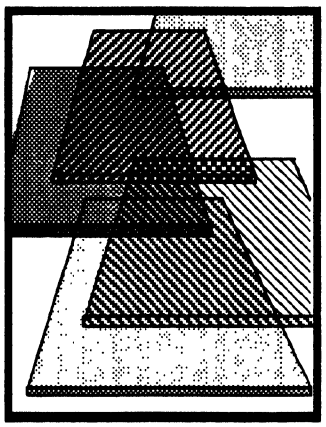
XEROX

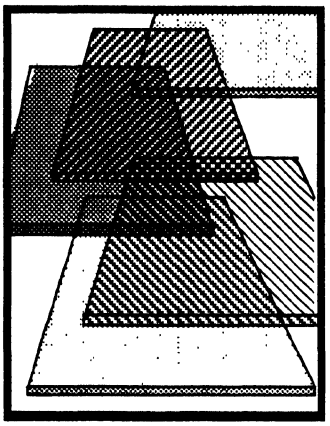
Xerox Development Environment



Mesa Course

September 1988
610E00230

Xerox Corporation
Document Systems Business Unit
XDE Technical Services
475 Oakmead Parkway
Sunnyvale, California 94086



Introduction

The *Mesa Course* is a self-paced programming tutorial intended to give you hands-on experience with applications and systems programming in the Xerox Development Environment. The course introduces important concepts, illustrates those concepts with extensive examples, and provides exercises to ensure your familiarity with those concepts. The Mesa Course is intended for use at any XDE customer site.

The twenty one chapters of the Mesa Course are grouped into two major sections: the Mesa Language and the "Tajo" development environment. The experienced professional need only skim the Mesa Language chapters and can begin with serious study of the development environment, referring to language issues in the first section as required. The less experienced programmer should work through the material sequentially. The initial section of the course is designed to present Mesa programming to someone who is familiar with other structured languages, particularly Pascal, and has completed the *Introduction to XDE* on-line tutorials.

The Mesa Language section introduces you to Mesa programming concepts and essential components of the Xerox Development Environment. You will learn how to develop and run programs in our environment, including how to:

- convert standard Pascal constructs into their Mesa counterparts,
- use Mesa's interface mechanism to integrate independently developed programs and share information among them,
- allocate dynamic storage from a common pool,
- declare and manipulate strings, dynamic arrays, and variant records
- use processes and monitors effectively,
- handle exception occurrences via a software interrupt mechanism,
- debug your program when things go awry, and
- use the Mesa reference manuals to find the information you need.

Upon completing the first section you should have a well-grounded understanding of how to use Mesa and the development environment.

The last half of the course emphasizes advanced features of XDE and concentrates on fundamental aspects of tool creation. In this section you will learn how to

- write programs that run in the Executive window,
- interact with the Mesa file system including performing file I/O and attaching a stream to a file,
- allocate space from virtual memory and map it to a backing file,
- use the form subwindow layout tool to generate "standard" tool subwindow implementation code,
- implement tool features not provided by the form subwindow layout tool,
- handle terminal input for a tool, and
- paint into the windows of a tool

If you do not intend to be an active Mesa programmer, then this course is probably not for you. The *Introduction to XDE* on-line tutorials provide an explanation of the non-programming aspects of the development environment, and may be what you want.

Course structure

The course consists of twenty one chapters, six appendices, and a Glossary. The early chapters, Chapters 1 through 10, each concentrate on a single concept and build on the previous chapters. If this material is appropriate for your experience level, you should study each of these in order. The chapters of the environment section, from Chapter 11 on, are somewhat more independent and self-standing. Chapter 12 deals with the Executive, chapters 13 through 15 deal with aspects of the file system, chapters 16 through 19 cover fundamental aspects of tool construction, and chapters 20 and 21 discuss gathering input for tools and painting tool windows.

Some of the appendices cover basic debugging techniques. The remaining appendices, answers to questions, and the Glossary should be referenced as needed. The course suggests points when studying the appendices might be most helpful to you.

How to read a chapter

For the most part, each chapter contains the following sections in the following order:

- An *introduction* covering what it is about, what you will learn from it, and what you will do in it.
- A description of *preliminary readings* and where to find them. These are usually the sections in the reference documentation that describe the concepts to be discussed. You should read, but not disect, this information. We discuss the depth to which you should study these readings in the next section,Using the Course.
- A *glossary* of terms, which defines the terms new to that particular section.

- A *discussion* of the chapter's main topic. This section is the main body of the chapter. It usually takes the form of a general introduction to the concept, a discussion of the facilities you need, and at least one programming example.

- A *summary* of what you have learned. This helps you to check quickly that you have understood the major points of the chapter, and can later serve as a reference.

- A discussion of *style*-related issues related to the concept being learned. The section explains the choice and type of coding style used in the examples.

- A description of *reference materials* and where to find them. These are usually collected journal articles that relate to the concept being taught. Using these materials will extend the breadth of your knowledge or give you a different perspective on the topic.

- A set of *questions*. Questions and answers are provided so you can judge how well you have understood the material. The answers are collected in an appendix.

- A programming *exercise* that applies the new concept and provides experience with the Mesa language. It is primarily through these exercises, as well as through programming examples and readings in the *Mesa Language Manual*, and the *Mesa* and *Pilot Programmers Manuals*, that you will become familiar with the XDE.

Using the course

Beginning users of Mesa come with a wide range of experience. You can use the following guidelines to gauge the level appropriate for you and how best to use this course.

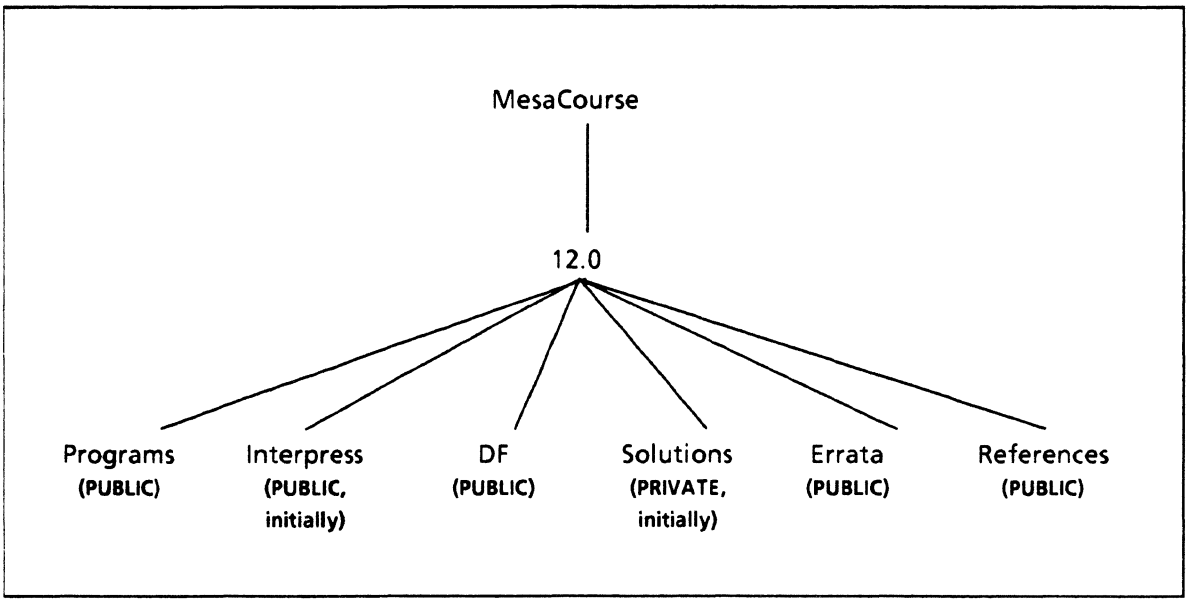
The primary purpose of this training is to initiate you to programming in the Xerox development environment. This environment is documented by well over one thousand pages of material. You need to know how to find, use, and understand information in these documents. The course presents the information in the reference materials around a framework of examples and exercises. There is no information in the course that is not also in at least one other document.

Many chapters ask you to do preliminary readings in reference manuals. If you understand the reference materials easily, then the chapter will not provide you with any more information. Instead, you may find it best, after completing the preliminary readings, to skim the chapter, check your understanding via the questions, and go straight to the exercises. On the other hand, if you find the reference readings overly difficult, do not pore over them. Instead, skim them and concentrate your efforts on the discussion section of the appropriate Mesa Course chapter. After you have finished the chapter, go back and re-read the reference material. This will give you more information on the subject, and will also give you experience in using the manuals.

Getting Started

This is version 12.0 of the Mesa Course. It assumes that you are using a Dandelion or Daybreak processor running the Sequoia release (12.0) of the Xerox Development

Environment with Tajo installed on a normal volume, CoPilot serving as a debugger for the volume on which Tajo is installed, and a User.cm that is set up for this configuration.



The Mesa Course Directory Structure

Interpress masters for the course text are stored electronically in the folder [CustomerNSFileServer]<MesaCourse>12.0>Interpress>. You can print copies of the course from these folders as you need them (universities may have this folder protected). Your local support group may have bound copies of the Mesa Course available.

The programs discussed in the chapters are stored in the [...] <...> ...> Programs>ChapterName(ChapterNumber) folder for each chapter. Retrieve all files from this folder before starting a chapter, e.g., retrieve all the files in [CustomerNSFileServer]<MesaCourse>12.0>Programs>Interfaces(2) before starting Chapter 2.

