logical negation), AND and OR. The syntax is

Negation	::=	**Relation | Not Relation**
Conjunction	::=	**Negation | Conjunction** AND **Negation**
Disjunction	::=	**Conjunction | Disjunction** OR **Conjunction**

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 may or may not be evaluated, depending on the first operand's value.

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*=NIL OR *p*.*f*=0".

Examples:

```
Negations:     NOT i=15           -- same as NOT(i=15)
               ~q                 -- q must be of type BOOLEAN
               ~(p AND q)
Conjunctions:  i<=j AND j<k
               p AND ~q
               i=5 AND j NOT IN [-1..1]
Disjunctions:  m>n OR m=15
               ~p OR ~q
```

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 for the **AssignmentStmt**:

AssignmentExpr ::= LeftSide ← 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:
            m←15
            m←n←15
            m←n←n+1                -- same as m←(n←(n+1))
            i←(j←(j+1) MOD n)*2    -- all these parentheses are necessary
```

Rules governing assignments of numeric values are summarized in section 2.4.5.

Fine point:

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

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:

```
delimited: BOOLEAN ← FALSE;
```

Variables (of the same type) can be initialized collectively:

```
n, n0: INTEGER ← -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: INTEGER ← ABS[n];            -- this will set i to 7
    iSquared: INTEGER ← i*i;        -- iSquared is initialized to 49
    j: INTEGER ← iSquared-i+1;      -- j is initialized to 49-7+1 = 43
```

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

```
    octalRadix: INTEGER = 8;
```

This means that *octalRadix* is to have a fixed value. *It is never valid as the* **LeftSide** *of an assignment*. We call *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 also be arbitrary expressions. Paraphrasing the earlier example:

```
    i0: INTEGER = ABS[-octalRadix]; i0Squared: INTEGER = i0*i0;
    j0: INTEGER = i0Squared-i0+1;
```

The initializing expression can use values that are not known at compile time. In this example, if *octalRadix* did not have fixed initialization, the values of *i0, i0Squared,* 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.

If the compiler does know initial values (for variables declared with fixed form initialization), it can use them wherever those variables are referenced. This knowledge can be exploited when analyzing expressions, processing other declarations, or generating object code.

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, i0Squared,* and *j0* in the previous section are all compile-time constants).

Example:

```
    beta: INTEGER = 3;
    alpha: INTEGER = beta-1;
```

In this case, *alpha* is a compile-time constant (with the value 2), since the expression *beta*-1 involves only compile-time constants. Compile-time constants need not occupy memory at run-time. The compiler replaces references to compile-time constants by their known values whenever possible. In this case, *alpha* and *beta* would not require storage at run-time.

A 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 INTEGER, but because its value is 2, it may also be used as a 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).

A fine point:

In certain contexts, an expression is required to yield a compile-time constant. 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 or REAL.

Any function (but not built-in function) application (chapter 5).

The @ operation (section 3.4).

The SELECT operation (section 4.3.3).

2.7. More general declarations

We have now introduced all the syntactic parts of a declaration, and can describe the general form of a **Declaration** as

```
Declaration         ::= IdList : TypeSpecification Initialization ;
```

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

```
TypeSpecification  ::= PredefinedType | . . .
PredefinedType     ::= INTEGER | CARDINAL | LONG INTEGER | REAL |
                       BOOLEAN | CHARACTER |
                       STRING |           -- see chapter 6
                       WORD |             -- see fine point below
                       UNSPECIFIED        -- see fine point below
```

Also, we can formally define an **Initialization** as

```
Initialization      ::= empty          |
                        ← Expression   |
                        = Expression   |
                        . . .             --other forms are given later
```

Fine points:

The built-in 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 built-in type UNSPECIFIED is a device for bypassing most type checking. It is a necessary evil. 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, (perhaps surprisingly) STRING, or 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 compiler treats the UNSPECIFIED value as if it were 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 table of values, 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 on 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:

```
        TypeDeclaration ::= idList    : TYPE = 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.,

```
        Declaration     ::= IdList    : TypeSpecification 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 "=") which is a **TypeSpecification**.

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

```
    SignedNumber: TYPE = INTEGER;
    UnsignedNumber: TYPE = CARDINAL;
    TruthValue: TYPE = BOOLEAN;
    Char: TYPE = 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

```
TypeSpecification  ::= PredefinedType    |
                       TypeIdentifier    |
                       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 **TypeIdentifier** 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 interchangably. 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). The simplest form of a **TypeIdentifier** is given by

```
TypeIdentifier  ::= identifier |     -- 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

```
TypeConstructor  ::= EnumerationTC      |  -- for enumerations
                     SubrangeTC         |  -- for subranges
                     ArrayTC            |  -- for arrays
                     RecordTC           |  -- for records
                     PointerTC          |  -- for pointers to data objects
                     LongTC             |  -- for long pointers
                     ProcedureTC        |  -- see chapter 5
                     ArrayDescriptorTC  |  -- see chapter 6
                     RelativeTC         |  -- see chapter 6
                     SignalTC           |  -- see chapter 8
                     PortTC             |  -- see chapter 9
                     ProcessTC             -- 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 like 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 general definition of the class **ElementType** is

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

A fine point:

Note that LONG INTEGER is not an element type. It is not possible to declare subranges of LONG INTEGER or to use LONG INTEGERs 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 ideally suited to this kind of 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 ← 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:

```
EnumerationTC     ::= { 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:

```
Color: TYPE = {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

```
Fruit: TYPE = {apple, orange};
```

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

```
Primary          ::= . . . | TypeIdentifier [ 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

```
SubrangeTC    ::= TypeIdentifier Interval |
                  Interval
```

The **TypeIdentifier** 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,

```
SymmetricRange: TYPE = INTEGER[-1..1];
PositiveNumber: TYPE = CARDINAL[1..maxCARDINAL];     -- see appendix C
UpperCaseLetter: TYPE = CHARACTER['A..'Z];
DegenerateLogic: TYPE = BOOLEAN[TRUE..TRUE];
CoolColor: TYPE = Color(yellow..LAST[Color]];     -- excludes red, orange, yellow
AthroughM: TYPE = UpperCaseLetter['A..'M];     -- subrange of a subrange
```

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 **TypeIdentifier** (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 permissable 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 (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.* The current version of Mesa provides no range checking, and *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:

```
n: CARDINAL [0..10]; m: INTEGER [-5..5];

m ← 0; n ← 0;          -- inherited literals
n ← n+1;               -- not valid if n = 10
n ← m;                 -- only valid if m IN [0..5]
```

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[*AthroughM*]='M. It is also possible to iterate over all values in a subrange (section 4.5).

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 permissable 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 ∩ CARDINAL. Similarly, it has an unsigned representation if both bounds are elements of CARDINAL and at least one is not an element of INTEGER ∩ CARDINAL. If both bounds are elements of INTEGER ∩ CARDINAL, values of that subrange type are considered to have *both* representations. Any other combination of bounds is illegal.

Examples:

```
s1: [-10..10];        -- signed representation
s2: [100..33000];     -- unsigned representation (if 33000 is not an INTEGER)
s3: [0..10);          -- both representations
```

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:

```
i: INTEGER;  n: CARDINAL;  -- plus the declarations above
(signed)       s1 + 1
               s1 + s3
               s3 - i
(unsigned)     s2 + 1
               s2 + s3
               s3 * n
```

A fine point:

The representation assumed for a literal also depends upon context. In fact, any 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 **Primary** $S[e]$ has the same value as e and is additionally an assertion that e IN [FIRST[$S \cap T$] .. LAST[$S \cap T$]] is TRUE. In addition to user defined types, the basic types INTEGER and CARDINAL may be used in range assertions.

In the current version of Mesa, *such assertions must be verified by the programmer.* In addition to providing documentation, the subrange assertion does affect the attributes attached to the expression. For example, an assertion of an 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 operations when a unique representation cannot be inferred from the operands (section 3.6).

Examples:

```
    i: INTEGER;   n: CARDINAL;   S: TYPE = [0..10];
    CARDINAL[i]        -- i is asserted to be nonnegative
    S[n]               -- asserts n IN [0..10]
```

3.2. Arrays

Arrays are indexable collections of homogeneous components. In other words, a given array's components 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 *index type*) and the *component type* determine the array type. For example:

```
    earningsPerQuarter: ARRAY [1..4] OF INTEGER;
```

declares a variable with a constructed array type having an index type of [1..4] and a component type of INTEGER. Thus, *earningsPerQuarter* is an array of four integer elements: *earningsPerQuarter*[1], *earningsPerQuarter*[2], ... , *earningsPerQuarter*[4]. *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 *component* is used and always means an item contained within an aggregate.)

An index type must be an element type (other than INTEGER or 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 *indexed reference* is the means by which one 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 **Expression** with a type conforming to the array's index type.

An index type can be specified using a type identifier:

```
    Quarter: TYPE = [1..4];
    profit, loss, earnings: ARRAY Quarter OF INTEGER;
    thisQuarter: Quarter;
    . . .
    earnings[thisQuarter] ← profit[thisQuarter] - loss[thisQuarter];
```

The arrays *profit, loss,* and *earnings* have *Quarter* as their index types, and *thisQuarter* is a subrange variable with type *Quarter.*

Index types may also be enumerations or subranges thereof. For example,

```
CallType: TYPE = {longDistance, tieLine, toll, local, inPlant};
closeCalls: ARRAY CallType[toll..inPlant] OF CARDINAL;
...
closeCalls[local] ← closeCalls[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:

```
Matrix3by4: TYPE = ARRAY [1..3] OF ARRAY [1..4] OF INTEGER;
mxy: Matrix3by4;
...
mxy[3][4] ← 0;          -- clear last component.
```

In the assignment statement, *mxy* is an expression of array type (with index type [1..3] and component type ARRAY [1..4] OF INTEGER). *mxy*[3] is an indexed reference to the third component of *mxy*. This in turn yields an expression of array type (whose index type is [1..4] and component type is INTEGER). Thus, *mxy*[3][4] is an indexed reference to the fourth component of that sub-array. Because of left-associativity, *mxy*[3][4] is the same as (*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., *octalChar*[*n*] holds the character denoting octal digit *n*:

```
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.*

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

```
freshVector: ARRAY [0..3) OF CARDINAL = [0, 0, 0];
currentVector: ARRAY [0..3) OF CARDINAL ← freshVector;
```

In this case, *currentVector* is initialized with *freshVector*'s value, i.e., all three of *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 *currentVector* above uses assignment in this way. Similarly, the expression "*currentVector* = *freshVector*" yields the result TRUE if and only if all three components of each array are equal (as they are in the above example). Because the declaration of *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**:

```
ArrayTC          ::= PackingOption ARRAY IndexType OF ComponentType
PackingOption    ::= empty  |     -- elements word aligned
                     PACKED       -- elements potentially packed within words
IndexType        ::= ElementType | TypeIdentifier
ComponentType    ::= TypeSpecification
```

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

Declarations of initialized array variables take the form

IdList : ArrayTC Initialization

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

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

```
IndexedReference    ::= Variable [ Expression ] |
                        ( Expression ) [ Expression ]
LeftSide            ::= . . . | IndexedReference
```

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, *arg* denotes an expression with some array type.

LENGTH[*arg*]	-- yields the number of array elements.
BASE[*arg*]	-- yields a pointer value for locating the first array element.
DESCRIPTOR[*arg*]	-- yields *arg's* array descriptor value (consisting of base and length).

3.2.2. Constructors for arrays

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:

```
Constructor       ::= OptionalTypeId [ ComponentList ]
OptionalTypeId    ::= TypeIdentifier | empty
ComponentList     ::= PositionalComponentList |
                      ...     -- other forms for record constructors
PositionalComponentList   ::=   Component |
                                PositionalComponentList , Component
Component                 ::=   empty |        -- elided component
                                Expression
```

The **empty** components in a constructor are said to be *elided*, and their values are undefined. The number of **Expressions** plus elided components in an array constructor 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 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;
triplet ← Triple[11, 12, 13];
```

The final statement assigns the following values to the array components:

```
triplet[1] = 11
triplet[2] = 12
triplet[3] = 13
```

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

```
triplet ← [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];
```

Some fine points:

The value of an elided 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. To declare a record type, the type of each individual component must be supplied, as in the following example:

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.

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*".

The Mesa compiler packs record components into machine words as efficiently as it can, using information such as the number of elements in a subrange. The components may be arranged in an "actual" order that differs from the *component order* given by their left-to-right order in the type constructor. All records of the same type *always* have the same component arrangement.

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)].

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":

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.

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:

```
    FieldList           ::=  [ UnnamedFieldList ] | [ NamedFieldList ]
    UnnamedFieldList    ::=  TypeSpecification |
                             TypeSpecification , UnnamedFieldList
    NamedFieldList      ::=  IdList : FieldDescription |
                             NamedFieldList , IdList : FieldDescription
    FieldDescription    ::=  TypeSpecification
```

Examples:

```
[i: INTEGER, b: BOOLEAN, c: CHARACTER]       -- a named field list
[INTEGER, BOOLEAN, CHARACTER]                -- a similar, but unnamed field list
[f1: CHARACTER, f2, f3: INTEGER]             -- components listed and declared together
[f1: CHARACTER, f2: INTEGER, f3: 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[b: B];
B: TYPE = RECORD[a: A];
```

Field lists occur in constructors of types other than records, such as PROCEDUREs, SIGNALs and PORTs (chapters 8 and 9), 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 this 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 as 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, perfectly 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; the record structure of *recVar* must be recognized (as in the last two assignments). In addition, 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 is defined. Second, variables are declared to be of that type. 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. In practice, an initializing expression is typically a constructor, another declared record of the same type, or a call on a function that returns a record value of the correct type.

Initialization of record variables is shown in the next example.

```
Time: TYPE = RECORD
        [
        hr: [1..12],
        min, sec: [0..60),
        meridian: {am, pm}
        ];

noon:      Time = [hr:12, min:0, sec:0, meridian:pm];
midnight:  Time = [hr:12, min:0, sec:0, meridian:am];
time:      Time ← midnight;               -- Start time at midnight.
```

Some fine points:

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. Mesa accepts components exactly as given, positioning them left-to-right in machine words. In general, "fill" components are needed to insure that no field crosses a word boundary (unless it starts on one). Of course, components may themselves be aggregates occupying more than one word (*ChannelCommand* might be a record itself, for instance). Fill components may also be needed prior to fields with types that are always aligned, such as arrays and multi-word records.

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 machine-word boundary) and some may not. Components occupying a full machine-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. This allows DESCRIPTORs for packed arrays to contain only a word address for the base.

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

```
LatType: TYPE = RECORD[degs: [0..360),   mins, secs: [0..60)];
LngType: TYPE = RECORD[degs: [-90..90],  mins, secs: [0..60)];
PosType: TYPE = RECORD[latitude: LatType, longitude: LngType];
somePosition: PosType;
```

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

Qualified Reference	Refers To
somePosition.latitude	1st sub-record
somePosition.longitude	2nd sub-record
somePosition.latitude.degs	1st component of 1st sub-record
somePosition.longitude.secs	3rd component of 2nd sub-record

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..99] 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, now-33];
```

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

Component	Value	
day	25	
month	*Apr*	
year	*now*-33	(which is 1943)

A **Constructor** is a **Primary** and may only be used in a **RightSide** context (i.e., not as the **LeftSide** of an assignment).

Record constructors are of two kinds: *keyword constructors* and *positional constructors*. Within both kinds, component values may either be supplied or be omitted for various components of the record. If omitted, the values used are undefined.

Syntax equations:

```
Primary          ::=  ... | Constructor
Constructor      ::=  OptionalTypeId [ ComponentList ]
OptionalTypeId   ::=  TypeIdentifier | empty
ComponentList    ::=  KeywordComponentList |
                      PositionalComponentList
KeywordComponentList     ::=   KeywordComponent |
                               KeywordComponentList , KeywordComponent
PositionalComponentList  ::=   Component |
                               PositionalComponentList , Component
```

```
KeywordComponent    ::=   identifier : Component
Component           ::=   empty |       -- elided component
                          Expression    -- explicit component
```

This is a fairly complex set of syntax definitions. (Note that some of it is repeated from section 3.2.2.) The following examples and explanations illustrate the options implied by the equations.

The initial **TypeIdentifier**, 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, and all keywords must appear*, but the order is irrelevant. For example, the following keyword constructors are equivalent:

Date[*day*: 25, *month*: *Apr*, *year*: *now*-33]
Date[*month*: *Apr*, *day*: 25, *year*: *now*-33]

All of these keyword constructors specify values for all the components. The following keyword constructor *elides* the *month* component (the place for the component value is specified, but no value is given):

Date[*day*: 25, *month*: , *year*: *now*-33] -- *month* is *elided*

According to the declaration of *Date*, this constructs a record with a second component having an undefined value. Elided components *always* have 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 two constructors are equivalent:

```
Date[day: 25, month: , year: now-33]     -- value of month is undefined
Date[25, , now-33]                       -- value of 2nd component is undefined
```

Positional constructors may elide values, 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 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. 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 *birthDay* (defined in the previous section) to the variables *dd*, *mm*, and *yy*, in that order:

```
[dd, mm, yy] ← birthDay;
```

This has the same effect as the following three separate assignments, except that *birthDay* is evaluated only once:

```
dd ← birthDay.day;   mm ← birthDay.month;   yy ← birthDay.year;
```

An extractor roughly resembles a constructor in form, but there are some crucial 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 **TypeIdentifier.**

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] ← [25, Apr, 1943];          -- invalid
```

The statement should specify the type of the constructed value:

```
[dd, mm, yy] ← Date[25, Apr, 1943];      -- 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:

```
[day: dd, month: mm, year: yy] ← birthDay;
[month: mm, day: dd, year: yy] ← birthDay;
```

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

```
[day: dd, month: , year: yy] ← birthDay;    -- month elided
[day: dd, year: yy] ← birthDay;             -- month omitted
[dd, , yy] ← birthDay;                      -- 2nd component elided
```

A positional extractor may elide trailing components without having to supply trailing commas as required in a positional constructor. The *year* component of *birthDay* is elided below.

```
[dd, mm] ← birthDay;
```

An extraction operation (unlike an ordinary assignment) yields no value. *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 ← [x, y, z] ← s;      -- invalid
[x, y, z] ← r ← s;      -- valid
```

Syntax equations:

```
AssignmentStmt         ::=  . . . | Extractor ← RightSide
Extractor              ::=  [ KeywordExtractList ] |
                            [ PositionalExtractList ]
KeywordExtractList     ::=  KeywordExtract |
                            KeywordExtract , KeywordExtractList
KeywordExtract         ::=  identifier : ExtractItem
PositionalExtractList  ::=  ExtractItem |
                            ExtractItem , PositionalExtractList
ExtractItem            ::=  empty |    -- component is ignored
                            LeftSide   -- component is assigned to LeftSide
```

The **identifiers** in a **KeywordExtractList** must be field names for the record type. Note that **Extractors** cannot be nested.

Fine point:

An extractor can be empty. An empty extractor simply discards a record value. This can be useful when a procedure is called for its side effects only but returns an (unwanted) value. For example:

```
[] ← CloseFile[inputFile];
```

3.4. Pointers

Pointers allow efficient indirect access to objects. A pointer may refer to only one specific type of data. For instance, the following pointer provides access only to objects of type CARDINAL:

```
intPtr: POINTER TO CARDINAL;
```

Another pointer might be specified to point only to BOOLEAN objects:

```
boolPtr: POINTER TO BOOLEAN;
```

These are different types of pointers since they have different *reference types*, CARDINAL and BOOLEAN. Furthermore, since CARDINAL and BOOLEAN are incompatible types, these pointer types are also incompatible; i.e., assignment of *boolPtr* to *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 and can be used as a **LeftSide** or **RightSide**. Thus *intPtr*↑ and *boolPtr*↑ are variables of type CARDINAL and BOOLEAN respectively. The statement

```
boolPtr↑ ← (intPtr↑ = 0);
```

is executed by following *intPtr* to obtain a CARDINAL value, testing that value, and assigning the result to the BOOLEAN variable referenced by *boolPtr*.

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

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

Pointers to record objects may be used to qualify field names. If record *candidate1* is the referent of *winner*, then qualifications such as

```
winner.age     winner.sex     winner.party
```

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

```
FamilyMember:  TYPE = RECORD
        [
        someone: Person,
        mother, father: POINTER TO 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:

```
winner ← loser;
winner↑ ← loser↑;
```

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

```
orderedPtr: 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, NIL. It conforms to *any* unordered pointer type and denotes a pointer value that has no valid referent. For example:

```
IF intPtr = NIL THEN boolPtr ← NIL;
```

A pointer with value 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 POINTER TO *T*; for example:

```
winner ← @candidate1;
```

Pointer generation should be done with caution; it is possible to assign the resulting pointer to a variable that lives longer than the referenced object. A non-NIL pointer value with no valid referent is said to be a *dangling reference.* The Mesa language does not prevent dereferencing such a pointer, but doing so produces an undefined result. *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:

```
PointerTC     ::= Ordered Base POINTER TO TypeSpecification |
                  Ordered Base POINTER Interval TO TypeSpecification

Ordered       ::= empty | ORDERED
Base          ::= empty | BASE
```

The **TypeSpecification** in a **PointerTC** specifies the reference type of the pointer type. Two pointer types are equivalent if their reference types are equivalent and if the attribute **Ordered** is 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 and if either the **Ordered** attributes are identical or the first is ORDERED and the second is not. The **Base** attribute is ignored in determining conformance.

Fine points:

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. 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:

```
IndirectReference ::= Variable ↑ |
                      ( Expression ) ↑
LeftSide          ::= . . . | IndirectReference
```

The postfix operator ↑ performs explicit dereferencing of the pointer expression it follows. It 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]↑.mother↑.someone        -- (((group[i])↑).mother)↑).someone
```

If *p* is an arbitrary pointer expression, then *p*↑ can be read as "*p*'s referent" or "referent of *p*". Application of the ↑ operator produces a variable that may be used as a **LeftSide** or as a

RightSide.