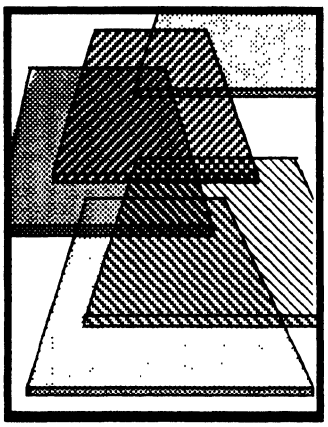
Solutions to programming exercises are stored in the [...] <...> ...> Solutions> folder. Your XDE training liaison will decide who has access rights to this folder: it may be read protected initially.

There are two papers cited in the Mesa Course that are not part of the XDE release documentation. They can be found in the [...] <...> ...> References> folder.

The Mesa Course is still under development, and we would appreciate your comments and corrections. We apologize for any inconveniences caused by inconsistencies or inaccuracies that have escaped our current review. Please check on [...] <...>...> Errata> for any update information.

If you run into any trouble getting started or while you are going through the course, do not hesitate to ask your XDE training mentor for help. Initially, please ask your mentor to make sure that your disk and User.cm are compatible with the course, and for the name of a CustomerNSFileServer near you that has a copy of the <MesaCourse> folder.

1



From Pascal to Mesa

This chapter will introduce you to the programming language Mesa by building on your knowledge of Pascal.

Pascal has become the instructional language of choice in the computer science academic community and is gaining in general popularity. It is a language that has integrated a small set of features into a powerful and efficient programming tool. One of Pascal's most attractive features is user-defined data types that enable data structuring capability and data abstraction. Standard Pascal does have a significant shortcoming in terms of writing a large system: there is no way to break the system down into small separately compiled units and then integrate them into a consistent whole. This prevents the compiler from checking the type correctness of actual parameters in distinct units, inhibits the development of "libraries" to extend the language, and generally complicates the implementation of large systems constructed by a group of programmers. Furthermore, standard Pascal does not support dynamic array bounds; it is difficult to write general routines that process arrays of different sizes. Standard Pascal has no exception handling facilities and does not support concurrent processes.

Mesa is a strongly typed, block structured programming language whose syntax is similar to that of Pascal. Mesa extends Pascal in a number of ways intended to make it more effective for the development of large systems, while preserving Pascal's data structuring and data abstraction facilities. We begin this chapter by examining the common ground between Pascal and Mesa: shared language concepts and constructs. Then we look at some of the ways in which Mesa differs from Pascal.

1.1 Definition of terms

Most of the concepts found in Pascal have counterparts in Mesa. The list below defines terms that are either distinctive to both Pascal and Mesa or terms whose Pascal and Mesa definitions differ slightly.

type definitions — *Type definitions* are the mechanism for describing data of Mesa programs.

name — A *name* (or identifier) is a sequence of alphabetic and numeric characters beginning with an alphabetic

character. Identifiers in Mesa can be up to 256 characters long; character case is significant in Mesa identifiers.

static variables — *Static variables* are variables for which an explicit variable declaration has been made.

dynamic variables — *Dynamic variables* are generated by a special procedure (NEW) that yields a pointer or reference value that subsequently serves in place of a name to refer to the variable.

strongly typed — The Mesa compiler uses static analysis to deduce the type of every constant, variable, and expression to ensure that all programs are type correct. Languages in which such type correctness is determined at compile time are called *strongly typed.*

procedural abstraction — A *procedural abstraction* is a mapping from a set of inputs to a set of outputs that can be described by a specification. The specification must show how the outputs relate to the inputs, but it does not reveal or imply the way the outputs are to be computed.

actual procedure — An *actual procedure* is a procedure initialized so that its meaning (defined by its body) cannot change. You cannot assign a value to an actual procedure.

procedure variable — A *procedure variable* is a procedure initialized in such a way that the procedure's value (body) can be changed by assignment.

1.2 A comparison of Mesa and Pascal constructs

This section presents a sequence of examples showing analogous Mesa and (standard) Pascal constructs.

Mesa	Pascal
Comments	
--This is a comment terminated by EOL	*{This is a comment}*
--This is a comment terminated by dashes--	
<<This is a comment extending over more than one line>>	*{This is a comment extending over more than one line}*

Mesa

```
Pi: REAL = 3.14;
  --Note
  -- Mesa is case sensitive.
  -- Reserved words are capitalized.
  -- Constants have explicit types.

MinusPi: REAL = -Pi;

linesPerPage: INTEGER = 60;

shortPage: INTEGER = linesPerPage - 6;

capA: CHARACTER = 'A;

smallA: CHAR = 'a;
--CHARACTER and CHAR are equivalent

message: LONG STRING = "Hello there";
--String literal allocated in global frame.

anotherMessage: LONG STRING = "Boo"L;
--The string literal is allocated in the local frame
--of the innermost procedure enclosing the
--literal. Thus, in Mesa you can choose whether
--to allocate from a local or global frame.
```

Pascal

```
CONST
  Pi = 3.14;
  {Pascal is not case sensitive.
   Capitalization is only for readability.
   Constants have implicit TYPE.}

  MinusPi = -Pi;

  linesPerPage = 60;

  {Pascal does not support general
   expression constants}

  capA = 'A';

  smallA = 'a';

  message = 'Hello there';
```

Constant declarations

Type declarations: One dimensional ARRAYs

```
Name: TYPE = ARRAY[0..9] OF CHAR;

packName: TYPE = PACKED ARRAY
  [0..9] OF CHAR;

Dashes: TYPE = ARRAY[0..7) OF CHAR ← ALL['-];
  --[0..n + 1) equivalent to [0..n]

RARRAY: TYPE = ARRAY[0..8) OF REAL;
```

```
TYPE
  Name = ARRAY[0..9] OF CHAR;

  packName = PACKED ARRAY[0..9] OF CHAR;

  Dashes = ARRAY[0..6] OF CHAR;
  {No default initialization}

  RARRAY = ARRAY[0..7] OF REAL;
```

Mesa

Pascal

Type declarations: Two dimensional ARRAYs

```
M3by4: TYPE = ARRAY[1..3] OF ARRAY[1..4]
  OF INTEGER ← ALL[0];
```

```
M3by4 = ARRAY[1..3] OF ARRAY[1..4]
    OF INTEGER;
{ No default initialization}

{or}

ALT3by4 = ARRAY[3,4] OF INTEGER;
{Compact representation of two dimensional ARRAY,
 no default initialization}
```

Type declarations: Records

```
Coordinate: TYPE = RECORD[
  horizontal: REAL ← 0.00;
  vertical: INTEGER ← 0];
  -- default field initialization

--or

Coordinate: TYPE = RECORD[
  horizontal: REAL,
  vertical: INTEGER] ← [0.00,0]
  -- default TYPE initialization
```

```
Coordinate =
    RECORD
        horizontal: REAL; {no initialization}
        vertical: INTEGER
    END;
```

Type declarations: Variant Records

```
Shape: TYPE = {point, line, circle};

Figure TYPE = RECORD[
  figureName: Name,
  specificFigure: SELECT fieldID: Shape FROM
    point = > [position: Coordinate],
    line = > [xCoef, yCoef, slope: REAL],
    circle = > [center: Coordinate,
                radius: REAL];
  ENDCASE];
```

```
Shape = (point, line, circle);

Figure =
    RECORD
        figureName: Name;
        CASE tag: Shape OF
            point :
                (postion: Coordinate);
            line:
                (xCoef, yCoef, slope: REAL);
            circle:
                (center: Coordinate;
                 radius: REAL);
    END;
```

Mesa	Pascal

Type declarations: Records containing pointers

```
personPtr: TYPE = LONG POINTER TO Person;
```

```
personPtr = ↑ Person;
```

```
Person: TYPE = RECORD[
  name: Name,
  age: [21..120].
  sex: {male, female},
  party: {Demo, GOP},
  contribution:[0..10000]];
```

```
Person =
  RECORD
    name: Name;
    age: 21..120;
    sex: (male, female);
    party: (Demo, GOP);
    contribution : (0..10000)
  END;
```

```
link: TYPE = LONG POINTER TO Node;
```

```
link = ↑ Node;
```

```
Node:TYPE = RECORD[
  voter: Person,
  next: link];
```

```
Node =
  RECORD
    voter: Person;
    next: link
  END;
```

Variable declarations

```
VAR
```

```
b: BOOLEAN ← TRUE;
--BOOLEAN and BOOL are equivalent
```

```
b:BOOLEAN; {no initialization possible}
```

```
li, lj: LONG INTEGER ← -7;
```

```
{no double precision or initialization}
```

```
i, j: INTEGER ← 41;
iSquared: INTEGER ← i*i;
k: INTEGER ← iSquared - i + 1;
```

```
i, j: INTEGER;
iSquared: INTEGER;
k: INTEGER;
{Initialization of iSquared and k must be done
 in statement section.}
```

```
a: RARRAY ;
```

```
a: RARRAY;
```

```
mxy: M3by4;
```

```
mxy: M3by4;
altmxy: ALT3by4
```

```
control: [1..15];
```

```
control: 1..15;
```

Mesa

Pascal

Variant record variables

```
figure: Figure;
```

```
figure, pointFigure, lineFigure, circleFigure: Figure;
```

"Bound" variant record variables

```
pointFigure: point Figure;
lineFigure: line Figure;
circleFigure: circle Figure;
```

```
{Pascal has no concept of bound variant RECORDS.}
```

Dynamic storage allocation

```
z: UNCOUNTED ZONE ← NIL;
--source of dynamically allocated objects
```

```
{Nodes are automatically allocated from a
 system heap}
```

Variables for pointer examples

```
cand1, cand2, cand3, cand4: Person;
preswinner, presloser, vpwinner,
vploser: personPtr;
p, rootNode: link;
```

```
cand1, cand2, cand3, cand4: Person;
preswinner, presloser, vpwinner
vploser: personPtr;
p, rootNode: link;
```