The syntax used for address generation is

```
        Primary             ::=  . . . | @ 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 ↑; e.g., @*x*↑ is equivalent to @(*x*↑) (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*.

Elements of packed arrays are not aligned.

Any component of a record that occupies less than a single word may not be aligned (but arrays, even if packed, are always aligned).

All other variables are aligned.

When pointing at a simple variable, one must carefully consider whether the pointer value will *live* longer than the variable to which it points. (It is assumed here that the reader is familiar with procedures, local variables, and global variables.) Consider the following procedure:

```
pointer1, pointer2: POINTER TO INTEGER;      -- two global variables
. . .
RiskyProc:   PROCEDURE[i: INTEGER] =         -- i is a local variable
    BEGIN
    myLocal: INTEGER;                        -- and so is myLocal
    . . .
    pointer1 ← @i;           -- risky: i will die upon RETURN
    pointer2 ← @myLocal;     -- also risky
    . . .
                -- The "risky" pointers are valid up to this point, but
    RETURN;     -- NOT after this statement is executed.
    END;
```

The problem here is that 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 whose referents exist 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 existing 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., only the length attribute is changed.

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:

> 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 anomolous 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: TYPE = RECORD [f: T, ...];
p, q: POINTER TO R;
pp, qq: LONG POINTER TO R;
pT: POINTER TO T;
ppT: LONG POINTER TO T;

-- the following are valid.

pp ← qq;  pp ← NIL;  pp ← p;
pp = qq, pp = NIL, pp = q;      -- long comparisons
pT ← @p.f;  ppT ← @pp.f;
ppT ← @p.f;                     -- pointer lengthened

-- the following are not valid.

pp = ppT;                       -- incompatible types
p ← pp;  pT ← @pp.f;            -- no automatic shortening
```

3.4.4. *Automatic dereferencing*

Automatic dereferencing converts a *pointer* **RightSide** of type POINTER TO *T* into one of type *T* in the context of a dot qualification (section 3.3.3), a procedure argument list, or an array indexing operation (the last two are syntactically identical). For example, in the following two statements, the **LeftSides** are equivalent (and the **RightSides** nearly so):

```
winner.party  ← Democratic;
winner↑.party ← Republican;
```

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

```
actualArray: ARRAY [0..20) OF INTEGER;
arrayPtr: POINTER TO ARRAY [0..20) OF INTEGER ← @actualArray;
arrayFinger: POINTER TO POINTER TO ARRAY [0..20) OF INTEGER ← @arrayPtr;
actualArray[1] ← 3;
arrayPtr[1] ← 3;          -- arrayPtr↑[1] ← 3
arrayFinger[1] ← 3;       -- 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:

Expression	Inherent Type of Expression
34	[34..34]
NIL	POINTER TO UNSPECIFIED
x < *y*	BOOLEAN
x	Type declared for *x*
array[1]	Type specified for *array*'s components
@*x*	POINTER TO *x*'s type
(*x* ← *e*)	*x*'s type

Mesa's type rules 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 **RightSide** expression. 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:

```
LType: TYPE = RECORD[c: CType];
lVar: LType;
. . .
lVar ← anyExp;               -- anyExp's target type is LType
lVar ← LType[c: someExp];    -- someExp's target type is CType ...
lVar.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* **RightSide**. (Of course, a **LeftSide** may contain **RightSide** expressions, such as array subscripts, that might be coerced.)

Certain other contexts imply a target type. For example, the target type for an array subscript is the array's index type. 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: TYPE = RECORD [i: INTEGER];
> v: R ← [R[3]];          -- the second R 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: RECORD [inner: RECORD[f1, f2: INTEGER]];
> [i, j] ← r.inner;       -- 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: RECORD[f: INTEGER];
i: INTEGER;
ii: LONG INTEGER;
...
i ← r;        -- i ← r.f
ii ← r;       -- ii ← 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. Mesa is defined so that conformance is generally not recursive, even when no computation is required for conversion. Thus ARRAY *IndexType* OF RECORD[INTEGER] does not conform to either of the above types.

There is one other automatic conversion, dereferencing, that is applied only in special circumstances (section 3.4.4). It is never applied automatically to achieve type conformance.

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:

```
b: BOOLEAN;   n: CARDINAL;
n ← LOOPHOLE[b, CARDINAL];        -- to discover the representation
n ← LOOPHOLE[b];                  -- 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[f: INTEGER];
RR: TYPE = RECORD[ff: LONG INTEGER];
i: INTEGER;
ii: LONG INTEGER;
r1, r2: R;
rr: RR;
...
i = ii        -- LONG[i] = ii
r1 = r2       -- compared as records
r1 = i        -- r1.f = i
r1 = rr       -- LONG[r1.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,

```
ii ← i + r;              -- ii ← LONG[i + r.f]
ii ← LONG[i] + r;        -- ii ← LONG[i] + LONG[r.f]
```

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.

3.6. Determination of representation

This section discusses the rules used by Mesa for choosing between signed and unsigned versions of operations on single-precision numbers. 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 and vice versa. In both directions, the "conversion" amounts to an assertion that the value is an element of INTEGER ∩ CARDINAL. Currently, *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 ∩ CARDINAL or a **Primary** with an inherent type that is a subrange of INTEGER ∩ 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;   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;   m, n: CARDINAL;   s, t: [0..77777B];   b: BOOLEAN
-- the statements on each of the following lines are equivalent.
i ← m+n;  i ← INTEGER[m+n]                     -- unsigned addition
i ← j+n;  i ← n+j;  i ← j+INTEGER[n]           -- signed addition
i ← s+t;  i ← INTEGER[s]+INTEGER[t]            -- signed (overflow possible)
n ← s+t;  n ← CARDINAL[s]+CARDINAL[t]          -- unsigned (overflow impossible)
s ← s-t;  s ← CARDINAL[s]-CARDINAL[t]          -- unsigned (overflow possible)
b ← s-t > 0;  b ← INTEGER[s]-INTEGER[t] > 0    -- signed (overflow impossible)
i ← -m;   i ← -INTEGER[m]
```

```
i ← m+n*(j+n);   i ← INTEGER[m] + (INTEGER[n]*(j+INTEGER[n]))
n ← m+n*(j+n);   n ← m + (n*(CARDINAL[j]+n))
i ← m+n*(s+n);   i ← INTEGER[m+(n*(CARDINAL[s]+n))]

b ← s IN [t-1 .. t+1];  b ← INTEGER[s] IN [INTEGER[t-1] .. INTEGER[t+1]]
FOR s IN [t-1 .. t+1] ...;  FOR s IN [CARDINAL[t-1] .. CARDINAL[t+1]] ...
```

The following statements are incorrect because of representational ambiguities.

```
b ← i > n;   b ← i+n IN [s .. j]
SELECT i FROM m => ...;  t => ...; ENDCASE
```

CHAPTER 4.

ORDINARY STATEMENTS

Statements control the flow of execution and the disposition of data. 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:

```
Statement       ::= AssignmentStmt | IfStmt | SelectStmt | NullStmt |
                    Block | GotoStmt | LoopStmt | ExitStmt | ...
```

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

```
Expression      ::= ... | AssignmentExpr | IfExpr | 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.

A statement normally runs to completion, although it could be purposefully aborted for some reason. The discussion of execution control assumes normal statement completion unless otherwise stated.

Where does execution continue after a given statement? This is not obvious, in general, since that statement may be contained in some other statement. For simplicity, the discussion assumes that most statements occur in the middle of a hypothetical series of statements. Execution paths out of this series are described for each control statement, and resumption is then discussed in terms of a mythical "*Next-Statement*". *Next-Statement* represents nothing more than completion of a given statement; a statement may or may not exist at that point in actual practice.

The discussion of execution control is further simplified by denoting arbitrary statements by *Stmt-0*, *Stmt-1*, *Stmt-2*, etc. When shown in examples, these stand for statements whose exact nature is irrelevant.

4.1. Assignment statements

Syntax:

```
AssignmentStmt    ::= LeftSide ← RightSide |
                      Extractor ← RightSide
```

The **RightSide** must be an expression conforming to the left-hand side's type. 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 item may also be an extractor (section 3.4.5).

Examples:

```
i ← 3;   a ← b+c;
birthDay.month ← Apr;   birthTable[Tom].year ← 1955;
[mm, dd, yy] ← birthDay;      -- 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

AssignmentExpr ::= LeftSide ← 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 it as a separate statement:

```
x2 ← x1 ← x0 ← v;       -- set x0, x1, and x2 to the value in v
array[j ← j+1] ← x[i];  -- j is changed while changing the array component
```

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

```
x2 ← (x1 ← (x0 ← v));
```

Note that *x2* ← (...) is an assignment *statement*. The assignment expression, *x0* ← *v*, yields the value actually assigned to *x0*, and 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 value conversion) of the **RightSide**. An assignment operator has the *lowest possible precedence*. As a rule, therefore, when an assignment expression is embedded in another expression, enclose the assignment in parentheses.

Fine point:

In an embedded assignment such as the following:

```
a[k←k+1] ← b[k];
```

Mesa may evaluate the embedded assignment before using *k* on the right side. Such use of embedded assignments should be avoided.

The type of an assignment expression is actually that of the **RightSide**, when the **RightSide** is evaluated looking for a type assignable to that of the **LeftSide**. For example, in the case of variant records (section 6.4), the type of the assignment expression may be a bound variant while the **LeftSide** is not.

4.2. IF statements

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

```
IfStmt        ::= IF Predicate ThenClause ElseClause
Predicate     ::= Expression
ThenClause    ::= THEN Statement
ElseClause    ::= empty | ELSE Statement
```

A simple IF statement is shown below.

```
IF v = 0 THEN WriteString["Done."] ELSE v ← v-1;
Next-Statement
```

The BOOLEAN expression (*v* = 0) is called the **Predicate** for 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* ← *v*-1", 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 ← i + iDelta  ELSE i ← m;

IF winner ~= NIL THEN
    BEGIN        -- this Statement is a block (sec. 4.4)
    averageAge ← averageAge + winner.age;
    IF winner.party = Democratic THEN demoScore ← demoScore+1
    ELSE gopScore ← gopScore+1;
    END;             -- end of the ThenClause
Next-Statement
```

Note that a **ThenClause** *is not terminated by a semicolon when an* **ElseClause** *is present.*

If the **Statement** in a **ThenClause** is a second IF statement, then the outer IF may only have an **ElseClause** when 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
ELSE b ← -1;                                -- a < 0 means set b to -1
```

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, and let *Stmt-1* and *Stmt-2* represent arbitrary statements. Then the first statement following is executed as if it were written as the second:

```
IF p1 AND p2 THEN Stmt-1;               -- recommended form (see sec. 2.5.3)
Next-Statement

IF p1 THEN IF p2 THEN Stmt-1;           -- longer form
Next-Statement
```

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 which is an expression. Its syntax is similar to 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 max ELSE x/y;  -- avoid division by zero.
b ← IF a >= 0 THEN (IF a > 0 THEN 1 ELSE 0) ELSE -1;
```

When the first IF expression is evaluated, the **Predicate** (*y* = 0) is first evaluated. If TRUE, the expression in the **ThenClause** (i.e., *max*) is evaluated, and its value becomes the IF expression's value. If the predicate is FALSE, the **ElseClause** expression (i.e., *x*/*y*) is evaluated, and its value becomes the IF expression's value. The second example, like its counterpart in section 4.2, 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 as to type (possibly after type conversion, as outlined in section 3.6.2). 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 from a set of statements, one (or, possibly none) to execute, based on the relation between a given expression and expressions associated with each selectable statement. Thus, it permits multi-way control switching, as opposed to the two-way switching 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;
    ENDCASE;
Next-Statement
```

The next four sections cover various forms for SELECT, 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:

```
SelectStmt        ::= SELECT LeftItem FROM         -- (the head)
                      StmtChoiceSeries             -- (the arms)
                      ENDCASE FinalStmtChoice      -- (the foot)
                      | ...
LeftItem          ::= Expression
StmtChoiceSeries  ::= TestList => Statement ; |
                      StmtChoiceSeries TestList => Statement ;
FinalStmtChoice   ::= empty |
                      => Statement
TestList          ::= Test | TestList , Test
Test              ::= Expression |       -- no operator means equality
                      RelationTail
```

Example:

```
i: [0..5];
. . .
SELECT i FROM
   0   => i ← i+1;                          -- i=0
   <3  => BEGIN j ← i;  i ← i-1; END;   -- i=1 or i=2
   =5  => i ← 0;                            -- i=5
   ENDCASE => i ← 2;                        -- i=3 or i=4 (none of the above)
Next-Statement
```

The **LeftItem** for an ordinary SELECT statement is evaluated when the statement is first executed. A sequence of comparisons then follows. Each arm of the SELECT statement designates one or more **Tests** for the **LeftItem's** value. They are tested (i.e., they are evaluated, and then compared with the **LeftItem** value), in order from left to right, until a **Test** succeeds, or the **TestList** for that particular statement is exhausted. If a test succeeds, control is given immediately to the statement following that **TestList** (no further **Tests** are evaluated, even in that same list). If all **Tests** in a given arm fail, the next arm is tried. When a test succeeds, and its associated statement is executed, control then passes to *Next-Statement*: thus, at most one statement can be chosen in a given execution of a SELECT statement.

The type of the **Expression** in a **Test** must conform to the **LeftItem's** type. If a **Test** uses "IN **Subrange**", the subrange (which, recall is really a type constructor) must conform to the **LeftItem's** type:

```
i: [0..5];
. . .
SELECT i FROM
   0          => i ← i+1;                       -- i=0
   IN [1..2]  => BEGIN j ← i;  i ← i-1; END;    -- i=1 or i=2
   IN [3..4]  => i ← 2;                         -- i=3 or i=4
   ENDCASE    => i ← 0;                         -- i=5 (i.e., none of the above)
Next-Statement
```

A single SELECT arm may also specify more than one test:

```
SELECT i*j+k FROM
    1, IN [7..10]    => Stmt-1;              -- values: 1, 7, 8, 9, 10
    2, 5, > 10       => Stmt-2;              -- values: 2, 5, 11, 12, ...
    ENDCASE;
Next-Statement
```

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

```
PriorityState: TYPE = RECORD[i0, i1, i2, i3, i4, i5: BOOLEAN];
oldState, newState: PriorityState;
. . .
SELECT TRUE FROM    -- picks the first TRUE state:
    oldState.i0                 => Stmt-0;
    oldState.i1, newState.i0    => Stmt-1;
    oldState.i2, newState.i1    => Stmt-2;
    oldState.i3, oldState.i4    => Stmt-3;
    ENDCASE                     => Stmt-99;
Next-Statement
```

If this SELECT statement does not choose one of the first four choice statements, the final statement (*Stmt-99*) is chosen.

Fine points:

If the **LeftItem** expression has a sufficiently small subrange type and the SELECT arms specify constant values for comparison, the compiler can produce code using a "jump table" to efficiently choose one arm of the SELECT statement to execute. Actually, it is capable of doing this even if two or more short tables would be required because a large, complete table would have many components choosing the same statement.

If the left item and the first j right items are compile-time constants such that the j-th choice is always taken, then the compiler will not produce object code for the unused text of a SELECT.

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.
Next-Statement
```

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          -- (the head)
                    ExprChoiceList                -- (the arms)
                    ENDCASE => Expression         -- (the foot)
                    | ...*
ExprChoiceList  ::= TestList => Expression , |
                    ExprChoiceList TestList => Expression ,
```

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

For example:

```
pt: INTEGER;                      -- Point on a line.
lo, hi: INTEGER ← 0;              -- Bounds for a line segment, initially a null segment
. . .
PointPosition: TYPE = {leftMargin, rightMargin, inside, outside, degenerate};
pointIsAt: PointPosition;
. . .
pointIsAt ←
    IF lo > hi THEN degenerate ELSE
        (
        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
        );
Next-Statement
```

A SELECT expression is executed just as a SELECT statement, except that a selected arm yields a value, which is then taken as 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 (sec. 3.6.2).

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.

Fine point:

The other alternatives for **SelectExpr** apply to variant records and are discussed in chapter 6.

4.4. Blocks

A block is a way of packaging a series of statements so that they can be used where normally only a single statement would be syntactically correct. 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; sec. 4.4.2
                EnableClause        -- optional; sec. 8.2.1
                DeclarationSeries   -- optional; sec. 5.3
```

```
                         StatementSeries    -- the important part of the Block
                         ExitsClause        -- optional; sec. 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.

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

```
IF lo > hi THEN
    BEGIN -- Exchange lo and hi.
    temp ← lo;
    lo ← hi;
    hi ← temp;
    END;
Next-Statement
```

Note: A semicolon follows each statement in the **StatementSeries**, but is optional after the last one.

A block can introduce new identifiers with scope smaller than an entire procedure (or module) body, but there are several subtleties to consider. During the execution of a Mesa program, frames are allocated at the procedure and module level only. 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. 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 labelled 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

```
    ExitsClause    ::= empty |
                       EXITS ExitSeries |
                       EXITS ExitSeries ;          -- optional final semicolon
    ExitSeries     ::= LabelList => Statement |
                       ExitSeries ; LabelList => Statement
    LabelList      ::= Label | LabelList , Label
    Label          ::= identifier
    GotoStmt       ::= GOTO Label | GO TO Label
```

A simple example:

```
IF input.status # open THEN
    BEGIN
    . . .
    IF input.fileHandle = defaultInput THEN GOTO useDefault;
    . . .                               -- processing for non-default file
    IF input.fileNumber = ttyNumber THEN GOTO alsoUseDefault;
    IF input.length = 0  THEN GOTO newFile;
    . . .                               -- compute number of pages in the file
    EXITS
        useDefault, alsoUseDefault =>   -- multiple labels are allowed
            BEGIN input ← ttyInput; pages ← maxPages END;
        newFile => pages ← 0;
    END;                                -- end of the ThenClause and the IF statement
Next-Statement
```

The **Labels** in this example are *useDefault, alsoUseDefault,* 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 goes immediately to the statement labelled with the identifier used in the GOTO. As soon as the labelled statement completes, control goes to *Next-Statement* (unless a labelled statement itself contains a GOTO, which is possible -- see below). If control is not interrupted by a GOTO, it will pass naturally to *Next-Statement* after the last statement in the main body of the block (i.e., the one just before EXITS) is finished.

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

```
BEGIN                                   -- outer block
. . .
    BEGIN                               -- inner block
    . . .
    IF i = iMax THEN GO TO endOfArray;  -- jump to end of outer compound
    . . .
    END;                                -- end of inner
. . .
i ← i+1;
EXITS
    endOfArray => i ← 0;
END;                                    -- end of outer
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 label in a number of blocks. If this is done in nested blocks, a GOTO naming that identifier will always go to the closest statement with that label. Generally, using the same label in nested blocks is a bad idea.

Since Mesa allows declarations in any block, it is possible to have a nested procedure (section 5.6) declared syntactically within the scope of the **Labels** of an outer block. The exection of a GOTO with one of these outer labels as a target would require unwinding the stack some arbitrary number of levels. For this reason, 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[];
. . .
EXITS
    panicExit => ...
END;
```

The desired result of the above illegal program could be achieved with the following program (see chapter 8 for a description of signals and catchphrases):

```
BEGIN
Panic: SIGNAL = CODE;
. . .
p: PROCEDURE =
    BEGIN
    . . .
    SIGNAL Panic;
    . . .
    END;

. . .
p[ !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 => GOTO outOfData;
    END;                                    -- end of inner
. . .
EXITS
    outOfData => Stmt-99;
END;                                        -- end of outer
Next-Statement
```

4.4.2. OPEN *clauses*

An OPEN clause allows one to abbreviate references to a record object that would normally require many characters to name (e.g., *candidateList*[*tableOfObjects*[*i*]]). It does this by giving short names as synonyms for more complex names. The *scope* of an OPEN clause (the portion of the program over which the synonym can be used) is the body of a block, a procedure body, or a loop body, including the optional exits clause (sec. 4.5). Its syntax is

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

The **OpenItem** which uses an **AlternateName** allows one to use a simple identifier in place of an expression to designate some record object. For example, the two blocks after the following declaration are equivalent:

```
PersonChain: TYPE = RECORD [p:POINTER TO Person, next: POINTER TO PersonChain]
candidateList: POINTER TO PersonChain; -- Person is defined in sec. 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
```

When the **AlternateName** form is used, the alternate identifier is *always* a record type, even if the object that it renames is a pointer to a record, or an array component.

The form of **OpenClause** without an **AlternateName** allows one to access the fields of a record object (the expression must conform to a record type in this case) as though they were simple variables. For example, using this feature in the above example allows us to elide even the "*c*."s in it:

```
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 by the 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, *myRecord* and *myRecord.subRecord*. An unqualified occurrence of *x* is taken to be the *x* component of the *rightmost* opened record (*myRecord.subRecord*). To refer to an earlier opened record, explicit qualification is necessary (the **AlternateName** form should be used).

```
i, j: INTEGER;
RecordType: TYPE = RECORD
   [
        a, b, x: INTEGER,
        subRecord: RECORD[x, y: INTEGER]
        ];
    myRecord: RecordType;

    BEGIN OPEN r1: myRecord, myRecord.subRecord;
    i ← r1.a + r1.b * r1.x;
    j ← x-y;
    END;
    Next-Statement
```

The above block is equivalent to:

```
    BEGIN
    i ← myRecord.a + myRecord.b * myRecord.x;
    j ← myRecord.subRecord.x - myRecord.subRecord.y;
    END;
    Next-Statement
```

Fine points:

> The range of text affected by an **OpenItem** actually starts after that item (but stops before the **ExitsClause** if there is one). The **OpenClause** itself may use implied qualification or alternate names (from earlier **OpenItems**).
>
> **OpenItems** are essential when dealing with variant records, discussed in chapter 6.
>
> Opened expressions are evaluated *at each use*, whether used implicitly or explicitly under an alternate name. This can be helpful when the opened expression involves a pointer or an array index that is periodically updated -- the latest value will be used for qualification.
>
> To avoid confusion, however, it is recommended that such 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., apply to the same data objects throughout these statements.

4.5. Loop statements

Loops make the world go 'round. In Mesa, a loop is a statement containing a series of statements which 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 contained the following loop statement:

```
    UNTIL n = 0
        DO
        gcd ← n;
        n ← m MOD n; -- n gets remainder of "m/n"
        m ← gcd;
        ENDLOOP;
    Next-Statement
```

"UNTIL *n*=0" is the *loop control* for this loop. A variety of loop controls are offered 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 [1..10)
        DO
        Stmt-1;
        Stmt-2;
        . . .
        ENDLOOP;
    Next-Statement
```

This will execute 9 times, with *i* starting at 1, then becoming 2, and so on, until (but not including when) *i* is 10. The portion between DO and ENDLOOP is the body of a loop, and is very similar to the body of a block.

The formal syntax for loops is

```
LoopStmt      ::= LoopControl          -- optional; may be empty
                  DO
                  OpenClause           -- optional (sec. 4.4.2)
                  EnableClause         -- optional (sec. 8.2.1)
                  StatementSeries
                  LoopExitsClause      -- optional; may be empty
                  ENDLOOP
```

The body of a loop may contain an **OpenClause** and an **EnableClause** just as in a block. The next section discusses the various forms of **LoopControl**, and the one after that, the **LoopExitsClause** and GOTOs in loops.

4.5.1. Loop control

The syntax for **LoopControl** is

```
LoopControl    ::= IterativeControl ConditionTest  -- either may be empty

ConditionTest  ::= empty | WHILE Expression | UNTIL Expression
```

If both the **IterativeControl** and the **ConditionTest** are missing from a loop, it will repeat forever. A GOTO may be used to terminate any loop from within, however, so this degenerate form does have some uses. Nevertheless, the shortest infinitely running Mesa program needs only the single statement:

```
DO ENDLOOP;            -- empty body and LoopControl
```

We deal with **IterativeControl**s below, but since any loop may also be controlled by a **ConditionTest** (either together with an **IterativeControl**, or by itself), we treat it separately first.

All **LoopControl**s run *before* each execution of the body to avoid entering the loop even once if it should not be. If a loop uses a **ConditionTest**, the Boolean expression in the test is completely reevaluated each time before entering the loop body. If the **ConditionTest** *succeeds*, the body of the loop is entered; if it *fails*, the loop is finished (we say: "*terminates conditionally*") and control continues at the *Next-Statement*. A WHILE test succeeds if the value of the expression is TRUE, as in the following example (it might be useful to read WHILE as "As long as"):

```
i ← 1;                        -- this statement is not part of the loop
WHILE i < 10
    DO
    . . .
    i ← i+1;                  -- increment i
    . . .
    ENDLOOP;
Next-Statement
```

In this example, *i* will have 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 only the statement shown changes the value of *i*. An UNTIL test succeeds if the value of its expression is FALSE:

i.e., it is the opposite of WHILE. The following loop is equivalent to the one using WHILE:

```
i ← 1;                        -- this statement is not part of the loop
UNTIL i >= 10
    DO
    . . .
    i ← i+1;                  -- increment i
    . . .
    ENDLOOP;
Next-Statement
```

An **IterativeControl**, which may be followed by a **ConditionTest**, specifies a way of counting the repetitions of a loop and of terminating the loop should the counting reach a specified, final value. It may also assign the current count value to a **ControlVariable** so that statements in the body may use the count value for computation. A loop which finishes by satisfying the implicit test associated with an **Iteration** or a **Repetition** is said to have *terminated normally*.

IterativeControl	::=	**empty** | **Repetition** | **Iteration** | **Assignation**
Repetition	::=	THROUGH **LoopRange**
Iteration	::=	FOR **ControlVariable** **Direction** IN **LoopRange**
LoopRange	::=	**SubrangeTC** | **TypeIdentifier** | BOOLEAN | CHARACTER
Direction	::=	**empty** | INCREASING | DECREASING
Assignation	::=	FOR **ControlVariable** ← **InitialExpr** , **NextExpr**
ControlVariable	::=	**identifier**
InitialExpr	::=	**Expression**
NextExpr	::=	**Expression**

The only form of **IterativeControl** which does not include a **ControlVariable**, the **Repetition**, allows a program to specify how many times a loop body should be repeated by specifying a **LoopRange**. For example,

THROUGH [1..100] DO *body* ENDLOOP

Will execute the *body* 100 times, and the count value will run through the range 1, 2, 3, ..., 99, 100. If the interval used had been [1..100) instead, the *body* would only have been repeated 99 times. The bounds of intervals in a loop range can be general expressions and do not have to be compile-time constants (as they do in a normal **SubrangeTC).**

An subrange of an enumerated type may also be used as the **LoopRange** in a **Repetition.** Its type should be self-evident. This requirement may be met by qualifying the type constructor, as in the example following:

THROUGH *MonthName*[*Jun* .. *Aug*] DO . . . ENDLOOP

The body of this loop would be repeated 3 times because the subrange covered by bounds *Jun* and *Aug* in type *MonthName* contains 3 values: *Jun*, *Jul*, and *Aug*.

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

THROUGH [*low*..*high*] WHILE *lineIsConnected* DO . . . ENDLOOP

Normal termination occurs after *high*-*low*+1 passes or conditional termination occurs before any pass where *lineIsConnected* proves FALSE, whichever comes first. Note that if *low* >

high, the interval [*low*..*high*] is empty and *no* passes occur.

Iteration and **Assignation,** the two forms of **IterativeControl** which include a **ControlVariable,** begin with the keyword FOR. The **ControlVariable** must be a variable declared separately in the program. Its type will be the target type for the various expressions whose values may be assigned to it in the course of executing the loop.

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 INCREASING) 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 **ControlVariable** is assigned the current count value each time around the loop, and also before the loop terminates.

If the **LoopRange** specifies an empty interval, the loop body is not executed at all, the **ControlVariable** is not assigned to, and therefore retains whatever value it had before the loop. If the **LoopRange** is not empty, the body will be executed. When a loop terminates *normally*, the **ControlVariable**'s final value is *not* defined. The only way to ensure that the **ControlVariable**'s final value is the last one it had when the loop body executed is to terminate the loop *conditionally* (e.g., using EXIT or GOTO--sec. 4.5.2). The following example of an **Iteration** control shifts the components of an array left one position, leaving the rightmost element unchanged:

```
FOR i IN [0..LENGTH[vec]-1)          -- assume that LENGTH[vec] > 0
   DO
   vec[i] ← vec[i+1];                -- "Left-shift" vec's elements.
   ENDLOOP;
Next-Statement
```

The above loop control increases the control variable's value in the specified subrange. It could have been written equivalently as:

```
FOR i INCREASING IN [0..LENGTH[vec]-1) DO vec[i] ← vec[i+1]; ENDLOOP;
```

There is also a DECREASING form for shifting *vec*'s components one place to the right:

```
FOR i DECREASING IN [0..LENGTH[vec]-1)
   DO
   vec[i+1] ← vec[i];                -- "Right-shift" vec's elements.
   ENDLOOP;
```

In this case *i* is set to the value LENGTH[*vec*]-2 the first time through the loop and decremented by one for each subsequent pass. The last time around this loop, *i* will have the value 0.

Note: Bounds expressions for a controlling subrange are evaluated *exactly once*, at the beginning of a loop statement's execution. Consequently, if variables used in the bounds expressions are altered during a pass, this does not affect the number of passes. When an **Iteration** is combined with a **ConditionalTest** in a single loop control, the **ControlVariable** is updated and tested before the **ConditionTest** is evaluated and tested.

In an **Assignation**, the **InitialExpr**'s value is assigned to the **ControlVariable** for the first pass. On each subsequent pass, the **NextExpr** is reevaluated and assigned to it. There is no implicit test associated with an **Assignation** as there is for an **Iteration**; thus, the user must either use a GOTO (sec. 4.5.2) to terminate the loop or include a **ConditionTest** in the **LoopControl** with the **Assignation**. This form is useful for scanning down a list structure, as in the following example (note, as with an **Iteration**, the **ControlVariable** is updated before the **ConditionTest** is evaluated and tested each time around the loop):

```
ListPtrType: TYPE = POINTER TO ListType;
listPtr, headOfList: ListPtrType;
ListType: TYPE = RECORD
    [
    listValue: SomeType,
    next: ListPtrType              -- either NIL (end of list) or pointer to next element
    ];
. . .
FOR listPtr ← headOfList, listPtr.next UNTIL listPtr=NIL
    DO
    . . .
    ENDLOOP;
Next-Statement
```

Fine points:

A control variable can be altered within the 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). This undermines the credibility of the loop control.

4.5.2. GOTO*s,* LOOP*s,* EXIT*s, and loops*

A loop may be *forcibly* terminated by a GOTO (or an EXIT, see below) from within it to its **LoopExitsClause**. This clause serves the same purpose as the **ExitsClause** in a **Block**; there are just four differences:

(1) The **LoopExitsClause** is bracketted by REPEAT on the top and ENDLOOP on the bottom instead of EXITS and END;

(2) The **LoopExitsClause** may contain a final statement labelled 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 LOOP, which causes the remaining statements in the **StatementSeries** to be skipped, and the control variable, if any, to be updated and tested. In other words, it is equivalent to a GOTO the bottom of the loop.

(4) 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**.

Syntax equations:

```
LoopExitsClause       ::= empty | REPEAT LoopExits

LoopExits             ::= ExitSeries |
                          ExitSeries ;     |
                          FinishedExit     |
                          ExitSeries ; FinishedExit

FinishedExit          ::= FINISHED => Statement     |
                          FINISHED => Statement ;

LoopCloseStatement    ::= LOOP

ExitStmt              ::= EXIT
```

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

```
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;
```

Saying LOOP is equivalent to a GOTO with a target just before the **LoopExitsClause** (if any).

The loop example from the previous section used to illustrate **ConditionTests** may be rewritten using a GOTO and a **LoopExitsClause** as:

```
i ← 1;
    DO
    IF i >= 10 THEN GOTO quit;          -- first statement in the body
    . . .
    i ← i+1;                            -- increment i
    . . .
    REPEAT
        quit => NULL;                   -- do nothing; just get out of the loop
    ENDLOOP;
Next-Statement
```

Forcible loop termination in which the statement in the exits clause is NULL is common. The EXIT statement simplifies this case by not requiring a labelled statement in the **LoopExitsClause.** In fact, no **LoopExitsClause** need be present at all. The above example, using EXIT instead of GOTO, looks like this:

```
i ← 1;
    DO
    IF i >= 10 THEN EXIT;               -- first statement in the body
    . . .
    i ← i+1;                            -- increment i
    . . .
    ENDLOOP;
Next-Statement
```

An EXIT is a little less powerful than the more general GOTO, however. For instance, if one had loops nested within one another and wanted to exit from both, EXIT could not be used because it would only terminate the inner loop. A GOTO could be used because it may completely abandon the inner loop and 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 could appear in any **ExitsClause** (provided there was an outer loop) and would cause forcible termination of the closest surrounding loop.

The example below shows an **Assignation** form of **IterativeControl** which is not combined with a **ConditionTest.** Instead, the loop is forcibly terminated by an EXIT statement.

```
BufIndexType: TYPE = [1..max];
buf: ARRAY BufIndexType OF INTEGER;
i, x: BufIndexType;
. . .
FOR i ← x, (IF i = max THEN 1 ELSE i+1)        -- Starting at point x,
    DO
    Stmt-1;                                    -- do something and then
    IF buf[i] = 0 THEN EXIT;                   -- quit on a "clear" entry, or
    buf[i] ← 0;                                -- clear until one is found.
    ENDLOOP;
Next-Statement
```

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

Sometimes one must detect normal (vs. abnormal) completion, perhaps to take some "finishing" action. A final labelled statement with the label FINISHED (which may *not* appear as the identifier in a GOTO) provides this facility. For example,

```
UNTIL currentChar=blank DO
    . . .
    IF currentChar NOT IN ['A..'Z] THEN EXIT;
    . . .
    IF charCount > limitForReservedWords THEN GOTO syntaxError;
    . . .
    REPEAT
        syntaxError   => Stmt-99;
        FINISHED      => currentChar←GetNB[];     -- Get next non-blank character.
    ENDLOOP;
Next-Statement
```

The FINISHED exit is taken if and only if the loop terminates normally or conditionally (i.e., when *currentChar* is *blank*, in the case above). Note that if the EXIT statement is executed, the FINISHED statement is *not* executed.

CHAPTER 5.

PROCEDURES

A procedure is a means of packaging code so that it can be used (*called*) by other parts of a program. A caller can communicate with a procedure by passing *arguments* to it when calling it. Similarly, when a called procedure finishes, it can return *results* to its caller. Then the caller resumes execution from the place that the call was made.

The parameters of a procedure may have different types. In fact, they constitute a *parameter record* type for the procedure. When calling the procedure, the arguments are assembled into an *input record* using a constructor (sec. 3.3.4.). Consider the following procedure call:

```
Pxy[23, FALSE];          -- naming a procedure is all that is needed to call it
```

This constructs an input record with two arguments and then calls procedure *Pxy*, passing it the input record.

A procedure may return values to the point of its call. These *results* actually constitute a *result record* type. There may be any number of results, and their types may differ. Results are assembled into an *output record* by the procedure as it RETURNs control to its caller.

Procedures are declared very much like other data types. The type of any procedure depends upon the types of its parameter and result records. We say these are "empty" records if a procedure has no parameters or no results. A few of the possible procedure types are shown below:

```
PROCEDURE                                     -- takes no arguments; returns no results
PROCEDURE[x: INTEGER, flag: BOOLEAN]          -- takes two arguments
PROCEDURE RETURNS[i: INTEGER]                 -- returns a single value
PROCEDURE RETURNS[i: INTEGER, b: BOOLEAN]     -- returns two results
PROCEDURE[x: INTEGER] RETURNS[y: INTEGER]     -- takes and returns one value
```

These are all different procedure types. This enables the compiler to check that a proper input record is given for each procedure call (i.e., arguments conform to matching parameters) and that the output record is used correctly (in the text surrounding the procedure call).

The declaration of an *actual procedure* has a fixed form **Initialization** which gives the source code for that procedure. For example,

```
SetAllSwitches: PROCEDURE[setting: SwitchSettingType] =
    BEGIN                              -- this is the source code for the procedure
    i: SwitchRange;                    -- declares loop control variable i
    FOR i IN SwitchRange DO switch[i] ← setting ENDLOOP;
    END;
```

This closely parallels the declaration of an ordinary variable with = initialization (e.g., *octalRadix*: CARDINAL = 8;). Of course, the initializing "**Expression**" is special (i.e., everything from BEGIN to END); it is called the *body* of the actual procedure. The other forms of initialization may also be used, and this allows one to have procedure variables which may be changed by the program to access different actual procedures. *Procedures are data objects just like any other piece of Mesa data.* It is not the code for a procedure that is data, but rather the information about where that code resides, which allows it to be invoked.

Like a block, a procedure body may contain *declarations* (they must precede the first statement in the body). These declare *local variables* for that procedure, variables which are created when the procedure is called, which may be directly accessed only from within it, and which are destroyed when the procedure returns. (Thus, *i* and *setting* are both local to *SetAllSwitches*.) This way of allocating variables for procedures makes any Mesa procedure capable of being called recursively and its code capable of being used in a reentrant fashion.

A procedure body may also access variables declared outside of the actual procedure (e.g., *switch* in this case). These are *non-local* variables in this procedure: they exist longer than any single invocation of the procedure and are defined in the program around it.

Mesa also has extensive facilities for supporting the separate compilation of packages of procedures and data; these packages are called *modules* (chapter 7). Some of these facilities allow one module to name and use the procedures in another and still check for valid parameter and return record use at compile time.

The foregoing discussion is only an introduction to procedures. The rest of chapter 5 covers this topic more completely.

5.1. Procedure types

Procedure types are specified by the type constructor, **ProcedureTC**, following:

```
ProcedureTC     ::= PROCEDURE ParameterList ReturnsClause
ParameterList   ::= empty | FieldList
ReturnsClause   ::= empty | RETURNS ResultList
ResultList      ::= FieldList
```

This type constructor is used when declaring actual procedures or procedure variables, but it may also be used wherever Mesa allows a **TypeSpecification**. This means that procedure types can be employed in constructs such as:

A defined type:

```
ListProc: TYPE = PROCEDURE[in: List] RETURNS[out: List];
GoForth, GoBack, AddNewNode, RemoveNode:  ListProc;
```

A field list (notice parameter *GetNewPage* below):

```
AllocateBlkInPage: PROCEDURE
    [
    blkSize: [0..pageSize), currentPage: POINTER TO Page,
    GetNewPage: PROCEDURE RETURNS[POINTER TO Page]
    ]
      RETURNS[blk: POINTER TO Blk, page: POINTER TO Page];
```

A pointer declaration:

```
stdError: POINTER TO PROCEDURE[error: ErrorType];
```

The **ParameterList** and the **ResultList** define unique record types: if either is missing, its record type is "empty". A procedure type is fully determined by its input and output record types. These records, unlike regular records, are not *packed*: i.e., every component is *aligned* (begins on a word boundary) in order to allow efficient passing of arguments and results.

ParameterLists and **ResultLists** are **FieldLists** (sec. 3.4.1). When a call is made on a procedure, the arguments are packaged in a record. Therefore, a procedure call may use all the syntax for record constructors in passing arguments, including specifying components (arguments) by keyword or by position.

If the **ReturnsClause** in a **ProcedureTC** is not empty, then its **ResultList** specifies the number and types of the results returned by a procedure of that type. It may be a named or an unnamed **FieldList** (sec. 5.3.1 on the RETURN statement discusses how it is used). Here are some examples of **ProcedureTCs**:

```
Square: PROCEDURE[i: INTEGER] RETURNS [CARDINAL]
InitBuffer: PROCEDURE[buf: Buffer, from: CARDINAL, to: CARDINAL, value: CARDINAL]
Direction: TYPE = {up, down};        -- a type used in the following type constructor
Sort: PROCEDURE[a: DESCRIPTOR FOR ARRAY OF INTEGER --(sec. 6.2)--, order: Direction]
```

5.2. Procedure values and compatibility

Values of a procedure type are possible in Mesa: one may have procedure variables, arrays of procedures, records with components which are procedure values, and procedures with procedure parameters or results. A procedure value is really a complex pointer which can be used to call the procedure it represents.