Procedure declarations

```
Fact: PROCEDURE[n: LONG INTEGER]
  RETURNS [LONG INTEGER] =
  BEGIN
    RETURN[IF n = 0 THEN 1
           ELSE n*Fact[n - 1]]
  END;
--Mesa does not differentiate between
--FUNCTION and PROCEDURE.
```

```
FUNCTION Fact(n: INTEGER): INTEGER;
  BEGIN
    IF n = 0 THEN Fact := 1
    ELSE Fact := n*Fact(n - 1)
  END;    {Fact}

{Pascal FUNCTIONS can only return "simple" TYPES,
 i.e., CHAR, INTEGER, and REAL.}
```

```
Swap: PROCEDURE[iptr, jptr:
  LONG POINTER TO INTEGER] =
  {temp: INTEGER;
   temp ← iptr↑;
   iptr↑ ← jptr↑;
   jptr↑ ← temp};
```

```
PROCEDURE Swap(var i, j: INTEGER);
  VAR t: INTEGER;
BEGIN
  t := i;
  i := j;
  j := t
END;
```

```
--All arguments are passed by value in Mesa:
--i.e., the value of an argument, not its address
--is assigned to the parameter. Of course, this
--value itself can be an address.
```

```
--In Mesa, a block can be delimited either by
--BEGIN ... END or by { ... }
```

Mesa

Statements

```
a[1] ← 3.8E6;
mxy[2][3] ← 7;


IF b THEN PROCEDURE1[];

IF i # j / 2
   THEN PROCEDURE1[]
   ELSE PROCEDURE2[];

a[1] ← IF boolvar1
   THEN 4.56
   ELSE 8.71;
--An IF expression

--control: [1..15];
SELECT control FROM
   1, IN [7..10] = > statement1;
   2, 5, >10     = > statement2;
   ENDCASE       = > statement3;


SELECT TRUE FROM
   boolvar1 = > statement1;
   boolvar2 = > statement2;
   ...
   boolvarn = > statementn;
ENDCASE;


a[1] ← SELECT control FROM
   1, IN [7..10] = > 1.12;
   2, 5, >10     = > -4.856;
   ENDCASE       = > 73.2;
--A SELECT expression

i: INTEGER ← 1;
WHILE i < 10
   DO ... i ← i + 1; ...ENDLOOP;
```

Pascal

```
   a[1] : = 3.8E6;
   mxy[2][3] : = 7;
   altmxy[2,3] : = 7;

IF b THEN PROCEDURE1;

IF i < > j div 2
   THEN PROCEDURE1
   ELSE PROCEDURE2;

IF boolvar1
   THEN a[1] : = 4.56
   ELSE a[1] : = 8.71;


{control: 1..15;}
CASE control OF
   1, 7, 8, 9, 10: statement1;
   2, 5, 11, 12, 13, 14, 15: statement2;
   3, 4, 6: statement3
END;

IF boolvar1 THEN
   statement1
ELSE IF boolvar2 THEN
   statement2
...
ELSE IF boolvarn THEN
   statementn;

CASE control OF
   1, 7, 8, 9, 10: a[1] : = 1.12;
   2, 5, 11, 12, 13, 14, 15: a[1] : = -4.856;
   3, 4, 6: a[1] : = 73.2
END;

i : = 1;          {assume i defined earlier}
WHILE i < 10 DO
   BEGIN ... i : = i + 1; ...END;
```

Statements continued

Mesa

```
i: INTEGER ← 1;
DO
  ...i ← i + 1; ...
  IF i > = 10 THEN EXIT;
ENDLOOP;
```

```
--The Mesa construct
--
--UNTIL condition DO
--   {StatementSeries};
--ENDLOOP;
--
--is similar to that of Pascal except that the
--condition is tested at the "top "of the LOOP
--and, if false, the LOOP is not executed. REPEAT
--is a Mesa reserved word whose semantics are
--not the same as Pascal REPEAT.
```

```
FOR i: INTEGER IN [1..n) DO
 ... sum ← sum + a[i]; ...
ENDLOOP;
```

Pascal

```
i : = 1;
REPEAT ... i: = i + 1; ...
UNTIL i ≥ 10;
```

```
{In the Pascal construct

   REPEAT StatementSeries
   UNTIL condition;

the condition is tested only after the StatementSeries
has been executed once, i.e., the test is at the "bottom"
of the LOOP.}
```

```
{i: INTEGER; defined earlier}
FOR i : = 1 to n - 1 DO
BEGIN ...sum := sum + a[i]; ...END;
```

Unbound variant record initialization

Mesa

```
figure.figureName ← ['a, 'r, 'b, 'i, 't, 'r, 'a, 'r, 'y];
WITH f: figure SELECT FROM
  point = > f.position ← [-1.37, 14];
  line = > {f.xCoef ← 2.81,
            f.yCoef ← 4.2,
            f.slope ← -.7};
  circle = > {f.center ← [0.00,3.00],
              f.radius ← 5.00};
ENDCASE;
```

```
--the variable figure must be renamed
--within the WITH statement
```

Pascal

```
figure.figureName[0] : = 'a';
figure.figureName[1] : = 'r'; ...
WITH figure DO
  CASE tag OF
    point: WITH position DO
      BEGIN horizontal : = -1.37;
            vertical : = 14;
      END;
    line: BEGIN
          xCoef : = 2.81;
          yCoef : = 4.2;
          slope : = -.7;
        END;
    circle: WITH center DO
      BEGIN horizontal : = 0.00;
            vertical : = 3.00;
            radius : = 5.00;
      END
  END;
```

Bound variant record initialization

Mesa

```
pointFigure.figureName ← ['p, 'o, 'i, 'n, 't, ' ,'1, ' ,' ];
pointFigure.point← [-1.37, 14];
```

Pascal

```
{Pascal has no notion of bound variants}
```

Mesa

Pascal

Some pointer examples

```
cand1 ← Person[
  name : Name['R, 'e, 'a, 'g, 'a, 'n, ' , ' , ' ],
  age : 72,
  sex : male,
  party : GOP,
  contribution : 0];
```

```
WITH cand1 DO
  BEGIN
    name[0] : = 'R'; name[1] : = 'e'; . . .
    age : = 72;
    sex : = male;
    party : = GOP;
    contribution : = 0;
  END;
```

```
--Similarly initialize cand2 to MondaleData,
--cand3 to BushData, and cand4 to FerraroData.
```

```
z← Heap.Create[initial:1];
--Initialize source FOR dynamically
--allocated objects
```

```
{Pascal allocation will be from an anonymous
 system heap.}
```

```
preswinner ← z.NEW[Person ← cand1];
presloser ← z.NEW[Person ← cand2];
vpwinner ← z.NEW[Person ← cand3];
vploser ← z.NEW[Person ← cand4];
```

```
NEW(preswinner); preswinner ↑ : = cand1;
NEW(presloser);   presloser ↑ : = cand2;
NEW(vpwinner);   vpwinner ↑ : = cand3;
NEW(vploser); vploser ↑ : = cand4;
```

```
preswinner ← presloser;
--preswinner and presloser both point to
--the same RECORD (initialized to MondaleData).
--No access path remains to the RECORD initialized
--with ReaganData.
```

```
preswinner : = presloser;
```

```
vpwinner ↑ ← vploser ↑ ;
--vp winner and vploser point to distinct
--RECORDS, each initialized to FerraroData.
```

```
vpwinner ↑ : = .vploser ↑ ;
```

```
FOR p: LONG POINTER TO Node ←
rootNode, p.next UNTIL p.next = NIL DO
  IF p.voter.contribution > 100
    THEN AskForMoney[p.voter.name]
ENDLOOP;
```

```
p : = rootNode;
WHILE p < > NIL DO
  BEGIN
    IF p ↑ .voter.contribution > 100
      THEN AskForMoney[p ↑ .voter.name];
    p : = p.next
  END;
```

```
--When applied to a pointer, the operation
--of selection implies dereferencing. In Mesa,
--this type of dereferencing is done
--automatically. Thus, it is not necessary to
--write p ↑ .voter.contribution or
--p ↑ .voter.name.
```

1.3 Mesa extensions of Pascal

1.3.1 Modules and interfaces

Mesa programs look quite similar to Pascal programs when viewed in the small. However, Mesa provides and enforces a modularization capability that is far more powerful than that of Pascal. In Mesa, you build large systems from a collection of smaller, separately compiled components called modules. The Mesa *binder* (the binder is similar to a linking loader in Pascal) enforces *strong type checking* among the modules that make up a system. In Pascal, you must make a choice when developing a large system. Either you construct a monolithic program to ensure type correctness, or you link separately complied program units without any guarantee that the type of variable X in one unit matches the type of variable X in another unit. In the latter case, type mismatches are discovered only at run-time.

Type checking across module boundaries in Mesa is only part of its modularization power. There are two categories of module in Mesa. *Definitions* (or interface) modules declare types, constants, and procedure headers of procedures that manipulate values of types declared in the module. An interface defines an abstraction by collecting all operations on a class of objects into a single module. An interface module contains no executable code; it only contains enough information to allow the compiler to type check other modules that use the declared symbols. The body of a procedure declared in an interface is not part of the interface. Interface modules compile into symbol tables.

The second category of module is the *Program* module. A program module acts as an *implementor* of an interface if it contains code that implements procedures declared in an interface module. A program module acts as a *client* of an interface if it calls procedures defined in that interface module.