The fundamental operations, ←, =, and # may be applied to two *conforming* procedure values. Procedure types conform if their **ParameterLists** and their **ResultLists** are *compatible*, which means that their respective components must have equivalent (not just conforming) types. Moreover, if both **ParameterLists** (or **ResultLists**) are named, then the names must match (if one or both are unnamed, no such requirement applies, of course). Thus, the two following procedures are compatible:

```
Atype: TYPE = INTEGER;  Btype: TYPE = INTEGER;  -- two compatible types
ActualProc: PROCEDURE[p: Atype] RETURNS[INTEGER] =
    BEGIN                          -- this is the body of this actual procedure (next section)
    RETURN[p];
    END;
BProc: PROCEDURE[p: Btype] RETURNS[INTEGER];
```

Mesa allows the assignment, *BProc←ActualProc* (we then say that *BProc* is *bound* to *ActualProc*) because their **ParameterList** and **ResultList** types have equivalent component types (INTEGER - recall that equivalence ignores type identifiers).

Fine points:

Consider the following declaration:

```
IntegerProc: PROCEDURE[i: INTEGER] RETURNS[INTEGER];
```

The assignment "*IntegerProc ← ActualProc*" is also legal because the name of the component in a single-element record is ignored when checking for type compatibility.

A procedure variable may be declared with = initialization. For the example below, assume *largeMemory* is a compile-time constant whose value is TRUE:

```
    ProcVar: PROCEDURE =  IF largeMemory THEN FastProc ELSE SmallButSlowProc;
```

If *FastProc* is the identifier of an actual procedure, then *ProcVar* is indistinguishable from that actual procedure. In this case, we call *ProcVar* a "procedure constant", using "actual procedure" only when a procedure body is specified.

5.3. Declaring actual procedures

An actual procedure declaration looks like the declaration of a procedure variable followed by a special kind of = initialization, a **ProcedureBody**. The **ProcedureTC** determines the procedure's type, as covered in the previous section (actually, any **TypeSpecification** equivalent to a **ProcedureTC** may be used). **ProcedureBody** is a special form of initialization defined as follows:

```
Initialization      ::= . . . | = ProcedureBody | ← ProcedureBody
ProcedureBody       ::= Block                    -- see sec. 4.4
Block               ::= BEGIN
                        OpenClause
                        EnableClause
                        DeclarationSeries
                        StatementSeries
                        ExitsClause
                        END
DeclarationSeries   ::= empty | DeclarationSeries declaration
```

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

Only a procedure initialized with = to a **ProcedureBody** is called an *actual procedure*: its meaning cannot change because it cannot be assigned to. If, however, it is initialized with a **ProcedureBody** but with ← initialization, its value can be changed by assignment, and it is considered a procedure variable.

Besides ordinary statements, a procedure body can contain RETURN statements (described in the next section). Here are some simple examples of actual procedures:

```
Square: PROCEDURE[i: INTEGER] RETURNS [CARDINAL] =
   BEGIN RETURN [i*i] END;

SumArray: PROCEDURE[a: DESCRIPTOR FOR ARRAY OF INTEGER] RETURNS [sum: INTEGER] =
   BEGIN              -- returns the value of a[0]+a[1]+...+a[LENGTH[a]-1] (sec. 6.2)
   i: CARDINAL;       -- a single local variable declaration
   sum ← 0;           -- first statement in StatementSeries
   FOR i IN [0..LENGTH[a]) DO sum ← sum+a[i] ENDLOOP;
   END;               -- there is an implicit RETURN at the END if one is not given
```

A **ProcedureBody** defines a *scope* for declarations: i.e., identifiers declared in it are local to the procedure and are unknown outside it. There must be no duplicates among the names in a procedure's **ParameterList**, **ResultList**, or local variables. Names in the **ParameterList** can be used to write a keyword constructor (sec. 3.3.4) when calling a procedure. Similarly, names in the **ResultList** can be used in keyword extractors (sec. 3.3.5) and as qualifiers (sec. 3.3.3) to access the results returned by a procedure. When called, the values of parameters and named results act just like local variables within the scope of the procedure. A **ParameterList** for an actual procedure should be a *named* field list so that the procedure body can reference the parameters.

5.3.1. RETURN *statements*

There are two basic forms of RETURN statements: RETURN and RETURN followed by a constructor. When either form is executed, control transfers back to the point at which the procedure was called. In addition, the RETURN can supply results in the form of a constructor conforming to the procedure's **ResultList** type:

```
        ReturnStmt      ::= RETURN | RETURN [ ComponentList ]
```

There may be any number of RETURN statements in an actual procedure. The form of a RETURN statement depends upon the **ReturnsClause**. There are three cases:

no **ReturnsClause** (empty output record)
an *unnamed* field list as the **ResultList**
a *named* field list as the **ResultList**

If *no* **ReturnsClause** is given, then the RETURN statements must consist of RETURN (alone). An explicit RETURN statement can be omitted at the end of the procedure; Mesa will provide an implicit RETURN there.

If an *unnamed* field list is used for the **ResultList**, then the RETURN statements must include a positional constructor. This constructor must match the field list perfectly, with one **Expression** component for every field (*omissions or elisions are not allowed*). In this case, Mesa will not supply an implicit RETURN at the end of the procedure, so the user must supply at least one in the **ProcedureBody**; if not, it is an error.

Fine point:

In this case, the compiler will insert code that raises the unnamed ERROR (see sec. 8.1) if control reaches the end of the procedure.

If the **ResultList** is a *named* field list, then the user may employ RETURN statements with or without constructors, using either positional or keyword notation. If no constructor is used, the current values of the named result variables will be used as the return values. A RETURN statement is optional at the end of the procedure; if omitted, Mesa will provide an implicit RETURN, and use the result variables to construct the output record. An example is shown below:

```
ReturnExample: PROCEDURE[option: [1..4]] RETURNS[a, b, c: INTEGER] =
    BEGIN
    a←b←c←0;
    SELECT option FROM
        1  => RETURN[a: 1, b: 2, c: 3];         -- keyword parameter list
        2  => RETURN[1, 2, 3];                  -- positional version of option 1
        3  => RETURN;                           -- a=b=c=0
        ENDCASE => b←4;
    c←9;
    END;                                        -- implicit RETURN[a: 0, b: 4, c: 9]
```

5.4. Procedure calls

Procedures that return results must be called from within **Expressions** which use the results in some way. Such a *function reference* is a valid **Expression**. Procedures that do not return results are used in call statements. The syntax for calling a procedure is

```
        CallStmt        ::= Variable | Call
        Call            ::= Variable [ ComponentList ] | . . .
```

where the **Variable** has some procedure type. Other forms of **Call** are discussed in chapter 8. These specify "catch phrases" for dealing with signals (or errors) that may be generated because of the call.

A procedure that does not return results is called by simply writing a **CallStmt** as a statement by itself. For example,

```
SocSecNums: ARRAY [1..100] OF SocialSecurityNumber;
. . .
Sort[DESCRIPTOR[SocSecNums], up]; -- DESCRIPTOR defined in sec. 6.2
```

A call statement is ordinarily used to obtain *side effects*. Most often, these take the form of changes to data that are not merely local to the invoked procedure, but they may also take the form of input or output for some device such as a display or a disk. A function may also have side effects, as well as returning 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 output record to an empty extractor:

```
[] ← F[x];                          -- Call F and discard its output record.
```

There are two ways to write a call that supplies *no arguments*: without a constructor or with an empty one "[]". Generally, these are equivalent, but you must use the bracketed form for function references. This will force the call to occur. Without the brackets, type determination may, in rare cases, cause the procedure type rather than the output record type to be used and therefore not actually call the procedure. For example, consider the two procedure variables in the following:

```
DoYourThing: PROCEDURE RETURNS[PROCEDURE];
DoMyThing:   PROCEDURE RETURNS[PROCEDURE];
. . .                  -- here the program assigns values to the procedure variables
IF DoYourThing=DoMyThing THEN . . .        -- compare the procedure variables
. . .
IF DoYourThing[]=DoMyThing[] THEN . . .    -- compare their results
```

At the time a call is executed, a specific *activation* is executing, the *caller's* activation. (A new activation is created when invoking the actual procedure, the *callee's* activation. This is discussed in the next section.) The caller's activation consists of:

a *code body*: The body of object code containing the code for the caller. A code body is constant data; it doesn't change during the course of execution.

a *frame*: A record containing all data local to this activation.

an *execution state*: The values in machine registers pertaining to the caller (for instance, the program counter).

Initially, the caller's activation is *running*, i.e., currently executing instructions on the machine. Only one activation can be running at any one time. Since a new activation (the callee's) will soon be running, the caller's activation is *suspended* by storing the machine's execution state in its frame.

An important consequence of this structuring of procedure control is that all Mesa procedures are inherently capable of being *recursive* and *reentrant*. The following recursive procedure could be used inside the *BinaryTree* module example (sec. 5.6.1):

```
CountNodes: PUBLIC PROCEDURE[here: Node] RETURNS[n: INTEGER] =
   BEGIN
   n ← IF here=NIL THEN 0 ELSE 1 + CountNodes[here.left] + CountNodes[here.right];
   END;          -- implicit RETURN[n] here
```

A call on this procedure is shown below:

```
totalNumberOfNodes ← CountNodes[TheRoot[]];
```

5.4.1. The mechanics of procedure calls

Invoking a procedure causes the following sequence of low-level events to occur (of course, most of these actions merge in the actual implementation; many fewer machine cycles are actually required to call a procedure than may seem apparent from this description):

The caller prepares an argument record to pass to the procedure.

The caller gives up control (after preparing to resume execution when the called procedure returns).

A frame is allocated and initialized for the new activation (this includes storing in the new frame a ***return link***, a pointer to the caller's frame for returning to later).

The code for the procedure begins executing and immediately stores the component values of the argument record into local variables corresponding to its named parameters. Next, any initialization of local variables is done, including those with "=" initialization in which the **Expressions** were not compile-time constants. Then the first executable statement begins.

The new frame contains *all* data local to this actual procedure: the parameters, the procedure's local variables, and its result variables, if any.

5.4.2. 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:

```
InRec: RECORD[arg1: Type1, arg2: Type2, ... ];
param1: Type1;
param2: Type2;
. . .
[param1, param2, ... ] ← InRec[arg1: val1, arg2: val2, ... ];
```

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 to a procedure need only *conform* to its parameter, just as for assignment.

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*).

5.4.3. Termination and results

Procedures terminate by executing a RETURN (perhaps an implicit RETURN inserted by the compiler just before the END of a procedure body). The sequence of low-level events involved in a procedure returning control and possibly returning results back to its caller are as follows (the sequence for a call is described in sec. 5.4.1):

The procedure prepares a result record to pass to its caller, if it gives results.

The return link (which points to the caller's frame) is saved, the procedure is deactivated, and the storage for its frame is reclaimed.

Using the saved return link, the caller is resumed by loading the machine state from its frame (including the program counter). Code at the call point uses the result record in whatever way is necessary, and then releases it.

The results of a procedure may be assigned to variables, either by simple assignment or by using an extractor for multiple-component result records, or the result may be used as part of an enclosing **Expression**. The following example uses the result of *G* as the argument for a call on *F* (note that this would *not* be legal if *G* returned a two-component result):

```
F: PROCEDURE[in: SomeType];
G: PROCEDURE[y: OtherType] RETURNS[out: SomeType];
. . .
F[G[x]];  -- the single component in G's output record becomes the single
             element in F's input record
```

If a procedure returns a single-component output record, the component may be accessed directly because of the automatic coercion from single-component record to its single component:

```
SimpleResult: PROCEDURE RETURNS[result: INTEGER] =
    BEGIN
    x, y: INTEGER;
    RETURN[x+y];
    END;
. . .
j ← SimpleResult[].result;    -- (selection)
[i] ← SimpleResult[];         -- (extraction)
i ← SimpleResult[]+64;        -- (access component directly)
```

In the above example, RETURNS[*result*: INTEGER] defines a unique record type for the output record. You could not assign the output record, as such, to a variable whose type was compatible; assignment demands conformance, and between two record types the only conformance is complete equivalence (section 3.6.2). If you need to return a record value (which is perfectly legal), have the procedure return a single-component output record where that component is a *record* of that type.

Since the result of a procedure is a record, it may be assigned to an extractor. The following procedure returns three results and may be used as indicated in the statements following it:

```
ComputeDate: PROCEDURE[i: INTEGER]
                        RETURNS[mm: MonthName, dd: [1..31], yy: [1900..2000]] =
    BEGIN
    . . .
    RETURN [Sep, 22, 1976];
    . . .
    END;
aDay, nextDay: [1..31];  aMonth: MonthName;  aYear: [1900..2000];
. . .
aDay ← ComputeDate[1900].dd;                   -- get day part only
[mm: aMonth, dd: aDay] ← ComputeDate[1900];    -- assign results by extractor
nextDay ← (ComputeDate[1900].dd+1) MOD 32;  -- select one component
```

5.5. A package of procedures

This section contains an example of a simple module, *BinaryTree*, which is designed to create and manage a simple data base structured as a binary tree. It is typical of the ways in which related procedures are packaged together. The example gives a broad view of procedure syntax and semantics, touching only lightly on specific issues.

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; a sophisticated data base would surely contain more complex objects. When the program is first loaded, the tree is empty, and any calls to *SeekValue* to see if a value is in the tree will result in a count of zero being returned.

The nodes in this particular binary tree are records with four components:

value	-- an integer value (whose purpose we will ignore),
count	-- the number of duplications of the value in the data base,
left	-- pointer to a "left" node (or NIL), and
right	-- pointer to a "right" 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 *count*ed.

If node E points to "left" node L, then all the values in the subtree rooted at L are *less than* the value in E. If node E points to "right" node 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, as other modules obtain new values to be inserted in the tree, they call *PutNewValue*.

PutNewValue calls another of *BinaryTree*'s procedures, *FindValue*, which tracks through the tree looking for a node that already has a given value. *FindValue* may find such a node, or it may be stymied -- reaching a higher-valued node with a NIL left node pointer or a lower-valued node with a NIL right node pointer. 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 at which *FindValue* was stymied.

This strategy is chosen for simplicity, but it is a poor way to construct a binary tree. For instance, if the values were entered in strictly decreasing order, the tree would be a linear list of left nodes. To find the lowest-valued node, *every* node would have to be examined.

It is recommended that the reader skip to the explanation following the example rather than read it first.

Example 2. A Package of Procedures

```
DIRECTORY
    ExampleDefs: FROM "exampledefs" USING [AllocateBlk],
    OrderedTableDefs: FROM "orderedtabledefs" USING [UserProc];

BinaryTree: PROGRAM IMPORTS ExampleDefs EXPORTS OrderedTableDefs =
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 compile-time constant

-- a global variable
root: Node;

-- public actual procedures:
SeekValue: PUBLIC PROCEDURE[val: INTEGER] RETURNS[count: CARDINAL] =
    BEGIN
    node: Node;
    found: BOOLEAN;
    [found, node] ← FindValue[val];                -- see if it is in the tree
    RETURN[IF found THEN node.count ELSE 0];
    END;

PutNewValue: PUBLIC PROCEDURE[val: INTEGER] =
    BEGIN
    node, nextNode: Node;
    alreadyInTree: BOOLEAN;
    -- Use FindValue to find where to put val:
    [alreadyInTree, node] ← FindValue[val];
    IF alreadyInTree THEN node.count ← node.count+1
    ELSE BEGIN -- name "external" procedure AllocateBlk by qualification
        nextNode ← ExampleDefs.AllocateBlk[nodeSize];
        nextNode↑ ← BinaryNode[val, 1, NIL, NIL];  -- initialize the new node
        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: OrderedTableDefs.UserProc] =
    BEGIN
    keepGoing: BOOLEAN ← TRUE;
    Walk: PROCEDURE [node: Node] =                 -- a local procedure (sec. 5.6)
        BEGIN  -- walk through the tree in order by increasing value using recursion
        IF node = NIL OR NOT keepGoing THEN RETURN; -- don't follow empty (sub)trees
        Walk[node.left];                -- enumerate the lesser-valued nodes first
        keepGoing ← userProc[node.value, node.count]; -- enumerate myself
        Walk[node.right];               -- then enumerate the greater-valued nodes
        END; -- of Walk
    Walk[root];                         -- just start enumerating at the root
    END;
```

```
52: -- a procedure that is private to this module
53: FindValue: PROCEDURE[val: INTEGER] RETURNS[inTree: BOOLEAN, node: Node] =
54:     BEGIN
55:     nextNode: Node ← root;                   -- always start at the root
56:     IF root=NIL THEN RETURN [FALSE, NIL];
57:     UNTIL nextNode=NIL
58:         DO
59:         node ← nextNode;
60:         nextNode ← SELECT val FROM
61:             < node.value  => node.left,
62:             > node.value  => node.right,
63:             ENDCASE   => NIL;
64:         ENDLOOP;
65:     RETURN[val=node.value, node];
66:     END;
67:
68: -- mainline statements
69: root ← NIL;                                  -- make tree initially empty
70: END.
```

5.5.1. The example

Each line of the source code for Example 2 is numbered for convenience; other than that, the code could be compiled as it stands.

The DIRECTORY section at the beginning maps identifiers acceptable to Mesa into file names (which are, in general, not simple identifiers). The identifier *ExampleDefs* must be the name of the (DEFINITIONS) module that is stored on the file named "exampledefs". This allows the definitions in that module to be shared among a number of modules and further allows those definitions to be compiled separately from modules like *BinaryTree* here. *ExampleDefs* and *OrderedTableDefs* are said to be *included* by the module *BinaryTree* here. The USING clause provides compiler checked documentation of exactly which identifiers are used in this program whose definitions come from the given included module (sec. 7.2.1).

Following that DIRECTORY section is the header of the *BinaryTree* PROGRAM module:

```
5: BinaryTree: PROGRAM IMPORTS ExampleDefs EXPORTS OrderedTableDefs =
```

The IMPORTS list (sec. 7.4.1) on line 5 allows *BinaryTree* to have access to procedures defined in the DEFINITIONS module *ExampleDefs* (*AllocateBlk* is the only procedure of that interface used here - details below). The EXPORTS list (sec. 7.4.3) contains only the single name *OrderedTableDefs*: this module provides actual, PUBLIC procedures corresponding to those defined by *OrderedTableDefs* as an interface (see below). Here is the DEFINITIONS module *OrderedTableDefs*:

```
OrderedTableDefs: DEFINITIONS =
    BEGIN
    -- Types and compile-time constants
    UserProc: TYPE =
            PROCEDURE [val: INTEGER, count: CARDINAL] RETURNS[continue: BOOLEAN];
    -- The interface
    SeekValue: PROCEDURE[val: INTEGER] RETURNS[count: CARDINAL];
    PutNewValue: PROCEDURE[val: INTEGER];
    EnumerateValues: PROCEDURE[userProc: UserProc];
    END.
```

The *ExampleDefs* DEFINITIONS module has the following (skeletal) form:

```
ExampleDefs: DEFINITIONS =
   BEGIN
   . . .
   AllocateBlk: PROCEDURE[size: CARDINAL] RETURNS[theStorage: POINTER];
   . . .
   END.
```

The interface defined by this set of definitions consists of the three procedure definitions for *SeekValue, PutNewValue*, and *EnumerateValues*. *UserProc* is the only type definition (it is in the DEFINITIONS module because any program that uses *BinaryTree* needs it to define a procedure that could be passed as a parameter to *EnumerateValues*).

The body of a PROGRAM module resembles a procedure body: BEGIN, followed by declarations, then some statements, and finally END (which, it should be noted, is followed by a period, unlike other ENDs in Mesa). The declarations and statements are both optional, but it would be unusual to omit the declarations. The statements are called the module's *mainline statements* and are executed by STARTing it (sec. 7.8).

BinaryTree declares four actual procedures: *SeekValue* (lines 17-23), *PutNewValue* (lines 25-38), *EnumerateValues* (lines 40-51), and *FindValue* (lines 53-66). It also declares two types (*Node* and *BinaryNode*), a compile-time constant (*nodeSize*), and a single global variable (*root*).

After the declarations, there is a single program statement (line 69), which simply initializes *root* to be NIL to make the tree initially empty.

An inspection of the actual procedure, *SeekValue*, will help to explain some things (*SeekValue* exists to allow other modules to determine whether a given value is in the tree):

```
17: SeekValue: PUBLIC PROCEDURE[val: INTEGER] RETURNS[count: CARDINAL] =
18:    BEGIN
19:    node: Node;
20:    found: BOOLEAN;
21:    [found, node] ← FindValue[val];          -- see if it is in the tree
22:    RETURN[IF found THEN node.count ELSE 0];
23:    END;
```

PUBLIC indicates that this procedure is publicly available and can therefore be part of the interface *OrderedTableDefs*.