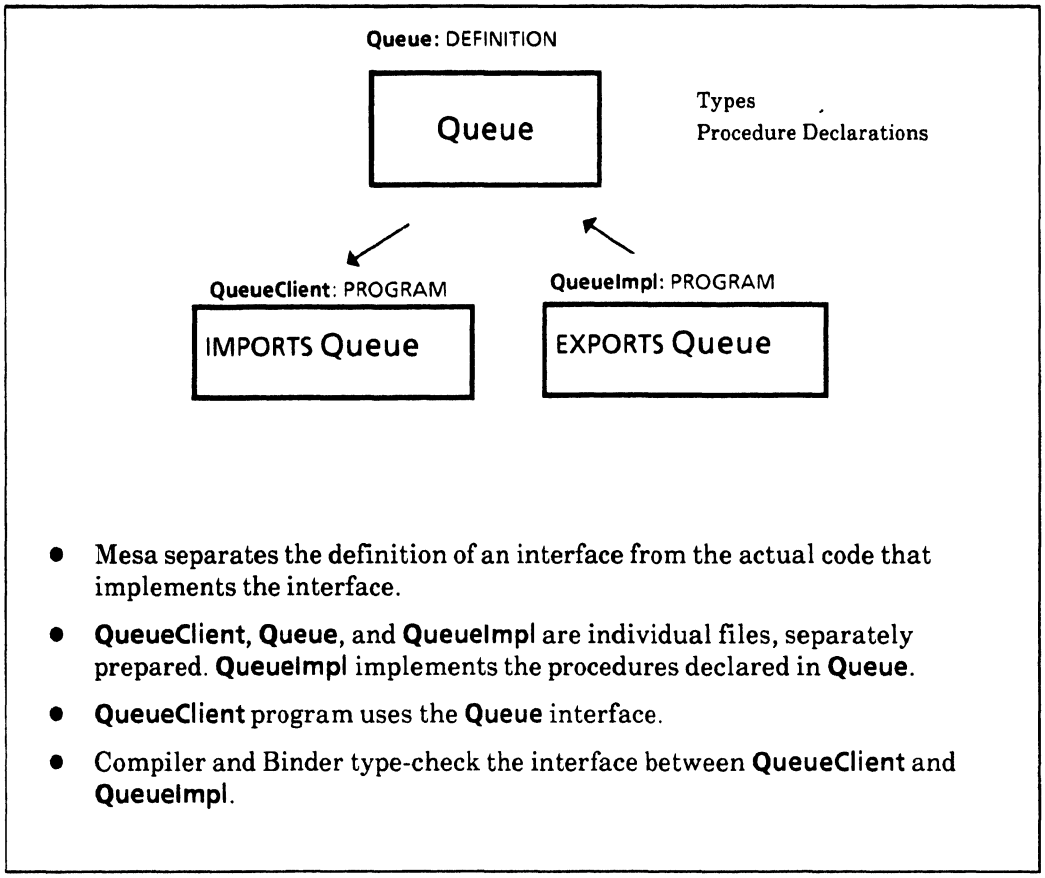
An interface is a contract between client and implementor: the interface specifies items that are available for clients to use, but doesn't say how they will be provided; the implementing module determines the details of the implementation.

There are several advantages of interfaces:

- Once an interface has been agreed upon, construction of the implementor and client can proceed independently. Thus interfaces and implementations are decoupled. This facilitates information hiding and permits changes to implementing modules without requiring a change to a client. Once an abstraction has been defined in a **DEFINITIONS** module (the interface) and implemented in one or more **PROGRAM** modules, an arbitrary (client) **PROGRAM** module can access the services advertised in the interface.

- Interfaces enforce consistency in the connections among modules. Operations upon a class of objects are collected into a single interface, not defined individually and in potentially incompatible ways.

- Nearly all of the work required for type-checking interfaces is done by the compiler.



Mesa modularity

1.3.2 Exceptions: signals and errors

Mesa provides *signals* to indicate exception conditions. Signals provide an orderly means for dealing with exceptions that is inexpensive if they occur infrequently. Examples of exceptions are invalid inputs, the inability of an abstractions to respond (e.g.,an allocator out of space), or any unusual or "impossible" event.

A Mesa SIGNAL can be thought of as the association of a procedure with an exceptional condition. "Raising" a signal when the exception occurs is similar to invoking the associated procedure except that the code to be executed is determined dynamically and is found in a "handler". The binding to a handler is determined by searching *catch phrases* (that contain handlers) in the call stack of the process in which the exception is raised; the dynamically innermost catch phrase that accepts the signal (by having a handler prepared to deal with the signal) is selected and executed. Often, parameters are passed when the signal is raised to help a handler determine what went wrong. Catch phrases are written in a distinctive syntax that clearly identifies them as the location of handlers containing code to respond to signals.

The cost of raising a signal is significantly higher than the cost of calling a procedure, but exceptions are events that should not happen very often. The system guarantees that all exceptions are handled at some level; those that the program fails to catch are accepted by the debugger. The debugger keeps intact the state of the program that raises a signal.

1.3.3 Processes, monitors, and condition variables

Mesa provides efficient mechanisms for concurrent execution of multiple processes within a single system. This allows programs that are inherently parallel in nature to be clearly expressed.

Example

```
GetInput: PROCEDURE[buffer: LONG POINTER TO Buffer]
    RETURNS [bytesRead: CARDINAL] =
BEGIN
p: PROCESS RETURNS [CARDINAL];
...

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

FORK makes it possible to start the execution of another procedure concurrently with the program that started it. FORK returns a process, which may either be detached to proceed independently, or saved for a future JOIN. A process type is declared similarly to a procedure type, except that only the type of the result is specified.

All processes execute in the same address space. Consequently, they are not protected from each other (certainly acceptable in a single-user system) but process creation and switching between processes is cheap (about the same as a procedure call).

Mesa provides facilities for synchronizing processes by means of entry to monitors and waiting on condition variables. A monitor has shared data in its global frame, and its own procedures for accessing it. To prevent two processes from executing the the same monitor at the same time, a *monitor lock* is used for mutual exclusion. Calling one of a monitor's ENTRY procedures automatically acquires the monitor lock (WAITing if necessary), and a return releases it. The monitor lock serves to guarantee the integrity of the global data, which is expressed as the monitor invariant, 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.

Example

```
StorageAllocator: MONITOR =
  BEGIN
  StorageAvailable: CONDITION;
  Block: TYPE = RECORD[...];
  ListPtr: TYPE = LONG POINTER TO ListElmt;
  ListElmt: TYPE = RECORD[block: Block, next: ListPtr];
  FreeList: ListPtr;

  Allocate: ENTRY PROCEDURE RETURNS [p: ListPtr] =
    BEGIN
    WHILE FreeList = NIL DO
      WAIT StorageAvailable
      ENDLOOP;
    p ← FreeList; FreeList ← p.next;
    END;

  Free: ENTRY PROCEDURE[p: ListPtr] =
    BEGIN
    p.next ← FreeList; FreeList ← p;
    NOTIFY StorageAvailable
    END;

  END.
```

It may happen that one process enters the monitor, finds the monitor data in a valid state, but cannot continue until some other process enters the monitor and alters the state (for example, a process may find that there is no storage available). 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 associated by agreement with the actual condition required. 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) and testing the condition again.

1.3.4 New data types

In Mesa, the predefined type LONG STRING is really "LONG POINTER TO *Stringbody*"; a *StringBody* contains a packed array of characters, a maxlength field giving the length of that array, and a length field indicating how many of the characters are currently significant. Each program contains the following predeclarations:

Example

```
LONG STRING: TYPE = LONG POINTER TO StringBody;
StringBody: TYPE = MACHINE DEPENDENT RECORD[
  length: CARDINAL,
  maxlength: --readonly-- CARDINAL,
  text: PACKED ARRAY[0..0) OF CHARACTER];

whatWasThat: LONG STRING = "Eh?";   --constant STRING
answer: LONG STRING ← [256];        --allocate a StringBody with maxlength 256
```

A sequence is an indexable collection of items, all of which have the same type. In this respect a sequence resembles an array; however, the length of the sequence is not part of its type. The (maximum) length of a sequence is specified when the object containing that sequence is created, and it subsequently cannot be changed. It is the responsibility of the programmer to keep track of the number of items in the sequence at any time. Sequences are declared as the last field in a record.

Example

```
lptscr: TYPE = LONG POINTER TO SequenceContainingRecord;
finger: lptscr ← NIL;
SequenceContainingRecord TYPE = RECORD[
  a: BOOLEAN,
  b: BOOLEAN,
  seq: SEQUENCE length:CARDINAL OF LONG INTEGER];

...
finger ← Heap.systemZone.NEW[SequenceContainingRecord[10]];
--SequenceContainingRECORD[10] is a TYPE specification describing a RECORD with a
--sequence part, seq, containing 10 LONG INTEGERS. The effect of the call is to allocate
--enough storage to hold two BOOLEANS and 10 LONG INTEGERS and return a long
--pointer to this storage.
```

Dynamic variables in Mesa are allocated in *zones*. Zones are not necessarily associated with fixed areas of storage; rather they are objects characterized by procedures for allocation and deallocation. There is a standard system zone, **systemZone**, but programs that allocate substantial numbers of similar dynamic variables can often improve performance by segregating each kind into its own zone. **NEW** is used to allocate a dynamic variable from a zone, and **FREE** to release it.

Mesa allows a default initial value to be associated with a type. Default values for arguments can simplify procedure applications; default initial values are useful to ensure that the corresponding storage is always well-formed, even before the variable has been used by the program.

1.3.5 Mesa extensions of Pascal constructs

This section mentions a number of areas where Mesa provides "convenience" extensions or conceptually small changes.

SELECT statements generalize Pascal's CASE construct by allowing several ways to specify how one statement is to be chosen for execution from an ordered list. The most common form is based on the relation between the value of a given expression and those of expressions associated with each selectable statement. The relation may be equality (the default), any relational operator appropriate to the types of the values involved, or containment in a subrange. A single selection may be prefixed by several selectors and an optional **ENDCASE** statement is selected only if none of the others are. *Discriminating* selection is used to branch on the type of a variant record value. **SELECT** expressions are analogous, but choose from an ordered list of expressions.

Examples