SeekValue and *PutNewValue* use variables named *val* (a parameter in both cases) and *node* (a local variable declared by both procedures). Remember that these are distinct variables, local to each procedure, and only exist in activations of the procedure in which they appear.

The procedure *FindValue* is PRIVATE to *BinaryTree* (because it is not declared PUBLIC), so it is only called from within the module (from lines 21 and 30).

The procedure *EnumerateValues* has two major distinguishing features: it takes a procedure value as a parameter, and it contains a local procedure (*Walk*). For each node in the tree, *EnumerateValues* will call the procedure value *userProc* that it received as an argument, passing it the value in that node and its replication count. If *userProc* returns TRUE, the enumeration of the values continues; if it returns FALSE, *EnumerateValues* will complete and return to its caller. The values are produced in order from least to greatest.

The local procedure *Walk* is recursive and walks around the tree by first walking over the left subtree, then the root, and finally the right tree. This delivers the values in increasing order (convince yourself that it does). Section 5.6 treats local procedures in more detail.

5.5.2. Invoking the example's procedures

FindValue must be called as a function because it returns a result:

30: [*alreadyInTree*, *node*] ← *FindValue*[*val*];

Procedures that do not return values are called by writing the call as a statement in its own right: there is no special keyword required. For example:

PutNewValue[*ww*];

A function cannot be called using a statement, and a procedure (which does not return a result) cannot be called in an **Expression.** You cannot call *PutNewValue* in an **Expression** such as

huh ← *PutNewValue*[*ww*]; -- WRONG! (no result is available)

and you cannot call *SeekValue* using a statement form such as

SeekValue[*val*: 5]; -- WRONG! (something must be done with the result)

The actual procedures in *BinaryTree* become available for use when the module is instantiated. Each procedure is inactive until it is invoked. An inactive procedure is represented only by a *code body* -- part of the object code for the entire module. The *code body* never changes.

When *PutNewValue* is invoked, control transfers to the procedure's code, which allocates a frame, stores control data, assigns the argument to parameter *val*, and then continues with its first statement. *PutNewValue* calls *FindValue* (line 30). Before invoking it, Mesa saves the current value of the "program counter" in *PutNewValue*'s frame (in preparation for resuming execution after *FindValue* returns).

Invocation of *FindValue* ensues as described for *PutNewValue*, and a pointer to *PutNewValue*'s frame is saved in *FindValue* frame for the subsequent return. Execution runs its course, and *FindValue* returns (line 56 or 65). The output record (for *FindValue*) is generated in order to pass it back to *PutNewValue*. Then *FindValue*'s frame is deallocated. The activation of *PutNewValue* is restored to running status, and *PutNewValue* resumes execution. Eventually, it too returns (line 37) and its activation frame is deallocated.

5.6. Local procedures

Actual procedures may be declared within actual procedures. These are *local procedures* nested in (and local to) an *outer* procedure. An example containing local procedures follows and is discussed below. The example comes in two parts: first a DEFINITIONS module to define some TYPEs (including procedure types which are *not* part of the interface because they are types) that would be needed by a program which wanted to use this game-playing program. The procedure *NewGame* is also defined in the DEFINITIONS module as the only procedure in its interface. The second part is the program module containing a single PUBLIC procedure *NewGame*, which in turn contains four local procedures, *FreshDeck, Deal, WhereIam,* and *Peek*. These two parts are separately compiled and must exist in separate files.

```
DealerDefs: DEFINITIONS =
BEGIN      -- type definitions
Card: TYPE = RECORD[value: Value, suit: Suit];
Value: TYPE =
      {two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, ace, joker};
Suit:  TYPE = {clubs, diamonds, hearts, spades, none};
Shuffle:   TYPE = PROCEDURE;
DealCard: TYPE = PROCEDURE RETURNS[Card];
TopIs:     TYPE = PROCEDURE RETURNS[top: INTEGER];
YouScan:   TYPE = PROCEDURE
   [DoYourThing: PROCEDURE[with: Card] RETURNS[again: BOOLEAN], startAt: [1..52]];
-- the interface
NewGame: PROCEDURE[PlayGames: PROCEDURE [Shuffle, DealCard, TopIs, YouScan]];
END.
```

The PROGRAM module:

```
DIRECTORY DealerDefs: FROM "DealerDefsFile" USING [
      Card, DealCard, Shuffle, TopIs, YouScan];
TheDealer: PROGRAM EXPORTS DealerDefs =
BEGIN
NewGame: PUBLIC PROCEDURE
                    [PlayGames: PROCEDURE [Shuffle, DealCard, TopIs, YouScan]] =
   BEGIN
   deck: ARRAY [1..52] OF Card;
   n: INTEGER;
   -- local procedures for NewGame:
   FreshDeck: Shuffle =
      BEGIN n ← 0; END;
   Deal: DealCard =
      BEGIN RETURN[IF (n←n+1)<53 THEN deck[n] ELSE Card[joker, none]]; END;
   WhereIam: TopIs = BEGIN RETURN[IF n<52 THEN n+1 ELSE 53]; END;
   Peek: YouScan =
      BEGIN
      i: [1..52];
      onward: BOOLEAN ← TRUE;
      FOR i IN [startAt..52] WHILE onward
         DO onward ← DoYourThing[with: deck[i]]; ENDLOOP;
      END;
   -- statements for NewGame:
   FreshDeck[];
   PlayGames[FreshDeck, Deal, WhereIam, Peek];
   END;
END.
```

TheDealer is a module designed to support an *abstract data structure*, a card deck, and a set of operations on it, the *only* operations which can alter the deck. It does not know or care about the card games played by those using the data structure. *TheDealer* protects its decks of cards from outside intervention through the use of local procedures.

NewGame is the outer procedure for local procedures *FreshDeck*, *Deal*, *WhereIam*, and *Peek*. When invoked it prepares a fresh *deck* and calls *PlayGames* (a procedure that the calling program provides as an argument), passing arguments that are also procedure values (for the local procedures). In effect then, *NewGame* provides two facilities to *PlayGames*:

(1) The ability to invoke certain local procedures which perform "card-handling" services.

(2) A specific "deck of cards" for those local procedures to use (i.e., the current values of local variables *deck* and *n*).

For instance, *PlayGames* may invoke *Deal* to receive the top card from the deck. If it invokes *Deal* again, the next card is obtained, and so forth. *PlayGames* is the only procedure that "deals" from the given deck; no one else is *able* to deal from this deck.

Of course, *PlayGames* may choose to pass that local procedure value to some other procedure, *PlayDrawPoker* for instance. This would allow it to invoke *Deal*, and it in turn could pass that value to still other procedures, say *DrawNewCards*. The set of procedures (*NewGame, PlayGames, PlayDrawPoker, DrawNewCards*, etc.) that have access to the original local procedure values form a "complex", all working with the same deck.

Eventually, this complex concludes its work, and *PlayGames* returns to *NewGame*. Since *NewGame* has nothing left to do, its activation terminates and the frame disappears. This means that all of *NewGame*'s local variables cease to exist -- *including the local procedures*! (Anyone still holding procedure values for the original local procedures is in a precarious position. If those values are called, the consequences are undefined: "All bets are off.")

Let's go back to the complex and suppose *NewGame* is invoked again, from somewhere within that complex (i.e., *NewGame* is called recursively). Then an entirely new game begins -- with a second deck and different local procedure values. (When these values are called, they invoke local procedures that can operate only on the second deck.) As this game progresses, a second complex is formed, does its work, and eventually concludes as described before.

Note that the second complex cannot directly access the original deck. (For example, no procedure in the second complex could assign a value to *n* in the first complex.) The second complex may only affect the original deck by calling the original local procedure values. It can't even do that unless the first complex made those values accessible to the second one. Notice that the first complex exercises full control here; it may choose to allow access to all, none, or just some of those procedure values.

Peek is an interesting procedure: It permits another (arbitrary) procedure to peek at the cards, one at a time, without "disturbing" the deck. *Peek* is an example of a procedure sometimes called an *enumerator* of a data structure. Look at *Peek* in the example while reading the following explanation:

Suppose a game is in progress and *Peek* is invoked. It repeatedly calls *DoYourThing*, passing it the value of each card in the deck (starting at position *startAt*). In effect, *Peek* "walks through" the deck calling some unknown procedure to process the card values. That procedure may stop *Peek* by returning the result FALSE; otherwise *Peek* "walks" to the end of the deck.

Peek neither knows nor cares what actual procedure is invoked when *DoYourThing* is called. (That's the point of this exercise.) Separate invocations of *Peek* may use different procedure values for parameter *DoYourThing*. A few of the possibilities would be:

A procedure that looks at cards yet to be dealt in order to optimize some game-playing strategy [a high-sounding way of saying it cheats].

A procedure that peeks at the deck before cards are dealt [and asks for a fresh deck if it thinks it may lose].

A procedure that looks at cards already dealt [trying to figure out how you managed to win the last game].

A procedure that displays the hands dealt for a game it found interesting [it finally won and wants to gloat].

TheDealer module shows two levels of nesting: *NewGame* is nested within the module, and *FreshDeck*, *Deal*, etc. are nested within *NewGame*. Further levels are possible; for example, *FreshDeck* might declare its own actual procedures. It would then be the outer procedure for these new local procedures.

In a sense, all procedures are "local" procedures. They are either local to some outer procedure or local to some module (recall that static variables are local to the module declaring them). Therefore, Mesa treats all local procedures the same regardless of their level of nesting. (The level is important only to the extent that it influences name scopes, a topic covered in the next section.) The important issues in the general case are outlined below:

1. A *local procedure* is a local variable and represents an actual procedure declared by some outer procedure. Like all local variables, it does not exist until its *host* (i.e., the outer procedure) is invoked. In other words, an activation of its host is required. (Note that a local procedure cannot be invoked until that "outer" activation does exist.)
2. When the host is invoked, a procedure value (for the local procedure in this "outer" activation) becomes available. If the host is invoked again, a *different* procedure value (for the local procedure in *that* "outer" activation) becomes available.
3. A *procedure value* has two parts. One part contains data for invoking the procedure (indicating the location of the code body). The second part specifies which activation contains the local procedure (this data provides a link to the host's frame for accessing non-local variables).
4. Whenever a given procedure value is called, Mesa invokes that local procedure and "ties" the new frame to its host's frame (using the second part of the procedure value).
5. Subject to the rules of scope, a local procedure may access variables in these specific frames:

   Its own frame -- thus, a local procedure may access its own locals.

   The frame to which its own frame is "tied" -- it may then access the host's locals.

   The frame to which that frame is "tied" -- it may also access locals of the next outer procedure, if there is one.

   This sequence ends with the (module) frame to which the outermost procedure's frame is "tied" -- it may access the module's static variables. A frame for the module is generated when that module is instantiated by a NEW operation (sec. 7.6.1).

5.6.1. Scopes defined by procedures

Each actual procedure defines a new scope for names declared in that procedure. Such names represent variables that are local to the actual procedure. The scope for a local variable is such that:

(1) the local variable is unknown *outside* of that actual procedure, and

(2) a non-local variable is unknown *inside* the actual procedure *if its name matches some local variable's name.*

In the following example, scopes for the procedures are indicated by comments:

```
SomeModule: PROGRAM =
BEGIN
var: INTEGER;
. . .                        -- the var of INTEGER type is used here
OuterProc: PROCEDURE =
    BEGIN
    var: BOOLEAN;
    . . .                        -- the var of BOOLEAN type is used here
    LocalProc: PROCEDURE =
        BEGIN
        var: CHARACTER;
        . . .                        -- the var of CHARACTER type is used here
        END;
    . . .                        -- the var of BOOLEAN type is used here
    END;
. . .                        -- the var of INTEGER type is used here
END.
```

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 fròm zero up to its maximum length.
text	an indexable sequence of characters.

STRING is a predefined type in Mesa. Each program contains the following relevent pre-declarations:

```
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 STRING type 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 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: *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] ← anotherChar; s.length ← s.length+1;
```

Actually, characters are seldom appended in this manner. *The recommended practice is to use string-handling procedures provided by the Mesa system.* These are documented in the *Mesa System Documentation* and in appendix 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:

When an item has type STRING, *think "string-pointer".*

6.1.1. 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 **Expression** of type STRING. Its value is a constant *pointer* to a constant **StringBody** in which:

length	=	number of characters given, and
maxlength	=	*length*

The fundamental operations are defined for string **Expressions**. *They deal with them as pointer values*; e.g., ← assigns one string pointer to another string pointer, = compares two strings for the same pointer value, and # compares two strings for different pointer values.

If *exp* is a string **Expression**, then *exp.length* and *exp.maxlength* are **Expressions** of type CARDINAL, *exp.text* is an **Expression** of type PACKED ARRAY [0..0) OF CHARACTER, and *exp*[*i*] is an **Expression** of CHARACTER type. The two forms *exp*[*i*] and *exp.text*[*i*] are equivalent.

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):

```
Initialization      ::= ... ← [ Expression ] | = [ Expression ]
```

The **Expression** must be a compile-time constant **Expression** of type CARDINAL. At run-time, Mesa creates a string structure with *maxlength* equal to this **Expression**'s value, *length* equal to zero, and *text* uninitialized. The declared string variable is then set to point to this string structure. *If an* **IdList** *is declared with this form of initialization, all of the listed variables initially point to the same string structure. It is recommended that this "feature" be avoided.*

Some examples:

```
currentLine: STRING ← [256];
stringBuffer: STRING ← [stringMax+someExtra];
```

This would cause allocation of two string structures. The string *currentLine* would point to one whose *maxlength* is 256. The string *stringBuffer* points to the other string structure. (Note that *stringMax* and *someExtra* must be compile-time constants.) It is legal to assign new pointer values to these string variables; this is not fixed form initialization.

The following are examples of fixed form string initialization:

```
whatWasThat: STRING = "Eh?";
goofed: STRING = whatWasThat;
```

In this case, Mesa would allocate and fill in a string structure for string constant "Eh?". *whatWasThat* and *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:

```
huh: STRING = "Eh?";
```

String variables can be declared with ← initialization or without any initialization:

```
stdErrorMsg: STRING ← "It seems that you have made a mistake."
firstReply, reply: STRING ← "Yes";
oldBuffer, newBuffer: STRING;
. . .
IF quickDialog THEN stdErrorMsg ← whatWasThat;
. . .
IF reply[0]='? THEN
    IF firstReply[0]='? THEN HelpaLot
    ELSE HelpaLittle;
. . .
```

```
oldBuffer ← newBuffer ← stringBuffer1;
. . .
IF stringBuffer1#stringBuffer2 THEN newBuffer ← stringBuffer2;
```

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 = INTEGER,
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:

```
ArrayDescriptorTC ::= DESCRIPTOR FOR ArrayTC    |
                      DESCRIPTOR FOR PackingOption ARRAY OF
                         TypeSpecification
PackingOption     ::= empty  |  PACKED
```

For example,

```
events: DESCRIPTOR FOR ARRAY [1..999] OF Date;
```

If the second form, where no **IndexType** is given, the index type is an integer subrange starting at zero.

A value for *events* is an array descriptor (a record-like object containing items similar to those described previously for *schedule* except that the *base* is not fixed). The next declaration specifies an array descriptor in which the *base* and the *length* are variable:

```
history: DESCRIPTOR FOR ARRAY OF Date;
```

Indexing can be used to access components of *events* and *history* as if they were actual arrays instead of array descriptors (see sec. 3.2.1). Since no index type is specified for *history*, it has an *indefinite* index type *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 DESCRIPTOR FOR ARRAY [0..2] OF T and DESCRIPTOR FOR ARRAY [1..3] OF T are different types, even though the lengths and element types are the same. **Expressions** of equivalent descriptor types may be compared for equality (= or #).

The rules for assignment are somewhat more relaxed. If *a1* has type DESCRIPTOR FOR ARRAY OF *T*, and *a2* has type DESCRIPTOR FOR ARRAY [0..10) OF *T*, then the assignment *a1* ← *a2* is legal, but the assignment *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 *events* to *schedule* because the first is a descriptor and the second is an actual array.

There are three function-like operators relevant to array descriptors: DESCRIPTOR, BASE, and LENGTH. DESCRIPTOR returns an array descriptor result and has three distinct forms which are treated syntactically as built in functions:

```
BuiltinCall     ::= ... |
                    DESCRIPTOR [ Expression ] |
                    DESCRIPTOR [ Expression , Expression ]   |
                    DESCRIPTOR [ Expression , Expression , TypeSpecification ] |
                    BASE [ Expression ] |
                    LENGTH [ Expression ]
```

The first form takes an argument of some array type; e.g.,

```
events ← DESCRIPTOR[schedule];
```

The result is an array descriptor for *schedule*. The second form needs two arguments:

```
base: POINTER TO UNSPECIFIED       -- address of the minIndex component
length: CARDINAL                   -- 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 DESCRIPTOR construct is provided which takes three arguments. The third one is a **TypeSpecification**, the component type.

The following example provides a fresh array of 64 *Dates*:

```
Allocate: PROCEDURE[BlkSize] 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  ::= . . .  | ArrayDescriptorTC
```

All the standard operations on array desciptors (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 it 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 DESCRIPTOR[,] or 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:

```
d: DESCRIPTOR FOR ARRAY OF T;
dd: LONG DESCRIPTOR FOR ARRAY OF T;
i, n: CARDINAL;
pp: LONG POINTER TO ARRAY [0..10) OF T;

dd ← DESCRIPTOR[pp↑];          -- descriptor for the entire array
dd ← DESCRIPTOR[pp, 5];        -- descriptor for half of the array
dd ← d;                        -- automatic lengthening
pp ← BASE[dd];                 -- BASE of long is long
n ← LENGTH[dd];                -- LENGTH is always a CARDINAL
```

6.3. Base and relative pointers

Mesa 4.0 provides relative pointers, i.e., pointers that are *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:

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.

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.

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 BASE and RELATIVE pointer type constructors is as follows:

PointerTC	::=	**Ordered BaseOption** POINTER **OptionalInterval PointerTail**
BaseOption	::=	**empty** | BASE
TypeConstructor	::=	... | **RelativeTC**
RelativeTC	::=	**TypeIdentifier** RELATIVE **TypeSpecification**

In a **PointerTC**, a nonempty **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 **TypeIdentifier** must evaluate to some (possibly long) pointer type which is the type of the base, 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_r), the form

base[*offset*]

denotes a pointer (to T_r), and the value of that pointer is LOOPHOLE[*base*]+*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:

```
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 *BinaryNodes*'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

```
TZHandle: TYPE = BASE POINTER TO TreeZone;
Node: TYPE = TZHandle RELATIVE POINTER TO BinaryNode;
```

The procedure *FindValue* would be written as follows:

```
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 **TypeIdentifier** and the **TypeSpecification** evaluate to pointer types. Let a **RelativeTC** be

> **TypeIdentifier** RELATIVE **TypeSpecification,**

where

> **TypeIdentifier** is of type
>
> > [LONG] BASE POINTER [$SubRange_b$] TO T_b ,
>
> **TypeSpecification** is of type
>
> > [LONG] [ORDERED] [BASE] POINTER [$SubRange_r$] TO T_r ,

and the brackets indicate optional attributes. Relative pointer values must be relocated before they are dereferenced. The relocation yields an absolute pointer with type

> [LONG] [ORDERED] [BASE] POINTER TO T_r ,

where the result is LONG if either component is, any subrange restrictions are dropped, and the basing and ordering attributes are inherited from the **TypeSpecification.** In essence, the relocation takes a POINTER TO T_b and a RELATIVE POINTER TO T_r and produces a POINTER TO T_r, perhaps with some modification of range-related attributes also.

> The base type must be designated by an identifier (rather than a **TypeSpecification**) to avoid syntactic ambiguities. Note that the form
>
> > LONG **TypeIdentifier** 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
>
> > **TypeIdentifier** RELATIVE LONG **TypeSpecification** .

If *base* and *offset* are base and relative pointers respectively, note that *base*[*offset*] is not a variable; typical variable designators are *base*[*offset*]↑ or *base*[*offset*].*field*. (In addition, the usual rules for implicit dereferencing apply in, for example, an **OpenItem**). Relocation prior to dereferencing is mandatory; *offset*↑, *offset.field*, etc. are compiler-time errors.

Relative pointers are never widened 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

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:

baseA: BaseA;
baseB: BaseB;
OffsetA: TYPE = *BaseA* RELATIVE POINTER TO *T*;
OffsetB: TYPE = *BaseB* RELATIVE POINTER TO *T*;
offsetA: OffsetA;
offsetB: OffsetB .

If *BaseA* and *BaseB* are distinct types (see the preceding point), so are *OffsetA* and *OffsetB*. Expressions such as *baseA*[*offsetB*] and *offsetA* ← *offsetB* are then errors, but *baseA*[*offsetA*]↑ and *baseB*[*offsetB*]↑ have the same type (*T*).

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:

p: POINTER TO ARRAY *IndexType* OF ...;
q: BASE POINTER TO ARRAY *IndexType* OF ... .

The expression *p*[*e*] will cause implicit dereferencing of *p* and is equivalent to *p*↑[*e*]. On the other hand, *q*[*e*] is taken to specify relocation of a pointer, even if the type of *e* is *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., *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 **RelativeTC** constructs a relative array descriptor type whenever the **TypeIdentifier** evaluates to a pointer type and the **TypeSpecification** evaluates to an array descriptor type. Let a **RelativeTC** be

TypeIdentifier RELATIVE **TypeSpecification,**

where

TypeIdentifier is of type

[LONG] BASE POINTER [$SubRange_b$] TO T_b ,

TypeSpecification is of type

[LONG] DESCRIPTOR FOR ARRAY T_i OF T_c ,

and the brackets indicate optional attributes. Relative array descriptor values must be relocated before they are indexed. The relocation yields an absolute descriptor with type

[LONG] DESCRIPTOR FOR ARRAY T_i OF T_c , where the result is LONG if either component is.

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. Note the following:

> In the constructor of a relative array descriptor type, the **TypeSpecification** 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
>
> *base*[*offset*][*i*]
>
> 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 components that apply to particular variants of the record. The types of these components are specified for each possible variant when declaring the record type.

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
                                    |---- . . .
                                    |---- 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:

```
    DeRec[c1: val1, c2: val2,  vp: class1[x: val3, y: val4]]
```

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):

```
    WITH someClass SELECT currentTag FROM
        class1 => Stmt-1; -- someClass is a bound class1 variant here
        class2 => Stmt-2; -- someClass is a bound class2 variant here
        class3 => Stmt-3; -- someClass is a bound class3 variant here
        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 : TypeIdentifier 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 subsumes the partial definition given in section 3.4.2 and includes machine-dependent record types:

```
RecordTC          ::= MachineDependent RECORD [ VariantFieldList ]
MachineDependent  ::= empty | MACHINE DEPENDENT
VariantFieldList  ::= CommonPart identifier : VariantPart |
                      VariantPart   |
                      NamedFieldList       |
                      UnnamedFieldList  |
CommonPart        ::= empty |
                      NamedFieldList ,
VariantPart       ::= SELECT Tag FROM
                      VariantList
                      ENDCASE
Tag               ::= identifier : TagType |
                      COMPUTED TagType |
                      OVERLAID TagType
TagType           ::= TypeSpecification | *
VariantList       ::= Variant | VariantList Variant
Variant           ::= IdList => [ VariantFieldList ] , |
                      IdList => NULL ,
```

Notes: The **TypeSpecification** in **TagType** must be equivalent to some enumeration or enumerated subrange type. In the equations shown above for **VariantFieldList** and **Tag**, "**identifier** :" may be followed by an **Access** (e.g., PUBLIC or PRIVATE), discussed in sec. 7.4. The only kind of **CommonPart** which can precede a **VariantPart** is a **NamedFieldList**. If there is no **CommonPart**, the **VariantPart** itself need not be named.