```
--control: [1..15];
SELECT control FROM
  1, IN [7..10] = > statement1;
  2, 5, >10   = > statement2;
  ENDCASE   = > statement3;
```

```
a[1] ← SELECT control FROM
    1, IN [7..10] = > 1.12;
    2, 5, >10   = > -4.856;
ENDCASE   = > 73.2;
    --A SELECT expression
```

```
Shape: TYPE = {point, line, circle};

Figure TYPE = RECORD[
  figureName: Name,
  specificFigure: SELECT fieldID: Shape FROM
    point = > [position: Coordinate]
    line = > [xCoef, yCoef, slope: REAL],
    circle = > [center: Coordinate,
                radius: REAL];
  ENDCASE];
```

Iteration is provided by loop statements in which several different kinds of control can be freely intermixed. A loop has a *control clause* and a *body*. The control clause may specify a logical condition for normal termination, possibly combined with a range or a sequence of assignments for a *controlled variable*. In addition to ordinary statements, the body may contain **EXIT** or **GOTO** statements to explicitly terminate its execution, and may be followed by a **REPEAT** clause that acts like a selection on the **GOTO** used to terminate the loop. (**GOTO** cannot be used to synthesize arbitrary control structures. It is much like a "local" exception.)

Examples

```
i ← 1;
UNTIL i > = 10                    --UNTIL i> = 10 is the loop control
  DO ... i ← i + 1 ; ...ENDLOOP;
Next-Statement;
```

The following example is equivalent to the one above.

```
i ← 1;
DO
IF i > = 10  THEN GOTO quit;          --first statement in the body
... i ← i + 1 ; ...
REPEAT --REPEAT doesn't mean repeat, it means "location of exits options".
  quit = > NULL;
ENDLOOP;
Next-Statement;
```

An example of linked list traversal:

```
NodeLink: TYPE = LONG POINTER TO Node;
node, headOfList: NodeLink;
Node: TYPE = RECORD[
  listValue: SomeTYPE,
  next: NodeLink];

FOR node ← headOfList, node.next UNTIL node = NIL
  DO ... ENDLOOP;
```

The *loop control* variable is **node**. Its *initial value*, **headOfList**, is assigned prior to the first iteration. Before each subsequent iteration the *next expression*, **node.next**, is reevaluated and assigned to the control variable. The user must either use a **GOTO** to terminate the loop or include a *condition test*. The condition test **UNTIL node = NIL** was used in the above example.

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

```
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;
```

In Pascal, procedure execution must proceed somehow to the end of the body before terminating; in Mesa, it can be terminated anywhere by executing a **RETURN** statement. If the procedure's type includes results, the **RETURN** statement may supply the values to be returned - otherwise they are taken from the result variables named in the type. Each procedure body is followed by an implicit return.

Examples

```
ReturnExample1: 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];                -- position version of option 1
    3 = > RETURN;                          -- a = b = c = 0
    ENDCASE = > b ← 4;
  c ← 9;
  END; -- implicit return; a = 0, b = 4, c = 9

ReturnExample2: PROCEDURE[g: INTEGER] RETURNS[INTEGER ← 3, INTEGER ← 4] =
  BEGIN
  SELECT g FROM
    0 = > RETURN [ , 2];                   -- RETURNS [3,2]
    1 = > RETURN [8, ];                    --RETURNS [8,4]
    2 = > RETURN [ , ];                    --RETURNS [3,4]
    3 = > RETURN [5];                      --RETURNS [5,4]
    4 = > RETURN [ ];                      --RETURNS [3,4]
    ENDCASE = >
  END;                                     --implicit return: [3,4]
```

Pascal procedures are not values that may be assigned to variables; Mesa procedures are.

Example

```
InverseTrigValue: REAL;
InverseTrigFunction: TYPE = PROCEDURE [x: REAL] RETURNS [REAL];

ArcSin: InverseTrigFunction = BEGIN --PROCEDURE body-- ...END; --PROCEDURE constant
ArcCos: InverseTrigFunction = BEGIN --PROCEDURE body-- ...END; --PROCEDURE constant
ArcTan: InverseTrigFunction = BEGIN --PROCEDURE body-- ...END; --PROCEDURE constant
InverseTrigFunctionVariable: InverseTrigFunction;        --PROCEDURE variable
...
InverseTrigFunctionVariable ← ArcSin;
InverseTrigValue ← InverseTrigFunctionVariable[3.1415/4];
```

1.3.6 Input and output in Mesa

The Mesa language definition omits many of the features commonly expected in programming languages, such as input/output and string manipulation operations. These facilities are available to Mesa programmers, but they are provided by interfaces written in the language itself. Standard interfaces are documented in the *Mesa Programmer's Manual.*

1.4 References

The definitive reference for the language is the *Mesa Language Manual*, version 11.0. The remaining chapters in the Mesa Course will guide your reading of the Mesa Language Manual and will discuss in detail all of the topics mentioned only briefly in this chapter.

1.5 Exercises

1. Convert the following Pascal program fragment to Mesa.

```
CONST
  maxlength = 1000;
TYPE
  index = 1..maxlength;
  rowType = ARRAY [index] OF integer;
VAR
  inrow : rowType;
  ix: index;

PROCEDURE shellsort (VAR row : rowType; length : index);
  VAR
      jump, m, n : index;
      temp : integer;
      alldone : boolean;

  BEGIN
      jump : = length;
      WHILE jump > 1 DO
          BEGIN
              jump : = jump DIV 2;
              REPEAT
                  alldone : = true;
                  FOR m : = 1 TO length - jump DO
                      BEGIN
                          n : = m + jump;
                          IF row[m] > row[n]
                              THEN
                                  BEGIN
                                      temp : = row[m];
                                      row[m] : = row[n];
                                      row[n] : = temp;
                                      alldone : = false
                                  END
                      END { FOR}
              UNTIL alldone
      END {while}
  END; {sort}
```

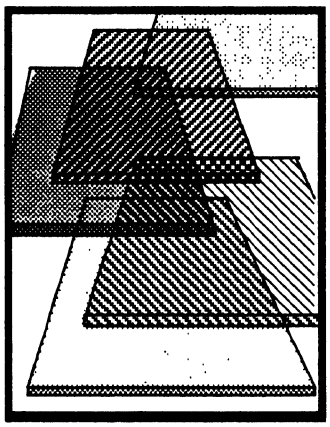
2. Convert the following Pascal program fragment to Mesa.

```
{straight list insertion}
TYPE
  ref = ↑word;
  word = RECORD
           key : integer;
           count : integer;
           next : ref
         END;
VAR
  root: ref;

PROCEDURE search (x: integer; VAR root: ref);
  VAR
    w : ref;
    b : boolean;
  BEGIN
    w : = root;
    b : = true;
    WHILE ( w < > nil) AND b DO
      IF w↑.key = x THEN b : = false ELSE w : = w↑.next;
    IF b THEN
      BEGIN {NEW ENTRY}
        w : = root;
        NEW(root);
        WITH root↑ DO
          BEGIN
            key : = x;
            count : = 1;
            next : = w
          END
      END
    ELSE
      w↑.count : = w↑.count + 1
  END; {search}
```

Notes:

2



Interfaces

As mentioned in the last chapter, the chief differences between Pascal and Mesa lie not in the syntax of the language, but rather in how modules interact to share information, and how individual modules are combined together into systems. Mesa's structured modularization allows modules to be created and tested individually, and then later integrated with complete type safety. Thus, Mesa effectively reduces the problems of programming in the large down to the problems of programming in the small. This chapter illustrates how Mesa's interfaces allow individual programs to share information; the next chapter discusses how interfaces are used in large-scale system building.

2.1 Preliminary readings

Skim the first five chapters in the *Mesa Language Manual* to get acquainted with the common Mesa constructs and syntax. You will need these chapters as a reference as you read this chapter and do the exercises.

Read Appendix B of the *Mesa Language Manual*, Programming Conventions, before you start to write your own programs.

2.2 Definition of terms

Client

A *client* is a program (as opposed to a person) that uses the services of another program or system.

Interface

An *interface* is a formal contract between pieces of a system that describes the services to be provided. A provider of these services is said to *implement* the interface; a consumer of them is called a *client* of the interface.

Interface module

An *interface* or **DEFINITIONS** *module* defines types, variables, constants, procedures, and signals, thus specifying the services to be provided by its implementation modules.

Implementation module

An *implementation* or **PROGRAM** *module* is a program that codes (*implements*) and makes available to clients (*exports*) items in an interface. One implementation module can export all or part of one or several interfaces, and an

interface can be implemented by several implementation modules jointly.

Load — *Loading* a module allocates memory space for its code and data, and links it to other modules that are already loaded, but does not start it.

Symbol — A *symbol* is any user-defined name in a program, such as a constant, type, variable, or procedure.

2.3 Discussion

There are two kinds of modules in Mesa: **DEFINITIONS** and **PROGRAM**. **DEFINITIONS** modules are also called *interface modules*, or just *interfaces* for short. You can think of an interface or **DEFINITIONS** module as a catalog containing a precise description of each item offered. The purpose of an interface is only to *define* procedures and variables that will be available to other programs; the interface does not contain the actual code for those procedures.

All executable code is contained in the second kind of module, called a **PROGRAM** module. A program module can act as a manufacturer of an interface (creating the items in the catalog), or as a customer (ordering items from the catalog). In Mesa, the "manufacturers" are called *implementors*, and the "customers" are called *clients*. Thus, program modules communicate via interfaces: a shared symbol is defined in an interface module, implemented by a program module, and used by other program modules. The interface is the link between the two program modules; there is no direct communication between client and implementation.

One advantage of this approach is information hiding; the client knows nothing of the implementation, and thus cannot take advantage of specific details of that implementation. Another important advantage is that the implementation is decoupled from the client; as long as the declaration in the interface remains the same, the implementation can be changed without affecting the client.

The rest of this chapter discusses the mechanics of linking together the three basic pieces of the interface mechanism, which are:

(1) an interface or **DEFINITIONS** module,
(2) an implementor of that interface, which is a **PROGRAM** module, and
(3) a client, which is also a **PROGRAM** module.

2.3.1 CompareImplA, which uses no interfaces

You can write Mesa code without using interfaces at all. CompareImplA.mesa is a simple example of a self-contained **PROGRAM** module. Take a look at the code:

```
CompareImplA: PROGRAM =
  BEGIN
  Compare: PROCEDURE [x,y: CARDINAL] RETURNS [same: BOOLEAN] =
    BEGIN
    IF x = y THEN RETURN[same←TRUE]
    ELSE RETURN[same←FALSE];
    END; --of procedure Compare
  END.
```

CompareImplA consists of one procedure, **Compare**, which takes two numbers as arguments, compares them, and returns a result of either TRUE (the numbers are the same) or FALSE (the numbers are not the same). However, there is no mainline code to call **Compare**, nor are there any I/O calls to get input or print results. Obviously, this program is of little use by itself. One way to make it useful is to "publish" it so that other programs can call our **Compare** procedure. This is called *exporting* the procedure.

2.3.2 Exporting

Exporting describes the relationship between an interface and its implementation. If you want to make a procedure available to the outside world, you define that procedure in an interface, implement it in a program module, and *export* the implementation to the interface. Client programs can then access the procedure directly from the interface. This process is called *exporting an interface*.

To use the earlier analogy, we want to publish a catalog from which clients can order a compare procedure, and we want to sign up as the manufacturer of the compare procedure advertised in the catalog. To do this, we have to write the interface and upgrade **CompareImplA** so that it exports **Compare**.

2.3.2.1 The interface

Here is the interface, which we have called **InterfaceB**:

```
InterfaceB: DEFINITIONS =      --keyword DEFINITIONS declares this to be an interface
  BEGIN
  Compare: PROCEDURE [x,y:CARDINAL] RETURNS[result:BOOLEAN];
  END.
```

This module is an interface; it defines procedures that are available to others. This particular interface contains only one definition, that of the procedure **Compare**. **InterfaceB** provides enough information about **Compare** so that the compiler can type-check client programs, but it does not contain the actual executable code for **Compare**. The actual code for **Compare** is in our implementation, which is a PROGRAM module.

2.3.2.2 The implementation

Here is **CompareImplB**, the implementation module:

```
DIRECTORY
  InterfaceB;
CompareImplB: PROGRAM EXPORTS InterfaceB =
  BEGIN
  Compare: PUBLIC PROCEDURE [x,y:CARDINAL] RETURNS[result:BOOLEAN] =
    BEGIN
    IF x = y THEN RETURN[result ← TRUE]
    ELSE RETURN[result ← FALSE] ;
    END; --of procedure Compare
  END.
```

This module is an upgraded version of **CompareImplA**; the code for the procedure is the same, but this time we are exporting the code to the interface. To export all or part of an interface, you need to do three things. You need to specify that you are referencing other

modules, you need to list the interfaces that you are exporting, and you need to list the specific procedures that you are exporting.

The DIRECTORY clause in **CompareImplB** accomplishes the first of these three; it tells the compiler which interfaces will be referenced during this compilation. If you want to use information from an interface, you must include that interface in your DIRECTORY clause. In this case, the compiler needs to reference **InterfaceB** to verify that the procedure declaration in the implementation matches the procedure declaration in the interface.

The EXPORTS clause accomplishes the second objective; it lists the interfaces that are being implemented, at least in part, by this module. An exporting module need not implement all the symbols in an interface; the implementation of an interface is often the cooperative effort of several modules. A PROGRAM module can also export more than one interface.

The third objective is achieved by declaring **Compare** to be a PUBLIC procedure. Symbols can be declared as being PUBLIC or PRIVATE. PUBLIC symbols can be exported to an interface, but PRIVATE symbols cannot. In PROGRAM modules, the default is PRIVATE: all symbols are assumed to be PRIVATE unless specifically declared PUBLIC. Thus, the word PUBLIC indicates that **Compare** is an implementation that is being exported to an interface. The compiler verifies that the declaration matches the declaration in the interface exactly, except for the word PUBLIC.

Figure 2.1 summarizes the communication between an interface and its implementation.

Interface

```
InterfaceName: DEFINITIONS =
  BEGIN
  ProcedureName: PROCEDURE...;
  END.
```

Implementor

```
DIRECTORY
  InterfaceName ;
InterfaceImpl: PROGRAM
 EXPORTS InterfaceName =
  BEGIN
  ProcedureName: PUBLIC PROCEDURE ... =
    BEGIN
    .....
    END ; -- of procedure
  END. -- of implementation module
```

Figure 2.1

2.3.3 Importing

Now that we have exported **Compare**, other programs can use it. Conveniently, we have a willing client, **CompareClient**, eagerly waiting on the sidelines to *import* our code.

Importing describes the relationship between a client program and an interface. A client that wishes to use a particular procedure only needs to know the definition of the procedure and the name of the interface from which to access it. It knows nothing about

the actual implementation. Thus, in our example, **CompareImplB** exported **Compare** to the interface **InterfaceB**, and now **CompareClient** can import **Compare** from **InterfaceB**. There is no direct communication between **CompareImplB** and **CompareClient**.

2.3.3.1 Importing a procedure

Here is the skeleton of **CompareClient**:

```
DIRECTORY
  InterfaceB USING [Compare] ;
CompareClient: PROGRAM IMPORTS InterfaceB =
  BEGIN
  .....
  f ← InterfaceB.Compare[a, b] ;
  .....
  END;
```

There are three steps to importing a procedure, which correspond to the three steps of exporting a procedure. First, you must list the interface in the **DIRECTORY** statement, just as in the exporting example. This tells the compiler that your module references **InterfaceB**. In this example, the **DIRECTORY** clause is further restricted by a **USING** clause, which lists the specific symbols that you will be using from that interface. Thus, **CompareClient** can use **Compare** from **InterfaceB**, but cannot use any other symbols from that interface. You do not have to have a **USING** clause, but it is a very good idea.

Second, you need to list **InterfaceB** in the **IMPORTS** list; *this specifies the interfaces for which implementations must be provided at run-time.*

Finally, you need to indicate that the procedure is imported by referring to it as **InterfaceB.Compare**, and not just **Compare**. You must always fully qualify the name of an imported symbol so that the compiler will know that it is coming from another interface.

2.3.3.2 Template for importing a procedure

Figure 2.2 diagrams the communication between an interface and a client that **IMPORTS** a procedure.

Interface

```
InterfaceName: DEFINITIONS =
  BEGIN
  ProcedureName: PROCEDURE...;
  END.
```

Client

```
DIRECTORY
  InterfaceName USING [ProcedureName];
ClientName: PROGRAM
  IMPORTS InterfaceName =
BEGIN
  ... InterfaceName.ProcedureName[...] ;...
  END.
```

Figure 2.2

2.3.3.3 Importing a constant

In the last section, we discussed how to import a procedure from an interface.However, not all information in an interface requires an implementation. Some of the symbols in an interface, such as variables, types, and constants, are *compile-time* symbols. Such symbols are available directly from the interface; no implementation is necessary. *Run-time* symbols, on the other hand, are symbols (such as procedures) for which code must be supplied at run-time. If you use only compile-time symbols from an interface, and not run-time symbols, you do not need to import the interface. For example, here is an interface:

```
IncrementDefs: DEFINITIONS =
  BEGIN
    inputTooBig: CARDINAL = LAST[CARDINAL]     --LAST returns largest value
  END.
```

and here is the module **IncrementImpl**, which imports **inputTooBig** from **IncrementDefs**.