The following example is more baroque than anything likely to appear in real programs. Its only virtue is that it shows many of the possible variations resulting from the above syntax definitions. In fact, it would be worthwhile parsing it yourself using the definitions. The example might be used to describe the various "accounts" in a bank; there would supposedly be a table of such entries:

```
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.

"*" 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,

```
NoCommon: TYPE = RECORD
    [ -- no common part
    variantPart: SELECT easyTag: * FROM
        i       => [compl: INTEGER],
        j, k    => [x, compl: 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.

Some fine points:

> 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:

```
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 **TypeIdentifier** can now be completed (it is only partially defined in Section 2.6.1):

```
TypeIdentifier   ::= ... | Adjective TypeIdentifier
Adjective        ::= identifier
```

where **Adjective** is an adjective of the variant part in the type specified by **TypeIdentifier.** Note that the recursive use of **TypeIdentifier** 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):

```
rDisk ← rLong;                       -- aggregate access
rDisk.Get ← rShort.Get;              -- selector access
r.body ← rDisplay.body;              -- selector access
[rDisk.Get, , rDisk.body] ← rLong;   -- extractor access
```

The actual tag, *type*, in the *body* variant part may also be accessed by qualification:

```
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):

```
myDisplay: display Stream ← [myGet, myPut, display[d1,d,h,8] ];
yourDisplay: display Stream ← myDisplay;
currentStream: Stream ← myDisplay;
s: Stream ← [SysGet,SysPut, disk[fp, 0, SysSetPos, long[al] ] ];
```

The *keyboard* variant of *Stream* is a NULL variant; so there are no components for that variant in a *keyboard* constructor:

```
rKeyb ← Stream[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* :

```
r.body ← keyboard[];
r.body ← rKeyb.body;        -- always a keyboard variant
```

If one is assigning a completely bound variant value, *bv*, say (which could be a constructor, of course) in an **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:

```
rDisplay.last ← rDisplay.first;
rDisk.position ← rShort.position;
```

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:

```
SelectVariant      ::= WITH OpenItem SELECT TagItem FROM
                       ChoiceSeries
                       ENDCASE FinalStmtChoice
                       | ...
ChoiceSeries       ::= AdjectiveList => Statement ; |
                       ChoiceSeries AdjectiveList => Statement ;
TagItem            ::= empty |           -- the actual tag is used
                       Expression        -- compute the tag value
FinalStmtChoice    ::= empty | => Statement
SelectExprVariant  ::= WITH OpenItem SELECT TagItem FROM
                       ChoiceList
                       ENDCASE => Expression
                       | ...
ChoiceList         ::= AdjectiveList => Expression , |
                       AdjectiveList => Expression , ChoiceList
ChoiceList         ::= AdjectiveList => Expression , |
                       ChoiceList AdjectiveList => 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 strm: r SELECT FROM
    display =>
        BEGIN
        strm.first←strm.last;
        strm.height←73;
        strm.nLines←4;
        END;
    disk => WITH strm SELECT FROM
        short  => b[0]←10;
        long   => b[0]←100;
        ENDCASE;
    ENDCASE => strm.body ← disk[GetFp["Alpha"], 0, SysSetPos, short[]];
```

In the first example, suppose *r* contains a variant record of *display Stream* type. Then the first arm is chosen by this SELECT. Within it, *strm* (but not *r*) is considered a record of *display Stream* type; so all components of the *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 *disk Stream* type. Then the actual tag has the value *disk*, and the second arm is chosen. In this example, only one of the *disk* components is accessed, its variant part. The inner SELECT uses variant record *strm*. Within the outer arm, Mesa knows that *strm* is a record of *disk Stream* type. Consequently, the tag implicitly used for this SELECT is the tag specified for *that* type (namely, *size*).

If the tag value is *short*, then the chosen arm accesses component *b* in the *short disk Stream* variant record; if it is *long*, then the chosen arm accesses component *b* in the *long disk Stream* variant record.

However, the ENDCASE for the inner SELECT could have accessed components that are common to a *disk Stream* (*file*, *position*, *SetPosition*, variant part *buffer*, and actual tag *size*; plus all the original common components: *Get*, *Put*, variant part *body*, and actual tag *type*).

Suppose, lastly, that *r* does not contain a variant record of *display Stream* or *disk Stream* type. Then the outer ENDCASE statement is chosen. This statement accesses the common component *body* (the *entire* variant part is considered a common component), and gives the record a specific variant type (*short disk Stream*) by wholesale assignment. An ENDCASE may only access common components; *it may not access components of variants in the given type.*

If the labels on an arm of a descrimination 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:

```
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:

```
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 indentifier 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:

```
WITH aDiskStream SELECT FROM
    short => b[0]←10;
    long  => b[0]←100;
    ENDCASE;
```

It would be illegal to rewrite this as follows:

```
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 descrimination. 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];
t: TrustMe;

t.c             -- legal
t.b             -- legal
t.next          -- illegal, both variants one and two contain such a field.
```

Fine points:

In the declaration of *TrustMe* above, the two *next* fields were of the same type, but occuppied different positions within the record. Even if they did occupy the same position, one could still not refer to *t.next*. The ambiguity is one of variant, not of value.

CHAPTER 7.

MODULES, PROGRAMS, AND CONFIGURATIONS

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

The notion of an interface is central to building systems in Mesa. An *interface* is the list of procedure, signal (chapter 8) and program (sec. 7.2) types defined in a single definitions module. For example, an interface for an allocator might consist of the names and types of the procedures for allocating and freeing 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 which 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 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:

```
IODefs: DEFINITIONS =
BEGIN
-- Interface definitions
   ReadChar: PROCEDURE RETURNS [CHARACTER];
   ReadLine: PROCEDURE [input: STRING];    -- reads from terminal into input

   WriteChar: PROCEDURE [ouput: CHARACTER];
   WriteLine: PROCEDURE [output: STRING];

-- Non-interface definitions
   CR: CHARACTER = 015C;   -- an ASCII Carriage-Return character
END. -- IODefs
```

The interface record for *IODefs* is imported by the following *Xerox* 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:

```
DIRECTORY
      IODefs: FROM "IODefs";

Xerox: 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                               -- infinite loop; only left by EXIT
   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
   ENDLOOP;
WriteLine["End of example."];       -- final output
WriteChar[CR];                      -- leave terminal on a new line
END.   -- Xerox
```

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 [ouput: 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: *Xerox, IOPkg,* and *Driver.*

```
MakeXeroxSystem: CONFIGURATION
   CONTROL Driver =
BEGIN  Xerox; IOPkg; Driver;  END.
```

This configuration specifies how the *Xerox, 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 *MakeXeroxSystem. Driver* is named as the CONTROL module for the BCD, so starting the loaded BCD would actually result in starting *Driver,* which follows:

```
DIRECTORY
        IOPkg: FROM "IOPkg",
        Xerox: FROM "Xerox";

Driver: PROGRAM IMPORTS Xerox, IOPkg =
BEGIN
START IOPkg;     -- so its variables (e.g., terminalState) are initialized
START Xerox;     -- to initialize its variables and run its mainline code
END.
```

This example is simple, but *MakeXeroxSystem* 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:

(a) 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:

```
CompilationUnit ::= Directory                          -- optional
                    DefinitionsFrom                    -- optional
                    ModuleName : ModuleHead = GlobalAccess
                    ModuleBody
Access          ::= empty | PUBLIC | PRIVATE
DefinitionsFrom ::= empty | DEFINITIONS FROM IdList ;
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 ShareList |
                    ProgramTC ImportsList ExportsList ShareList
InterfaceItem   ::= identifier | identifier : identifier
InterfaceList   ::= InterfaceItem | InterfaceList , InterfaceItem
LocksClause     ::= empty |                            -- sec. 10.4.1
                    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 contains declarations of constants and types, and definitions of the names and parameter types of procedures, signals, and programs. The sequence of definitions (i.e., all except types and constants) defines an *interface type*. This type is much like a record type, with a component for each item of the interface. A program may not declare variables of an interface type, but instances of it appear in program modules in which it is imported. These interface records are the connective tissue in a system of Mesa modules.

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. The interface records that it uses to call procedures in imported interfaces may be either in the frame or in the code.

We will first deal with the initial syntactic units which are common to all modules (the **Directory** and **DefinitionsFrom** clauses), then with DEFINITIONS modules as a whole. After these sections there is a complete example including

a DEFINITIONS module,
a 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 DIRECTORY *clause*

The source code for a given module may tell the compiler to *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 START them, or to access their data.

Suppose module *A* is included in module *B*. This means that when compiling *B*, the compiler must have access to *A*'s object file (i.e., *A* must have been compiled previously) in order to access its symbol table to obtain information needed by *B*. *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 DEFINITIONS module:

```
SimpleDefs: DEFINITIONS =
BEGIN
limit: INTEGER = 86;
Range: TYPE = [-limit..limit];
Pair: TYPE = RECORD[first, second: Range];
PairPtr: TYPE = POINTER TO Pair;
END.
```

Notice that all values expressed here are known at compile-time; this is a requirement for DEFINITIONS *modules.*

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 *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 **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 **ModuleName** in the module whose object file is named.

Here is an example of a DIRECTORY:

```
DIRECTORY
        SimpleDefs: FROM "simpledefs",
        StringDefs: FROM "stringdefs";
```

When a module includes some other module, it does not automatically gain the other's symbol definitions. (It might not need them; perhaps it just wants to load that module.) If it needs such symbols, the module must specify this in one of the ways discussed later.

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 **IncludeItem.** 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 three 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.

DEFINITIONS FROM: *User's* DEFINITIONS FROM clause is an OPEN clause encompassing the entire module. If *Defs* is named in it, then *p* suffices throughout *User*.

The remainder of this section gives more detail on these three 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 **TypeIdentifier** includes the case

TypeIdentifier ::= ... | 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 "simpledefs" USING [Pair],
        TableDefs: FROM "tabledefs";
TableUser: PROGRAM =
BEGIN OPEN TableDefs;        -- (Notice the OPEN-clause.)
vIndex: INTEGER ← limit;     -- this is TableDefs.limit because of the OPEN
vString: StringTable;
vPair: PairTable;
StoreString: PUBLIC PROCEDURE[s: STRING, v: Index] =
    BEGIN
    vString[v] ← s;
    vPair[vIndex←v] ← NIL;
    END;
```

```
StorePair: PUBLIC PROCEDURE[t: SimpleDefs.Pair] RETURNS[ok: BOOLEAN] =
    BEGIN
    ok ← vIndex <= limit;
    IF ok THEN vPair[vIndex] ← t;
    END;
END.
```

In the scope of the OPEN clause, the names *limit, StringTable, PairTable,* and *Index* are those in *TableDefs.*

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.

7.2.2.3. DEFINITIONS FROM

In most respects, DEFINITIONS FROM is equivalent to an OPEN clause following the **ModuleBody's** BEGIN. For instance, the previous example could be rewritten as follows:

```
DIRECTORY
        SimpleDefs: FROM "simpledefs",
        TableDefs: FROM "tabledefs";
DEFINITIONS FROM TableDefs;
TableUser: PROGRAM =
BEGIN   -- No OPEN-clause is needed now.
. . .
```

In either case, *TableUser* makes use of symbol definitions from *TableDefs* throughout its module body. DEFINITIONS FROM is only useful because it allows the use of *unqualified* predefined symbols in the module parameter list. In the following example, the header uses the unqualified name, *Index*:

```
DIRECTORY
        SimpleDefs: FROM "simpledefs",
        TableDefs: FROM "tabledefs";
DEFINITIONS FROM SimpleDefs, TableDefs;
AnotherTableUser: PROGRAM[maxSize: Index] =
BEGIN
x: Pair;     -- this is SimpleDefs.Pair
. . .
```

Both of the included modules define a symbol named *Index.* Which one applies in the above parameter list? The symbol defined by *TableDefs* is the one because of the ordering of the names listed after DEFINITIONS FROM -- the *rightmost* named module defining the symbol of interest "wins", just as for an OPEN-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. 7.1.1) 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 which 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:

> DEFINITIONS FROM
> OPEN clauses
> **Block**s 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

DEFINITIONS FROM and 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:

```
DIRECTORY
        SimpleDefs: FROM "simpledefs",
        TableDefs: FROM "tabledefs";
DEFINITIONS FROM SimpleDefs, TableDefs;
TableUser: PROGRAM = . . .
StorePair: PUBLIC PROCEDURE[t: Pair] RETURNS[ok: BOOLEAN] =
. . .
```

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 DEFINITIONS FROM was *TableDefs*; not finding *Pair* there either, it went on to the next (and outermost) scope given by the DEFINITIONS FROM, 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.

(4) Name an included module in a DEFINITIONS FROM clause only if it is needed for a number of references in the formal parameter and returns list of the program's module header.

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.

```
             Defs1
        _______|_______
        |             |
        |             |
      Defs2         Defs3
        |             |
        |_____________|
               |
             User1
```

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, i.e., compile-time constants, types, and procedure and signal definitions. 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 only want to call on 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 and signals), 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. 7.1.2). External interface definitions, however, are different. Normally, a declaration such as

```
    SampleProc: PROCEDURE [i: INTEGER];                    -- (A)
```

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 the next section.

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 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 *Defs* specify interface types:

```
DIRECTORY Defs1: FROM "defs1", Defs2: FROM "defs2";
Prog: PROGRAM IMPORTS IRec: Defs1, I2Rec: Defs2 =
    BEGIN . . . END.
```

Within the body of *Prog*, references like *Defs1.x* are only valid if *x* is a non-interface element of *Defs1*. However, *IRec.x* can refer to any element *x* of *Defs1*, whether interface or non-interface. This distinction is necessary because a call on a procedure, *proc*, defined in *Defs1*, 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

```
Prog: PROGRAM IMPORTS IRec: Defs1, Defs2 = . . .
```

is the same as writing

```
Prog: PROGRAM IMPORTS IRec: Defs1, Defs2: Defs2 = . . .
```

Then, within the body of *Prog*, *Defs2* refers to an interface record. In fact, it is impossible thereafter to refer to the interface type *Defs2*, although one can still refer to the interface type *Defs1* 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:

```
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. By analogy with interface types we call *X* a *program prototype*. One can use *X* to declare program *variables* of type POINTER TO FRAME[*X*], and one can create instances of it using "NEW *X*", as discussed in section 7.8.1 (It is not possible to declare variables of type FRAME[*X*] because frames cannot be embedded in other structures.)

A program module could also import *X* by naming it in its IMPORTS list. For example,

```
DIRECTORY XProg1: FROM "XProg1", XProg2: FROM "XProg2";
Prog: PROGRAM IMPORTS frame1: XProg1, XProg2 =
    BEGIN . . . END.
```

This has an effect similar to declaring

```
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 actual 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 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 actual procedures as its components. The price paid for such close coupling with a program is that the program accessing its frame must be recompiled whenever it is.

7.4.3. Exporting interfaces and program modules

A module can *export* an interface if it provides actual PUBLIC procedures, signals, or errors 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 actual elements for all the elements 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.

To gain access to the PRIVATE elements of an interface, a module must use a SHARES clause (sec. 7.5.4), even if it also EXPORTS that interface.

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*.

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

(i) Declared names

The form of **Declaration** specifying an **Access** for its declared names is as follows:

```
Declaration      ::= IdList : Access TypeSpecification Initialization ;
```

Examples:

```
    q1, q2: PUBLIC INTEGER ← 0;
    Mine: PRIVATE TYPE = {yes, no, maybe};
```

Mine can only be used in (i.e., seen from) privileged modules. To non-priviliged modules it is not visible at all.

(ii) Names specified in field lists

The forms for specifying **Access** in a **NamedFieldList** (sec. 3.4.1) are as follows:

```
        NamedFieldList ::= IdList : Access FieldDescription |
                           NamedFieldList , IdList : Access FieldDescription
        FieldDescription ::= TypeSpecification |
                           TypeSpecification ← Expression
```

Example:

```
    blk: PUBLIC RECORD
        [
        a: INTEGER,
        b: PRIVATE INTEGER ← 1234,
        c, d: BOOLEAN,
        e: PRIVATE BOOLEAN
        ];
```

A non-privileged module could only access components *a, c,* and *d* in this case, and then only using qualified references such as *blk.a.* Within a non-privileged module, extractors and constructors cannot be employed for a record type with any PRIVATE components.

(iii) 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:

```
        VariantFieldList ::= CommonPart identifier : Access VariantPart |
                             VariantPart      |
                             NamedFieldList   |
                             UnnamedFieldList
        CommonPart       ::= empty |
                             NamedFieldList ,
        VariantPart      ::= SELECT Tag FROM
                             VariantList           -- same as in sec. 6.3.1
                             ENDCASE
        Tag              ::= identifier : Access TagType |
                             COMPUTED TagType
        TagType          ::= TypeSpecification | *
```

Example:

```
VarRec: PUBLIC TYPE = RECORD
    [
    link: POINTER TO VarRec,              -- public common component
    vp1: SELECT tg1: PRIVATE Etype FROM  -- public variant, private tag
        adj1 =>
            [
            youGet: ThisItem,
            iGet: PRIVATE SELECT tg2: * 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 *vp1* but neither tag *tg1* nor variant part *iGet*. This only prevents it from referring to *tg1* by qualification; it may still use a discriminating SELECT (which implicitly accesses *tg1*) 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 actual 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 **TypeIdentifier** 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: PROGRAM = PUBLIC            -- specified GlobalAccess
BEGIN
. . .
END.

M3: PROGRAM =                   -- PRIVATE (by default)
BEGIN
. . .
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:

```
DIRECTORY
        SpecialDefs: FROM "specialdefs",
        StandardDefs: FROM "standarddefs",
        PrivateDefs: FROM "private";
Friendly: PROGRAM SHARES PrivateDefs, SpecialDefs =
BEGIN
. . .
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:

```
LexiconDefs: DEFINITIONS =
BEGIN
FindString: PROCEDURE [STRING] RETURNS [BOOLEAN];
AddString: PROCEDURE [STRING];
PrintLexicon: PROCEDURE;
END.
```

7.6.1. Lexicon: *a module implementing* LexiconDefs

The following module (*Lexicon*) implements the *LexiconDefs* interface. That is,

(a) *Lexicon* declares actual 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: *FspDefs*, *IODefs*, and *StringDefs*. We will not reproduce those definitions modules here; instead, the USING clauses of the DIRECTORY note which procedures are defined in *FspDefs* (*Fsp* stands for *F*ree *S*torage *P*ackage and is an allocation package), *IODefs*, and *StringDefs*. Section 7.1 contains a more complete definition of *IODefs*.

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.

```
    DIRECTORY
        FspDefs: FROM "fspdefs" USING [AllocateHeapNode, AllocateHeapString],
        IODefs: FROM "iodefs" USING [ReadChar, WriteChar, ReadLine, WriteLine],
        LexiconDefs: FROM "lexicondefs",
        StringDefs: FROM "stringdefs" USING [AppendString];

Lexicon: PROGRAM IMPORTS FspDefs, 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] =   -- a local procedure
            BEGIN OPEN FspDefs;
            n ← AllocateHeapNode[SIZE[Node]];
            n↑ ← Node[string: AllocateHeapString[s.length], llink: NIL, rlink: NIL];
            StringDefs.AppendString[n.string, s];
            RETURN;
            END;

        IF n = NIL THEN root ← NewNode[]   -- then just return
        ELSE
            SELECT LexicalCompare[s, n.string] FROM
                lessThan      => IF n.llink # NIL THEN InsertString[n.llink, s]
                                 ELSE n.llink ← NewNode[];
                equalTo       => NULL;        -- already there; just return
                greaterThan   => IF n.rlink # NIL  THEN InsertString[n.rlink, s]
                                 ELSE n.rlink ← NewNode[];
                ENDCASE;
        END;

    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;
        -- Characters match; so result depends on how the strings compare in length:
        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[IF 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;
        PrintNode[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.*

```
DIRECTORY
    IODefs: FROM "iodefs" USING [CR, ReadChar, ReadLine, WriteChar, WriteLine],
    LexiconDefs: FROM "lexicondefs" USING [AddString, FindString, PrintLexicon];

LexiconClient: PROGRAM IMPORTS IODefs, LexiconDefs =
BEGIN OPEN IODefs, LexiconDefs;
s: STRING ← [80];
ch: CHARACTER;
    DO -- loop until stopped by user typing q or Q (last case below).
    WriteChar[CR]; WriteLine["Lexicon Command: "];
    ch ← ReadChar[]; WriteChar[ch];-- Echo the character (ReadChar doesn't).
    SELECT ch FROM
        'f, 'F =>
            BEGIN
            WriteLine["ind:  "];              -- terminal will read: "find:  "
            ReadLine[s];       -- s will contain the string read from the terminal
            IF FindString[s] THEN WriteLine[" -- found"]
            ELSE WriteLine[" -- not found"];
            END;
        'a, 'A =>
            BEGIN
            WriteLine["dd:  "];               -- terminal will read: "add:  "
            ReadLine[s];
            AddString[s];
            END;
        'p, 'P =>
            BEGIN
            WriteLine["rint lexicon"];        -- terminal will read: "print lexicon"
            WriteChar[CR]; PrintLexicon[];
            END;
        'q, 'Q =>
            BEGIN
            WriteLine["uit"]; WriteChar[CR];   -- terminal will read: "quit"
            STOP;
            END;
        ENDCASE =>WriteLine[" is not a command character"];
    ENDLOOP;
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. Space for the procedure descriptors for imported procedures used by the module may be 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 procedure descriptors in all the static frames of the configuration instance. Some of these procedure descriptors will "point to" procedures in the same configuration. Others will "point to" procedures 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
Fsp;            --                                        EXPORTS FspDefs
IOPkg;          --                                        EXPORTS IODefs
Strings;        --                                        EXPORTS StringDefs
Lexicon;        -- IMPORTS FspDefs, IODefs, StringDefs    EXPORTS LexiconDefs
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  ::= Directory              -- optional; same as for Mesa
                       CPacking
                       Configuration .        -- note the final period
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 separate documentation of the Binder. They will not be further discussed here.

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 *FspDefs, 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 FspDefs, 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, *FspDefs* 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: FspDefs, 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: FspDefs, 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:*

```
c2x.1:  Config2X: CONFIGURATION
c2x.2:      IMPORTS FspDefs: FspDefs, IODefs: IODefs, StringDefs: StringDefs
c2x.3:      CONTROL LexiconClient =
c2x.4:  BEGIN
c2x.5:  LexiconDefs: LexiconDefs ← Lexicon: Lexicon[FspDefs, 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:

```
c4.1:  Config4: CONFIGURATION
```

```
c4.2:      IMPORTS alloc: FspDefs
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] ← StringsAndIO[];
```

In this case the types for *io* and *str* would be inferred from the types of the interface records exported by *StringsAndIO*. However, if the programmer had written instead,

```
[io, str] ← StringsAndIO[];
```

with the positions of *io* and *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 *Config4* could simply have been written as

```
c4a.1:  Config4A: CONFIGURATION
c4a.2:     IMPORTS FspDefs
c4a.3:     CONTROL LexiconClient =
c4a.4:  BEGIN
c4a.5:  StringsAndIO;
c4a.6:  Lexicon;
c4a.7:  LexiconClient;
c4a.8:  END.
```

This would assign the exported interfaces to the default-named records *StringDefs* and *IODefs* and would use them in the defaulted import parameter lists for *Lexicon* and *LexiconClient*. Line c4a.5 could also show what *StringsAndIO* exports using the default names for its exported records. This would give rise to the statement:

```
[StringDefs, IODefs] ← StringsAndIO[];
```

Cases like this require that the user be aware of the distinction between interface records and interface types: *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 *Lexicon* is divided into two modules *LexiconFA* and *LexiconP*, with *LexiconFA* providing the procedures *FindString* and *AddString*, and *LexiconP* providing *PrintLexicon*. Each exports *LexiconDefs*, but neither fully implements that interface. Still, *LexiconClient* will see a single interface in the following:

```
c5.1:   Config5: CONFIGURATION
c5.2:      IMPORTS FspDefs, IODefs, StringDefs
```

```
c5.3:     CONTROL LexiconClient =
c5.4:   BEGIN
c5.5:   lexRec: LexiconDefs ← LexiconFA[];      -- use default imports
c5.6:   lexRec ← LexiconP[];                    -- merge interface contributions
c5.7:   LexiconClient[IODefs, lexRec];
c5.8:   END.
```

The two separate assignments to *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 *PrintLexicon*, for example, an error would be generated during processing of *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

```
lexRecFA ← LexiconFA[];                     -- one part
lexRecP ← LexiconP[];                       -- the other part
lexRec ← lexRecFA PLUS lexRecP;             -- the merge
LexiconClient[IODefs, lexRec];              -- same as line c5.7
```

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:

```
lexRec ← Lexicon[];                         -- defines a complete interface
lexRecP ← LexiconP[];                       -- defines one procedure
lexRecNew ← lexRecP THEN lexRec;            -- this order is important
LexiconClient[IODefs, lexRecNew];
```

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:

```
NonexportingPM: CONFIGURATION = BEGIN ProgMod END.
```

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: NEW and START

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 acomplished using a single Mesa operator, NEW. 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 these methods are described below).

7.8.1. The NEW *operation for atomic modules*

The syntax for the NEW operation is

```
        Expression      ::=  . . . | NEW Variable
```

The **Variable** may be the name of a program prototype, an imported frame pointer, a pointer to the frame for a program module, or a program variable defined in an interface. Each of these cases is described below.

A program prototype is named by including it in the DIRECTORY section of a module. Thus, if the DIRECTORY section contains

```
    Prog: FROM "ProgFile"
```

then *Prog* is the name of a prototype. The expression "NEW *Prog*" loads an instance of *Prog* and yields a pointer to its frame. So, if a program that includes *Prog* contains a declaration and a statement such as

```
    progInst: POINTER TO FRAME[Prog];
    . . .
    progInst ← NEW Prog;
```

then *progInst* will point to its frame.

NEW can also be used to make a copy of a program instance. The 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.

A copy is made (as opposed to a new instance) whenever the **Variable** following NEW does *not* name a prototype. Thus, "NEW *progInst*" would copy *progInst*, while "NEW *Prog*" would make an entirely new instance from the prototype *Prog*. Similarly, if a module imports a program *Pimp* (sec. 7.4.2), the operation "NEW *Pimp*" copies *Pimp* rather than making a brand new instance.

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. They will all be copies, and not new instances. For example, assume that the following declaration appears in the definitions module, *Defs*:

```
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, and not like a program prototype.

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. For example, 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

```
StartStmt        ::= 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:

```
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 (howeyer, it cannot use both).

A program which executes a STOP can be RESTARTed later. RESTART is distinct from START primarily because it cannot pass parameters as START can. If a program does not return results, it can only stop either by an explicit use of STOP or by running off the end of the program. At the end, the compiler places an implicit RETURN (sec. 5.3.1).

A program which declares (in its **ModuleHeader**) that it returns results uses RETURN statements just as does a procedure (and it cannot use STOP). A RETURN from a program does *not* deallocate its frame as a RETURN from a procedure does. The syntax for RESTART and STOP is

```
RestartStmt    ::= RESTART Variable | . . .
StopStmt       ::= STOP | . . .
```

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 which 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. NEW *and* START *for configurations*

NEW can also make instances of configurations which are more than simple, atomic modules. To do this, one must include the configuration in the DIRECTORY section of a program (let's call it *LoadProg*). Then, one makes a new instance (never a copy, for a configuration) of the BCD (call it *Config*) by the operation "NEW *Config*". The result returned by NEW in this case is either a valid frame pointer to the configuration's CONTROL module, if it has one, or a *null frame pointer*. A null frame pointer is *not* the same as the value NIL. Rather, it is a value that can cause a trap if a program uses it in a START or RESTART operation.

The pointer to a control module's frame can only be assigned to a suitable variable. To delcare a program variable suitable for assigning the value returned by "NEW *Config*", *LoadProg* would have to do one of the following:

(a) include *config*'s control module (call it *Control*) in its DIRECTORY section, or

(b) declare itself a suitable program type for *Control.*

A configuration instance itself cannot be STARTed, 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:

```
START someInstance [ ComponentList ! CatchPhrase ]
STOP [ ! CatchPhrase ]
```

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 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 *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 **CatchPhrase** to the call; then if the allocator generates the signal *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: 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 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:

```
SignalTC        ::= SignalOrError ParameterList RETURNS ResultList |
                    SignalOrError ParameterList |
                    SignalOrError RETURNS ResultList |
                    SignalOrError

SignalOrError   ::= SIGNAL | ERROR
```

For example, the signal *BlockTooLarge* might be defined to carry along with it two parameters, a *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 *BlockTooLarge* would look like

```
BlockTooLarge: SIGNAL[z: Zone, needed: CARDINAL]
    RETURNS[newStorage: DESCRIPTOR FOR ARRAY OF CARDINAL];
```

A signal variable contains a unique name at run time, which is a code identifying an *actual signal*, just as a procedure variable must be assigned an *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 "= CODE" in place of "= 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:

```
Initialization  ::= = CODE | ...
```

A signal is generated by using it in a **SignalCall** as shown in the syntax below:

```
Statement       ::= SignalCall | ...

SignalCall      ::= SIGNAL Call | ErrorCall

ErrorCall       ::= RETURN WITH ERROR Call |
                    ERROR Call |
                    ERROR                    -- special error
```

Call is defined in section 5.4, and the called **Expression** must have some signal type in this case. A **SignalCall** can be used as an **Expression** as well as a **Statement.** For example,

```
newblock ← SIGNAL BlockTooLarge[zone, n];
```

Thus, generating a signal or error looks just like a procedure call, except for the additional word, ERROR or SIGNAL.

Fine point:

Although it is not recommended, the keywords SIGNAL and ERROR may be omitted (except in the RETURN WITH ERROR construct). This makes the signal look exactly like a procedure call.

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.

8.2. 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:

```
CatchItem      ::= Catch |
                   ANY => Statement
Catch          ::= ExpressionList => Statement
CatchSeries    ::= CatchItem |
                   Catch ; CatchSeries
```

The expressions in the **ExpressionList** (semantically restricted to a list of variables) must evaluate to the names of signals (unless otherwise stated, we use *signal* to stand for both ERROR and SIGNAL). The special identifier ANY will match any signal (sec. §.2.3).

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, **SignalCall**, NEW, START, RESTART, STOP, JOIN, FORK, or WAIT (but not in a RESUME or RETURN). A catch phrase may also be appended to the BEGIN of a block or the DO of a loop statement by means of an **EnableClause**. The applicable syntax for a call and for a block or loop statement is

```
Call           ::= Variable [ ComponentList ! CatchSeries ] |
                   Variable [ ! CatchSeries ] |
                   ...
Block          ::= BEGIN                    -- (from Section 4.4)
                   OpenClause
                   EnableClause
                   DeclarationSeries
                   StatementSeries
                   ExitsClause
                   END
EnableClause   ::= empty |
                   ENABLE CatchItem ; |
                   ENABLE BEGIN CatchSeries END ; |
                   ENABLE BEGIN CatchSeries ; END ;
```

Note that the **EnableClause** is always followed by a semi-colon, and BEGIN...END must be used if there is more than one **Catch** in an **EnableClause.**

The main difference between the two kinds of catch phrases (ENABLE and !) is the scope of their influence. A catch phrase on a **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 *any* call in the block (or generated in the block).

Fine points:

> For a **Block** with an **ExitsClause** (see sec. 4.4.1), the catch phrases enabled at the beginning of the body are not in force in the statements of the **ExitSeries.**
>
> Procedures declared in the **DeclarationSeries** (of any enclosed **Block**) do not inherit the catch pharses in the **EnableClause.**

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:

```
-- 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 *Signaller*. *Signaller*'s definition is

Signaller: PROCEDURE[*s*: *SignalCode*, *m*: *Message*];

Here *s* identifies the signal being generated, and *m* contains its arguments. (Actually, different procedures are used to distinguish between SIGNAL, ERROR, and RETURN WITH ERROR.)

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, *beginning with the procedure which generated the signal itself.* If the caller of a procedure is the outermost block of code for a program, the *Signaller* will follow its *return link* to continue propagating the signal (the return link points to the frame which last STARTed the module (sec. 7.8)).

If, in place of SIGNAL or ERROR, a RETURN WITH ERROR is used, the procedure which generated the error is first deleted (after releasing the monitor lock, if it is an ENTRY procedure), and propagation of the error begins with its caller.

As *Signaller* considers each frame, it looks to see whether that frame has any enabled catch phrases; if so, *Signaller calls* the innermost catch phrase as if it were a procedure, passing it the *SignalCode* and *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 ENABLE in the innermost enclosing block which contains that call.

Because signals can be propagated right through the call hierarchy, the programmer must consider catching not only signals generated *directly* within any procedure that is called, but also any generated *indirectly* as a result of calling that procedure. Indirect signals are those generated by procedures called from within a procedure which you call, unless they are stopped before reaching you.

When a catch phrase is called, it behaves like a SELECT statement: it compares the signal code passed to it with each signal value in the **ExpressionList** of each **Catch** in the catch phrase. If the signal code matches one of the signal values, control enters the statement following the "=>" for that **Catch**; if not, the next **Catch** is tried. A **Catch** consisting of "ANY => statement" automatically matches any signal code (and is the only way to catch the unnamed ERROR generated by the standalone ERROR statement discussed in section 8.1).

Fine point:

> The ANY catchall is intended primarily for use by the debugger, and should generally be avoided. It matches any signal, including 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 **Catch** is said to have *caught* or *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 *Signaller* with a value indicating that the signal was rejected, and *Signaller* propagates the signal to the next level in the call hierarchy. In fact, all catch phrases are called by *Signaller* as if they were procedures of the following type:

```
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). 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) 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 point:

> The UNWIND sequence gives each activation which 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:

```
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 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:

```
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.

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:

```
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

ResumeError: PUBLIC ERROR;

Since it is an ERROR, one cannot legally RESUME it.

8.2.6. *Examples of* CONTINUE, RETRY, *and* RESUME

The following example illustrates the use of CONTINUE, RETRY, and RESUME. The procedure *GetItem* reads characters from some source (using the procedure *ReadChar*) and collects a single *item*, which it puts into the string *s*. An item is a string of characters ending with a CR, SP, or TAB. On rare occasions, a terminating character may be included in an item; this is done by preceding it with an "escape" character. If two escape characters occur together, the current item is forgotten, and *GetItem* essentially starts over.

If an escape character is followed by any character other than a terminator or itself, scanning stops and that character is returned instead of the usual CR, SP, or TAB. If the string is filled before a terminating character is encountered, the error *StringBoundsFault* is generated.

```
g1: -- StringBoundsFault: SIGNAL[STRING] RETURNS[STRING];    (in StringDefs)
g2: -- ReadChar: PROCEDURE RETURNS[CHARACTER];               (in IODefs)
g3: GetItem: PUBLIC PROCEDURE[s: STRING] RETURNS[z: CHARACTER] =
g4:
g5:     BEGIN
g6:     i: CARDINAL;
g7:     Escape: SIGNAL = CODE;
g8:
g9:     BEGIN ENABLE Escape =>
g10:        SELECT z ← IODefs.ReadChar FROM
g11:            CR, SP, TAB  => RESUME;          -- (at g20)
g12:            escMark      => RETRY;           -- (at g9)
g13:            ENDCASE  => CONTINUE;            -- (at g26)
g14:    FOR i IN [0..s.maxlength)
g15:        DO
g16:        SELECT z ← IODefs.ReadChar FROM
g17:            CR, SP, TAB  => EXIT;
g18:            escMark  => SIGNAL Escape;
g19:            ENDCASE;
g20:        s[i] ← z;
g21:        s.length ← i+1;
g22:        REPEAT
g23:            FINISHED => [] ← ERROR StringDefs.StringBoundsFault[s];
g24:        ENDLOOP;
g25:    END;
g26:    RETURN[z];
g27:    END;
```

This example is used only for illustration; generally, signals like *Escape* which are intended to be caught only in the procedure in which they are generated are a bad idea; they are inefficient compared with conditional statements and normal transfers of control, and a programming error may cause them to propagate beyond the procedure defining them.

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.

CHAPTER 9.

PORTS AND CONTROL STRUCTURES

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 a 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

```
    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

```
    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 simple 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:

```
    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
    FileDefs: FROM "filedefs" USING [
        NUL, FileHandle, FileAccess, OpenFile, ReadChar, EndOfFile, CloseFile];

ReadFile: PROGRAM[name: STRING] IMPORTS FileDefs =
BEGIN OPEN FileDefs;
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
    FileDefs: FROM "filedefs" USING [
        NUL, FileHandle, FileAccess, OpenFile, WriteChar, CloseFile];

WriteFile: PROGRAM[name: STRING] IMPORTS FileDefs =
BEGIN OPEN FileDefs;
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, they must both be instantiated and initialized, 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, ReadFile, WriteFile =
BEGIN OPEN IODefs;
input: STRING ← [256];
output: STRING ← [256];
reader: POINTER TO FRAME[ReadFile];
writer: POINTER TO FRAME[WriteFile];
-- 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;
reader ← NEW ReadFile; START reader[input];
writer ← NEW WriteFile; START writer[output];
-- connect their ports and then restart them to get them synchronized
CONNECT writer.In TO reader.Out;
CONNECT reader.Out TO writer.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 StartTrap 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:

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:

```
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.

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:

```
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 him 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, and 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):

```
. . .
-- 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 which 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:

```
. . .
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,

```
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

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.

CHAPTER 10.

PROCESSES AND CONCURRENCY

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:

```
ReadLine: PROCEDURE [s: STRING] RETURNS [CARDINAL] =
    BEGIN
    c: CHARACTER;
    s.length ← 0;
    DO
        c ← ReadChar[];
        IF ControlCharacter[c] THEN DoAction[c]
        ELSE AppendChar[s,c];
        IF c = CR THEN RETURN [s.length];
        ENDLOOP;
    END;
```

The call

```
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 ← FORK ReadLine[buffer];
. . .
<concurrent computation>
. . .
n ← JOIN p;
```

This allows the statements labeled <concurrent computation> to proceed in parallel with user typing (clearly, the concurrent computation should not reference the string *buffer*). The FORK construct spawns a new process whose result type matches that of *ReadLine*. (*ReadLine* is referred to as the "root procedure" of the new process.)

```
p: PROCESS RETURNS [CARDINAL];
```

Later, the results are retrieved by the JOIN statement, which also deletes the spawned process. Obviously, this must not occur until both processes are ready (i.e. have reached the JOIN and the RETURN, respectively); this rendevous is synchronized automatically by the process facility.