```
DIRECTORY
  IncrementDefs USING [inputTooBig] ;        -- note interface and constant name
  IncrementImpl: PROGRAM =
    BEGIN
    Increment: PROCEDURE [x: CARDINAL] RETURNS [y:CARDINAL, error:BOOLEAN] =
      BEGIN
      IF x < IncrementDefs.inputTooBig THEN          -- note fully-qualified name
        RETURN [y ← x + 1, error ← FALSE]
      ELSE RETURN[y ← x, error ← TRUE] ;
      END ;
    END.
```

Thus, importing compile-time information is just like importing run-time information, except that you do not need to include the interface in the IMPORTS list. The IMPORTS list includes only those interfaces for which run-time implementations are needed.

2.3.3.4 Template for importing a constant

Figure 2.3 diagrams the communication between an interface and a client that is importing a constant from that interface.

Interface

```
InterfaceName: DEFINITIONS =
  BEGIN
  ConstantName: CARDINAL = ...;
  END.
```

Client using a constant

```
DIRECTORY
  InterfaceName USING [ConstantName];
InterfaceImpl: PROGRAM =
  BEGIN
  ...InterfaceName.ConstantName...;
  END.
```

Figure 2.3

2.3.4 Compiling and running your programs

As discussed above, a module's DIRECTORY clause lists all the interfaces referenced by that module. When you compile a module, the compiler needs to be able to read all the interfaces listed in the DIRECTORY clause so that it can type-check your program. This means that if you list an interface in your DIRECTORY clause, you must have the compiled version of that interface on your local disk when you compile your program, or you will get a compilation error. Thus, an interface must always be compiled *before* program modules that reference that interface.

Another important thing to remember is that when you recompile an interface, you will have to recompile all of its clients and implementors as well. The reason for this is that all Mesa object modules (.bcd files) contain a time stamp as part of their identification. When clients and implementors of an interface are compiled, the time stamp of the interface is noted and retained in both the client and implementation object code file identification. When you try to combine the client and the implementation into a larger system, the time stamps are checked against one another. If the client and the implementation do not reference the same version of the interface, a version mismatch will occur, which prevents the system from running.

Once you have compiled all the modules that make up a system, you can run the system. In the next chapter, you will learn how to use the *binder* to help you group your modules together, but for now you will have to load them all manually from CommandCentral. (All modules listed on the **Run** line of CommandCentral will be loaded.) You need to load all the program modules (your client, plus the implementations for any procedures that you have imported), but not the interfaces (since they don't contain executable code.) Implementation modules must be loaded before client modules, so that the implementation is ready when the client needs it.

Thus, to execute the Compare system, you would have to set up Command Central like this, and invoke **Go!**. You can run Compare now, if you like. (Note: **CompareClient** references some interfaces that you may not have on your local disk, so we have provided a compiled version of this module. Normally you would have to compile **CompareClient**.)

Compile: InterfaceB CompareImplB
Bind:
Run: CompareImplB CompareClient

2.3.5 Importing and exporting

In the previous example, each program module was either a client or an implementor. Generally speaking, however, a PROGRAM module can be a client, an implementor, or both. Most commonly, a given PROGRAM module is both client and implementor. The module can import and export the same interface, or it can export one or more interfaces and import another (or several others.) The terms *client* and *implementor* refer more to the function of a module than to the module itself; there is nothing to prevent a client module from also being an implementor, or vice versa.

Figure 2.4 is a diagram of the communication between an interface and another module, which is both an implementor and a client of the interface. This diagram is merely a composite of the client/interface and the implementor/interface diagrams.

Interface

```
InterfaceName: DEFINITIONS =
  BEGIN
  ConstantName: CARDINAL = ...;
  ExportedProcedureName: PROCEDURE...;
  ImportedProcedureName: PROCEDURE...;
  END.
```

Implementor and Client

```
DIRECTORY
  InterfaceName USING [ConstantName, ImportedProcedureName]
InterfaceImpl: PROGRAM
  IMPORTS InterfaceName
  EXPORTS InterfaceName =
  BEGIN
  ExportedProcedureName: PUBLIC PROCEDURE... = BEGIN...END;
  ...InterfaceName.ConstantName...;
  InterfaceName.ImportedProcedureName[] ;
  END.
```

Figure 2.4

2.3.6 System interfaces

System interfaces are general purpose interfaces that define comprehensive facilities for building everything from tools to whole systems. System interfaces serve as the entry point to an extensive library of procedures, variables, and data types, that saves you from reinventing and reimplementing utilities. Examples of system interface are **String**, which performs common string operations, and **Exec**, which handles communication with the Executive window.

System interfaces are nice because they provide so many useful utilities, but they have the attendant disadvantage that you must learn what interfaces are available, and what routines they implement. System interfaces that are part of Pilot (the operating system) are documented in the *Pilot Programmer's Manual*; interfaces that are part of the tools environment are documented in the *Mesa Programmer's Manual*.

You use symbols from a system interface just like private interfaces; you need to include the interface in the **DIRECTORY** clause and in the **IMPORTS** list, and refer to the symbol as **InterfaceName.Symbol**. In fact, system interfaces are just like all other interfaces except for one thing: the compiled versions of implementations of system interfaces are included in the XDE system bootfile. Thus, since the implementations are provided in the bootfile, you do not have to explicitly load implementation modules for system interfaces.

Recall from section 2.3.4 that when you use symbols from any interface, system or private, you must have the compiled version of the interface (not the implementation) on your local disk. If, for example, you want to use some procedures from the **Heap** interface (a system interface), you must make sure that Heap.bcd is on your local disk before you compile your program. Compiled versions of system interfaces are stored on a special directory, called the *release directory*; when you need to use a system interface, you will have to ask

someone where the release directory is and retrieve the appropriate object file for that interface from that directory.

Thus, to summarize: if you want to use procedures defined in the system interface **String**, you must import that interface and you must have the file String.bcd on your local disk when you compile your program (which is thus a *client* of the **String** interface), but you do not have to explicitly run the file that implements those procedures. In fact, you will not normally even know the name of the implementation file; remember, an interface is the link between programs, and the client need know nothing about the implementation.

2.3.6.1 An example of using system interfaces

To see an example, take another look at CompareClient.mesa, which uses procedures from several system interfaces. Here is the beginning of that program:

```
DIRECTORY
  FormSW USING [
    AllocateItemDescriptor, ClientItemsProcType, CommandItem, line0, line1,
    NumberItem, ProcType],
  Heap USING [systemZone],
  InterfaceB USING [Compare],
  Put USING [Line],
  Tool USING [Create, MakeFileSW, MakeFormSW, MakeMsgSW, MakeSWsProc,
    UnusedLogName],
  ToolWindow USING [TransitionProcType],
  Window USING [Handle];

CompareClient: PROGRAM IMPORTS FormSW, Heap, Put, Tool, InterfaceB =
...
```

CompareClient uses procedures from seven interfaces: six system interfaces and one private interface (**InterfaceB**). As you can see, the **USING** clause is a good way to document the exact symbols that this progam uses. Also notice that two of the interfaces are in the **DIRECTORY**, but not in the **IMPORTS** list. As discussed in section 2.3.3, this means that the symbols being used from that interface are compile-time values, and not run-time values.

2.4 Summary

Mesa's interfaces provide a formalized mechanism to allow individual modules to share types, constants, variables, and procedures. You can define your own interface, implement procedures declared in that interface, or use procedures implemented elsewhere. Interfaces thus encourage data abstraction and information hiding. As a quick review:

To *implement* a symbol defined in an interface you must:

- include the interface in your module's **DIRECTORY** clause;
- include the interface in your module's **EXPORTS** list;
- declare the symbol with the same name and type as appears in the interface;
- declare the symbol to be **PUBLIC**; and
- compile your module after the interface.

To *be a client* (use symbols defined in an interface), you must:

- include the interface name in the **DIRECTORY** clause;
- include the symbol in a **USING** clause
  (you do not have to have a **USING** clause, but it is a good programming habit);
- include the interface name in the **IMPORTS** list;
- use the symbol with its interface's name prefixed, as **Interface.Symbol**;
- compile the module after the interface has been compiled; and
- make sure the module that the implementation is available at run-time (loaded).

If you only use compile-time symbols, you do not need to **IMPORT** the interface.

Figure 2.5 on the next page summarizes the communication between an interface and its implementation and between an interface and its client. Implementations and clients are both **PROGRAM** modules, and a single module can function in both ways (although this is not shown in the figure.)

2.5 Questions

1) In what order must the following six modules be compiled? In what order must they be run?

   a) **Program1** is an implementation module that imports procedures from **Interface1** and **Interface2**. One of the procedures that it imports is implemented by **Program2**. **Program1** also exports a procedure to **Interface3**.

   b) **Interface1** is a definitions module.

   c) **Program2** is an implementation module that uses types from **Interface1** and exports a procedure to **Interface2**.

   d) **Interface2** is a definitions module that uses types from **Interface1**.

   e) **Program3** is a module that imports procedures from all three interfaces.

   f) **Interface3** is a definitions module

2.6 References

Chapter 7 of the *Mesa Language Manual* is essentially a denser statement of the information in this chapter and the next chapter.

Appendix A of the *Mesa Language Manual*, Pronouncing Mesa, tells you how to pronounce Mesa symbols.

Client

```
DIRECTORY
   InterfaceName USING [ProcedureName, ConstantName] ;
ClientName: PROGRAM
   IMPORTS InterfaceName =
   BEGIN...InterfaceName.ProcedureName[]; ... InterfaceName.ConstantName...END.
```

Notes:

1) This is a client module because it IMPORTS an interface.
2) The client can call procedures and use constants defined in the interface.
3) The interface must be listed in the DIRECTORY.
4) The procedures and constants must be in a USING clause.
5) The implementations of the procedures are bound at run-time, not at compile-time. The interface must be IMPORTed.
6) The constants are bound at compile-time. The interface need not be IMPORTed just to access them.

Interface

```
InterfaceName: DEFINITIONS =
   BEGIN
   ConstantName: CARDINAL = ...;
   ProcedureName: PROCEDURE ...;
   END.
```

Notes:

1) This is a interface module, as shown by the key word DEFINITIONS.
2) Interfaces can define constants that are available directly from the interface.
3) Interfaces can define procedures that are implemented by an implementation module.

Implementor