Note that the types of the arguments and results of *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 PROCESS is similar to the declaration of a PROCEDURE, except that only the return record is specified. The syntax is formally specified as follows:

TypeConstructor	::=	... | **ProcessTC**	
ProcessTC	::=	PROCESS **ReturnsClause**	
ReturnsClause	::=	**empty** | RETURNS **ResultList**	-- from sec. 5.1.
ResultList	::=	**FieldList**	-- from sec. 5.1.

Suppose that *f* is a procedure and *p* a process. In order to fork *f* and assign the resulting process to *p*, the **ReturnClause** of *f* and that of *p* must be compatible, as described in sec 5.2.

The syntax for the FORK and JOIN statements is straightforward:

Statement	::=	... | **JoinCall**
Expression	::=	... | **ForkCall** | **JoinCall**
ForkCall	::=	FORK **Call**
JoinCall	::=	JOIN **Call**
Call	::=	(see sections 5.4 and 8.2.1)

The **ForkCall** always returns a value (of type PROCESS) and thus a FORK cannot stand alone as a statement. Unlike a procedure call, which returns a RECORD, the value of the FORK cannot be discarded by writing an empty extractor. The action specified by the 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 **ReturnsClause.** 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 catchphrase can be attached to either a FORK or JOIN by specifying it in the **Call.** Note, however, that such a catchphrase 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 asynchonous 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:

```
MONITORLOCK: TYPE = 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: *locked* = FALSE,
2. setting: *locked* ← 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 procedure that do operations on the data. *Monitor modules* are declared much like program or definitions modules; for example:

```
M: MONITOR [arguments] =
    BEGIN
    . . .
    END.
```

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: ENTRY PROCEDURE [arguments] = . . .
```

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: INTERNAL PROCEDURE [arguments] = . . .
```

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: PROCEDURE [arguments] = . . .
```

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
↓
External procedure(s) -- inside module but outside monitor
↓
Entry procedure -- inside monitor
↓
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 readibility of a monitor module is to structure the source text in the corresponding order:

```
    M: MONITOR =
        BEGIN
        <External procedures>
        <Entry procedures>
        <Internal procedures>
        <Initialization (main-body) code>
        END.
```

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:

```
MDefs: DEFINITIONS =
    BEGIN
    M: PROGRAM [arguments];
    P, Q: PROCEDURE [arguments] RETURNS [results];
    .
    .
    END.
```

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:

```
Statement      ::= ... | WaitStmt | NotifyStmt
WaitStmt       ::= WAIT Variable OptCatchPhrase
NotifyStmt     ::= NOTIFY Variable | BROADCAST Variable
```

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:

```
WHILE ~BooleanExpression DO
   WAIT c
   ENDLOOP;
```

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 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:

```
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:

```
StorageAllocator: MONITOR =
    BEGIN
    StorageAvailable: CONDITION;
    . . .

    Allocate: ENTRY PROCEDURE [size: CARDINAL] RETURNS [p: POINTER] =
        BEGIN
        UNTIL <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.
```

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.

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*:

```
LOCK: MONITORLOCK;
```

This implicit variable is declared automatically in the global frame of any module whose heading is of the form:

```
M: MONITOR [arguments]
   IMPORTS . . .
   EXPORTS . . . =
```

In such a monitor it is generally not necessary to mention *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 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:

```
ProgramTC     ::= ... | MONITOR ParameterList ReturnsClause LocksClause
LocksClause   ::= empty | LOCKS Expression |
                  LOCKS Expression USING identifier : TypeSpecification
```

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: MONITORED RECORD [x: INTEGER, . . . ];
```

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:

```
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 . . . =
        BEGIN
        . . .
        p2: PUBLIC ENTRY PROCEDURE . . .
            x ← root.monitorDatum1;  -- access the protected data of the monitor
        . . .
        END.

    BigMonB: MONITOR
        LOCKS root
        IMPORTS root: BigMonRoot . . . EXPORTS . . . =
        BEGIN OPEN root;
        . . .
        p3: PUBLIC ENTRY PROCEDURE . . .
            monitorDatum2 ← . . .;  -- access the protected data via an OPEN
        . . .
        END.
```

The monitor *BigMon* is implemented by three modules. The modules *BigMonA* and *BigMonB* have a LOCKS clause to specify the location of the monitor lock: in this case, the distinguished variable *LOCK* in *BigMonRoot*. When any of the entry procedures *p*1, *p*2, or *p*3 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.

Another means of implementing multi-module monitors is by means of a MONITORED RECORD. Use of OPEN allows the fields of the record to be referenced without qualification. Such a monitor is written as:

```
    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.
```

The LOCKS clause in the heading of this module (and each other module of this monitor) leads to a MONITORED RECORD. Of course, in all such multi-module monitors, the LOCKS clause will involve one or more levels of indirection (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 LOCKS clause can evaluate to a MONITORLOCK (i.e. the example above is equivalent to writing "LOCKS *pm.LOCK*").

CAUTION: 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 *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 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 instance wishes to call an internal procedure in another instance. (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 monitor 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 insure 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:

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 delarations 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.

10.5. 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.

```
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:

```
Failure: ERROR [kind: CARDINAL] = CODE;

Proc: ENTRY PROCEDURE [. . .] RETURNS [c1, c2: CHARACTER] =
    BEGIN
    ENABLE UNWIND => . . .
    . . .
    IF cond1 THEN ERROR Failure[1];
    IF cond2 THEN RETURN WITH ERROR Failure[2];
    . . .
    END;
```

Execution of the construct ERROR *Failure*[1] raises a signal that propagates until some catch phrase specifies an exit. At that time, unwinding begins; the catch phrase for UNWIND in *Proc* is executed and then *Proc*'s frame is destroyed. Within an entry procedure such as *Proc*, the lock is held until the unwind (and thus through unpredictable computation performed by catch phrases).

Execution of the construct RETURN WITH ERROR *Failure*[2] releases the monitor lock and destroys the frame of *Proc* before propagation of the signal begins. Note that the argument list in this construct is determined by the declaration of *Failure* (not by *Proc*'s RETURNS clause). The catch phrase for UNWIND is not executed in this case. The signal *Failure* is actually raised by the system, after which *Failure* propagates as an ordinary error (beginning with *Proc*'s caller).

When the RETURN WITH ERROR construct is used from within an internal procedure, the monitor lock is *not* released; RETURN WITH ERROR will release the monitor lock in precisely those cases that RETURN will.

Another important issue regarding signals is the handling of UNWINDS; any entry procedure that may experience an UNWIND must catch it and clean up the monitor data (restore the monitor invariant):

```
P: ENTRY PROCEDURE [ ... ] =
    BEGIN ENABLE UNWIND => BEGIN <restore invariant> END;
    .
    .
    .
    END;
```

At the end of the 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 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 UNWIND (directly or indirectly). If the monitor invariant is already satisfied, no further cleanup need be specified, but *the null catch-phrase must be written* so that the compiler will generate the code to unlock the monitor:

```
BEGIN ENABLE UNWIND => NULL;
```

This should be omitted *only* when it is certain that no UNWINDs can occur.

Another point is that signals caught by the **OptCatchPhrase** of a 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*
p↑	*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* ...
f[*x*, *y*, *z*]	*f* of *x*, *y* and *z*
!	enabling

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., ln 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.):

```
lnOld, lnNew, lnBuffer: 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.

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.

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:

minINTEGER = -32768 = -2^{15} and has internal representation 100000B
maxINTEGER = 32767 = $2^{15}-1$ and has internal representation 077777B
maxCARDINAL = 65535 = $2^{16}-1$ and has internal representation 077777B
minLONGINTEGER = -2147483648 = -2^{31}
maxLONGINTEGER = 2147483647 = $2^{31}-1$

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.
```

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 ↑α. In addition, a number of special characters such as SP (space), DEL (rubout) are denoted by their generally accepted names.

Octal Value	Character Name(s)	Octal Value	Character Name(s)
000C	NUL	100C	'@
001C	↑A	101C	'A
002C	↑B	102C	'B
003C	↑C	103C	'C
004C	↑D	104C	'D
005C	↑E	105C	'E
006C	↑F	106C	'F
007C	↑G, BELL	107C	'G
010C	↑H, BS	110C	'H
011C	↑I	111C	'I
012C	↑J, LF	112C	'J
013C	↑K	113C	'K
014C	↑L	114C	'L
015C	↑M, CR	115C	'M
016C	↑N	116C	'N
017C	↑O	117C	'O
020C	↑P	120C	'P
021C	↑Q	121C	'Q
022C	↑R	122C	'R
023C	↑S	123C	'S
024C	↑T	124C	'T
025C	↑U	125C	'U
026C	↑V	126C	'V
027C	↑W	127C	'W
030C	↑X	130C	'X
031C	↑Y	131C	'Y
032C	↑Z	132C	'Z
033C	ESC	133C	'[
034C		134C	'\
035C		135C	']
036C		136C	'↑
037C		137C	'←
040C	' , SPace	140C	
041C	'!	141C	'a
042C	'"	142C	'b
043C	'#	143C	'c
044C	'$	144C	'd
045C	'%	145C	'e
046C	'&	146C	'f
047C	'', a single quote	147C	'g
050C	'(	150C	'h
051C	')	151C	'i
052C	'*	152C	'j
053C	'+	153C	'k
054C	',	154C	'l
055C	'-	155C	'm
056C	'.	156C	'n
057C	'/	157C	'o
060C	'0	160C	'p
061C	'1	161C	'q
062C	'2	162C	'r
063C	'3	163C	's
064C	'4	164C	't
065C	'5	165C	'u
066C	'6	166C	'v
067C	'7	167C	'w
070C	'8	170C	'x
071C	'9	171C	'y
072C	':	172C	'z
073C	';	173C	'{
074C	'<	174C	'|
075C	'=	175C	'}
076C	'>	176C	'~
077C	'?	177C	DEL

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.

```
DIRECTORY AltoDefs: FROM "altodefs";
DEFINITIONS FROM AltoDefs;
StringDefs: DEFINITIONS =
BEGIN
-- C O M P I L E - T I M E   C O N S T A N T S   A N D   T Y P E S
SubStringDescriptor: TYPE = RECORD
    [
    base: STRING,
    offset, length: CARDINAL
    ];
SubString: TYPE = POINTER TO SubStringDescriptor;
-- I N T E R F A C E   I T E M S
Overflow: SIGNAL;
InvalidNumber: SIGNAL;
StringBoundsFault: SIGNAL [s: STRING] RETURNS [ns: STRING];
WordsForString: PROCEDURE [nchars: INTEGER] RETURNS [INTEGER];
AppendChar: PROCEDURE [s: STRING, c: CHARACTER];
AppendString: PROCEDURE [to,from: STRING];
EqualString, EqualStrings: PROCEDURE [s1, s2: STRING] RETURNS [BOOLEAN];
EquivalentString, EquivalentStrings: PROCEDURE [s1, s2: STRING] RETURNS [BOOLEAN];
AppendSubString: PROCEDURE[to: STRING, from: SubString];
EqualSubString, EqualSubStrings: PROCEDURE [s1, s2: SubString] RETURNS [BOOLEAN];
EquivalentSubString, EquivalentSubStrings: PROCEDURE [s1, s2: SubString] RETURNS [BOOLEAN];
DeleteSubString: PROCEDURE [s: SubString];
StringToDecimal: PROCEDURE [STRING] RETURNS [INTEGER];
StringToOctal: PROCEDURE [STRING] RETURNS [UNSPECIFIED];
StringToNumber: PROCEDURE [STRING, CARDINAL] RETURNS [UNSPECIFIED];
StringToLongNumber: PROCEDURE [STRING, CARDINAL] RETURNS [LONG INTEGER];
AppendDecimal: PROCEDURE [STRING, INTEGER];
AppendOctal: PROCEDURE [STRING, UNSPECIFIED];
AppendNumber: PROCEDURE [STRING, UNSPECIFIED, CARDINAL];
AppendLongNumber: PROCEDURE [STRING, LONG INTEGER, 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 **CPacking** and **CLinks** clauses (section 7.7).

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). *Packing is a tradeoff, and should be used with caution.*

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 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.

```
ConfigDescription  ::= Directory CPacking Configuration .
CPacking           ::= empty | CPackSeries ;
CPackList          ::= PACK IdList
CPackSeries        ::= CPackList | CPackSeries ; CPackList
```

Each **PackList** defines a single segment; the code for all the modules in the **IdList** will be packed into it. The identifiers in the **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 **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.

```
CLinks        ::= empty | LINKS : CODE | LINKS : 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.

```
CRightSide    ::= Item | Item [ ] CLinks | Item [ IdList ] 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.

```
        CHead           ::= CONFIGURATION CLinks Imports CExports ControlClause
```

This construction works much like the PUBLIC / 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 overriden for a particular module (or nested configuration) by specifying a different link option.

D.2.2. Restrictions

This scheme has the consequence that, *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.

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 *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 (leading almost certainly to utter confusion).

ABS
AND
ANY
ARRAY
BASE
BEGIN
BOOLEAN
BROADCAST
CARDINAL
CHARACTER
CODE
COMPUTED
CONDITION*
CONTINUE
DECREASING
DEFINITIONS
DEPENDENT
DESCRIPTOR
DIRECTORY
DO
ELSE
ENABLE
END
ENDCASE
ENDLOOP
ENTRY
ERROR
EXIT
EXITS
EXPORTS
FALSE*
FINISHED
FIRST
FOR
FORK
FRAME
FROM
GO
GOTO
IF
IMPORTS
IN
INCREASING
INTEGER
INTERNAL
JOIN
LAST
LENGTH
LOCKS
LONG
LOOP
LOOPHOLE
MACHINE
MAX
MEMORY
MIN
MOD
MONITOR
MONITORED
MONITORLOCK*
NEW
NIL*
NOT
NOTIFY
NULL
OF
OPEN
OR
ORDERED
OVERLAID
PACKED
POINTER
PORT
PRIVATE
PROCEDURE
PROCESS
PROGRAM
PUBLIC
REAL*
RECORD
REGISTER
RELATIVE
REPEAT
RESTART
RESUME
RETRY
RETURN
RETURNS
SELECT
SHARES
SIGNAL
SIZE
START
STATE
STOP
STRING
StringBody*
THEN
THROUGH
TO
TRANSFER
TRUE*
TYPE
UNSPECIFIED*
UNTIL
UNWIND*
USING
WAIT
WHILE
WITH
WORD*

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. each head their respective sections.

CompilationUnit ::=

Directory
DefinitionsFrom
identifier : ModuleHead = GlobalAccess
ModuleBody

Directory	::=	empty | DIRECTORY IncludeList ;	
DefinitionsFrom	::=	empty | DEFINITIONS FROM IdList ;	
ExportsList	::=	empty | EXPORTS IdList	
FileName	::=	stringLiteral	
GlobalAccess	::=	Access	-- in TypeSpecification
IdList	::=	identifier | IdList , identifier	
ImportsList	::=	empty | IMPORTS InterfaceList	
IncludeList	::=	identifier : FROM FileName | IncludeList , identifier : FROM FileName	
InterfaceItem	::=	identifier | identifier : identifier	
InterfaceList	::=	InterfaceItem | InterfaceList , InterfaceItem	
ModuleBody	::=	Block .	-- in Statement
ModuleHead	::=	ProgramTC ImportsList ExportsList ShareList | DEFINITIONS ShareList	
ModuleParams	::=	empty | [NamedFieldList]	-- in TypeSpecification
ShareList	::=	empty | SHARES IdList	

TypeSpecification ::=

PredefinedType |
TypeIdentifier |
TypeConstructor

Access	::=	empty | PUBLIC | PRIVATE	
Adjective	::=	identifier	
ArrayDescriptorTC	::=	DESCRIPTOR FOR TypeSpecification | DESCRIPTOR FOR PackingOption ARRAY OF TypeSpecification	
ArrayTC	::=	PackingOption ARRAY IndexType OF TypeSpecification	
BaseOption	::=	empty | BASE	
ByteList	::=	Expression | ByteList , Expression	
CommonPart	::=	empty | NamedFieldList ,	
ConstantList	::=	Expression | ConstantList , Expression	
ElementType	::=	INTEGER | CARDINAL | CHARACTER | BOOLEAN | EnumerationTC | SubrangeTC	
EnumerationTC	::=	{ IdList }	
FieldList	::=	[UnnamedFieldList] | [NamedFieldList]	
IndexType	::=	ElementType | TypeIdentifier	
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 |
                   NamedFieldList , IdList : Access TypeSpecification
OptionalInterval ::= empty | Interval
Ordered         ::= empty | ORDERED
PackingOption   ::= empty | PACKED
ParameterList   ::= empty | FieldList
PointerTail     ::= empty |
                   TO TypeSpecification |
                   TO FRAME [ identifier ]
PointerTC       ::= Ordered BaseOption POINTER OptionalInterval PointerTail
PortTC          ::= PORT ParameterList ReturnsClause |
                   RESPONDING PORT ParameterList ReturnsClause
PredefinedType  ::= INTEGER | CARDINAL | LONG INTEGER |
                   REAL | BOOLEAN | CHARACTER | STRING |
                   MONITORLOCK | CONDITION |
                   UNSPECIFIED | WORD
ProcedureBody   ::= Block                                  -- in Statement
ProcedureTC     ::= PROCEDURE ParameterList ReturnsClause
ProcessTC       ::= PROCESS ReturnsClause
ProgramTC       ::= PROGRAM ParameterList ReturnsClause |
                   MONITOR ParameterList ReturnsClause LocksClause
RecordTC        ::= MonitoredOption MachineDependent RECORD [ VariantFieldList ]
RelativeTC      ::= TypeIdentifier RELATIVE TypeSpecification |
                   TypeIdentifier RELATIVE LONG TypeSpecification
ReturnsClause   ::= empty | RETURNS FieldList
SignalOrError   ::= SIGNAL | ERROR
SignalTC        ::= SignalOrError ParameterList ReturnsClause
SubrangeTC      ::= Interval | TypeIdentifier Interval
Tag             ::= identifier : Access TagType |
                   COMPUTED TagType |
                   OVERLAID TagType
TagType         ::= TypeSpecification | *
TypeConstructor ::= ArrayDescriptorTC | ArrayTC | EnumerationTC | LongTC |
                   PointerTC | PortTC | ProcedureTC | ProcessTC |
                   RecordTC | RelativeTC | SignalTC | SubrangeTC
TypeIdentifier  ::= identifier |
                   identifier . identifier |
                   Adjective TypeIdentifier
UnnamedFieldList  ::= TypeSpecification |
                     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 | StartStmt | RestartStmt |
                  StopStmt | WaitStmt
AdjectiveList   ::= Adjective | AdjectiveList , Adjective     -- in TypeSpecification
Assignation     ::= FOR identifier ← Expression , Expression
AssignmentStmt  ::= LeftSide ← RightSide |                    -- LeftSide, RightSide in Expression
                    Extractor ← RightSide
Block           ::= BEGIN
                    OpenClause
                    DeclarationSeries
                    EnableClause
                    StatementSeries
                    ExitsClause
                    END
Call            ::= Variable |                                -- in Expression
                    Variable [ ComponentList ] |              -- ComponentList in Expression
                    Variable [ ComponentList ! CatchSeries ]
                    Variable [ ! CatchSeries ]
Catch           ::= IdList => Statement                       -- IdList in CompilationUnit
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 EntryOption TypeSpecification Initialization ; | -- Access in TypeSpecification
                    IdList : Access TYPE = Access TypeSpecification ;
DeclarationSeries   ::= empty | DeclarationSeries Declaration
Direction       ::= empty | INCREASING | DECREASING
ElseClause      ::= empty | ELSE Statement
EnableClause    ::= ENABLE CatchItem ; |
                    ENABLE BEGIN CatchSeries END ; |
                    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 | Assignation
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 | TypeIdentifier | 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 | OpenList , 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 |
                    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
StartStmt       ::= 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
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
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 | 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
LeftSide          ::= identifier | Call |                    -- Call in Statement
                      IndexedAccess | QualifiedAccess | IndirectAccess |
                      LOOPHOLE [ Expression ] |
                      LOOPHOLE [ Expression , TypeSpecification ] |
                      MEMORY [ Expression ] | REGISTER [ Expression ]    -- not in this Manual
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 | @ 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
                             ENDCASE => Expression
SelectExprVariant    ::= WITH OpenItem SELECT TagItem FROM -- OpenItem, TagItem in Statement
                             ChoiceList
                             ENDCASE => Expression
Subrange          ::= SubrangeTC | TypeIdentifier          -- SubrangeTC, 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.