```
DIRECTORY
   InterfaceName ;
InterfaceImpl: PROGRAM
   EXPORTS InterfaceName =
   BEGIN
   ProcedureName: PUBLIC PROCEDURE ... = BEGIN ...END ;
   END.
```

Notes:

1) This is an implementation module because it EXPORTS an interface.
2) The *InterfaceName* must appear in the DIRECTORY.
3) The procedures being exported are declared as PUBLIC.
4) The EXPORTS list causes public procedures in this Implementation to be exported to the interface .
5) The module that implements interface *X* is conventionally called *X*Impl.
6) An implementation can also be a client provided the correct DIRECTORY ... USING clause is included. (see Figure 2.4.)

Figure 2.5

2.7 Exercises

Before beginning these exercises you should read Appendices A and B of this manual, which address Mesa syntax errors and debugger basics, respectively. Do the debugger exercises of Appendix B to start becoming familiar with the debugger.

2.7.1 Exercise in importing a procedure

Your assignment is to write a client program. We have provided an interface (**ReverseLettersDefs**) that defines a procedure, and an implementation module (**ReverseLettersImpl**) that supplies that procedure. The client module, which you should call ReverseLetters.mesa, will call the procedure **ReverseProc** from **ReverseLettersDefs**. **ReverseProc** in turn calls procedures that accept a character string from the user and output the string with the letters reversed.

Use the client template from Figure 2.5 to help you with this exercise. Once you have written your client program, compile the following modules (remember, an interface must be compiled before any modules that use it):

- ReverseLettersDefs.mesa -- the interface that defines **ReverseProc**
- ReverseLetters.mesa -- your client module
- ReverseLettersImpl.mesa -- the module that implements **ReverseProc**,.
- BasicIOImpl .mesa -- contains I/O procedures used by **ReverseLettersImpl**

Run the following modules

Run: BasicIOImpl ReverseLettersImpl ReverseLetters

BasicIOImpl implements procedures that are imported by **ReverseLettersImpl**, imported so it must be loaded before **ReverseLettersImpl**. When Tajo is ready, bring up the Tajo Executive window and type:

> ReverseLetters.~ hello -- *you type this*

The reversed letters are: olleh -- *the program returns this*

Experiment with reversing strings of letters and spaces.

2.7.2 Exercise in exporting a procedure

Now it's your turn to write an implementation module. You will write a procedure called **GetAverage** that computes the average of the integers passed to it. (You can do the average computation by any method, or do something else with the numbers, as long as you pass out an integer.) To keep the I/O simple, the average passed out of your procedure will be an integer value, and thus will be rounded up or down.

Your procedure will receive an array containing up to ten integers, and the actual number of integers to average. You will export your procedure **GetAverage** to the interface AverageDefs.mesa, which we provide. We also supply a client program to call your procedure and do the I/O.

After you have written your implementation module, compile the following modules:

- AverageClient.mesa -- this client program gets up to ten integers from the user, counts them, imports the interface **AverageDefs** to get your procedure, calls your procedure to compute the average of the numbers, and outputs the result.
- AverageDefs.mesa -- this is the interface that contains the definition of your procedure.
- AverageImpl.mesa (or whatever you called your implementation module).

Run the following files:

Run: AverageImpl AverageClient

Invoking **Run!** will put you into Tajo. Bring up the Executive and type:

>Average 2 4 -- *you type this*

The average is: 3 -- *the program returns this*

2.7.3 Exercise in importing and exporting using one interface

This exercise demonstrates importing and exporting using a single interface. First, you will import the interface **CombineDefs**. This imported interface provides the factorial routine **Fact**, which computes the factorial of a number for you. **CombineDefs** also contains some types and constants that you will need.

Your job is to write a procedure to compute a combinatorics problem, using the imported **Fact**. You will then export your procedure to the interface **CombineDefs** for a client to use. The client, which is provided for you, will create a tool window for you to enter data, and will use your code to compute a solution and display the result.

The first step is to write a procedure to calculate the following: Given a group of people of size **"baseSize"**, how many ways can you combine them into groups of size **"groupingSize"** ? The formula for this problem is

$$\frac{\text{baseSize!}}{\text{groupingSize! (baseSize - groupingSize)!}}$$

These variable names must be exact, and capitalization IS relevant. The name of your procedure will be **Combine**, and its type is **CombineDefs.CombineType**. You will find its definition in the interface **CombineDefs**. You will need to import **CombineType**, and the procedure **Fact** to perform the factorials from the interface **CombineDefs**. You will then export your procedure **Combine** to the interface **CombineDefs**.

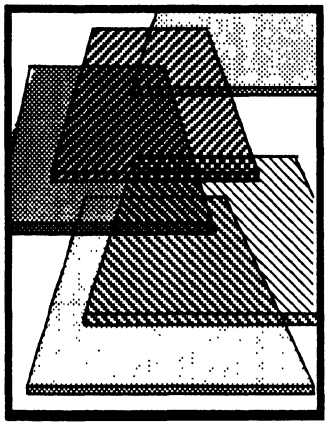
Using CommandCentral, compile the following 5 modules:

- CombineDefs.mesa -- the interface
- CombineImpl (or whatever you called it) -- the implementation module for **Combine**
- FactorialImpl.mesa -- supplies the factorial procedure for **Fact**
- CombinatoricsToolImpl.mesa -- supplies the user interface tool for the client
- CombineClient.mesa -- the client module

Run! the four implementation modules:

Run: CombineImpl FactorialImpl CombinatoricsToolImpl CombineClient

When you arrive in Tajo, you will see a tool window, which was produced by **CombinatoricsToolImpl**. Fill in the fields for **baseSize** and **groupingSize** and invoke **Combine!**. The answer will appear in the lower subwindow.



Binding

In the last chapter, we discussed how individual modules can use interfaces to share information. In this chapter, we will focus on how separately compiled modules are *bound* together into larger units.

3.1 Definition of terms

Configuration — A *configuration* is the bound code of one or more individual modules.

Configuration file — A *configuration file* is the file that contains the names of the modules that are to be bound together and describes how they are to be bound.

System interface — A *system interface* is an interface whose implementation is exported by the system bootfile.

3.2 Discussion

In the last chapter, you had to run several modules in a specific order to ensure that the implementation of an interface was available when a client program tried to reference it. This process is inconvenient, but manageable when there are few modules involved. When you are working on a large system, however, the job of keeping track of the necessary modules and their loading order becomes more difficult.

To help simplify things, the Mesa *binder* creates a logical structure called a *configuration* for the modules comprising a large system. This is analogous to the grouping of employees within a company. Groups of employees are organized into departments, with each department having certain duties. While the employees in a department do the actual work, the department itself can be thought of as doing the work, thus simplifying the world's view of things. Similarly, each configuration can be thought of as one logical entity that performs a certain task, although the task is actually performed by the modules within the configuration.

The binder processes a special file called a *configuration file*. This file contains a list of modules, which may be program modules or other configurations, and describes how they

are to be combined and initialized. The binder matches the import requests and export requests of the listed modules and creates an object module containing information about imported and exported items, object code for each module in the configuration, the names and versions of each module, and the interfaces referenced by those modules. This object module, the configuration, is also called a binary configuration description or "bcd" file.

There are several advantages to using a configuration instead of loading each module individually. One advantage is simplicity: after you have bound the modules together, you can type just the name of the configuration to run your program or system. Additionally, if other programmers want to use your system, they only need to obtain one module, the bound configuration, instead of finding and retrieving each individual module.

Another advantage of using the binder is version control. Every program module and definitions module has an associated time-stamp. This time-stamp can be thought of as an extension of the module's name; thus different versions of a module are different modules. For example, CompareImpl.bcd of Oct 14, 1984 1:15 p.m. is a different module from CompareImpl.bcd of Oct 15, 1984 10:12 a.m. When creating a configuration, the binder insures that all clients and implementors of an interface are referring to the *same version* of that interface; this effectively extends Mesa's strict type-checking across module boundaries.

3.2.1 A configuration file

The input to the binder is a *configuration file*, which contains a list of the modules to be bound, a list of imports and exports, and the order in which the modules are to be loaded. Here is Average.config, a configuration file for the program that you wrote in chapter 2:

```
Average: CONFIGURATION
IMPORTS Exec, String, Format, Heap
CONTROL AverageClient =
BEGIN
AverageImpl ;
AverageClient ;
END.
```

3.2.1.1 Reading a configuration file

Although **Average** looks much like a Mesa program, it is actually written in C/Mesa (configuration Mesa). There are five parts to a C/Mesa file:

(1) declaration (***Name*: CONFIGURATION**),
(2) **IMPORTS** list
(3) **EXPORTS** list
(4) **CONTROL** list
(5) **BEGIN-END** block

- The ***Name*** of the configuration file is the name that you will type to run your program after you have bound it.
- The **IMPORTS** list contains any interfaces that need to be imported from outside of the configuration; this is covered more fully in section 3.2.1.3.

- The EXPORTS list names all the interfaces for which this configuration exports an implementation. In this case, nothing is exported so there is no exports list. Exporting from a configuration is covered more fully in section 3.2.1.4 .
- The CONTROL list states which bound components are to be started and in which order. In most simple applications, only one component need be started explicitly. This is usually the component that contains mainline code. The other components are started implicitly when procedures in them are called.
- The BEGIN-END block itemizes the modules and configurations that are going to be bound together in the output configuration. This list corresponds to the list that you typed on the **Run:** line in the last chapter. In this case, the binder will use the information given in Average.config to bind together the files AverageClient.bcd and AverageImpl.bcd, and the resulting configuration will be stored in the file Average.bcd. The module names in the BEGIN-END block do not have to be listed in any particular order.

When you run the configuration **Average**, it will execute just as the individually loaded modules **AverageImpl** and **AverageClient** did in the chapter 2 exercise. If you want to try it, set up Command Central as follows and invoke **Go!:**

Compile:
Bind: Average
Run: Average

3.2.1.2 Importing into a configuration

The IMPORTS list of a configuration file is not simply a list of the imports of its components. It is a list of interfaces that need to be imported from *outside* the configuration. Interfaces that are imported by one module of the configuration and exported by another module in the same configuration are referred to as "self-contained" within the configuration, or "resolved." Such interfaces do not need to be imported by the configuration, but you must make sure that their implementation modules are listed in the configuration file.

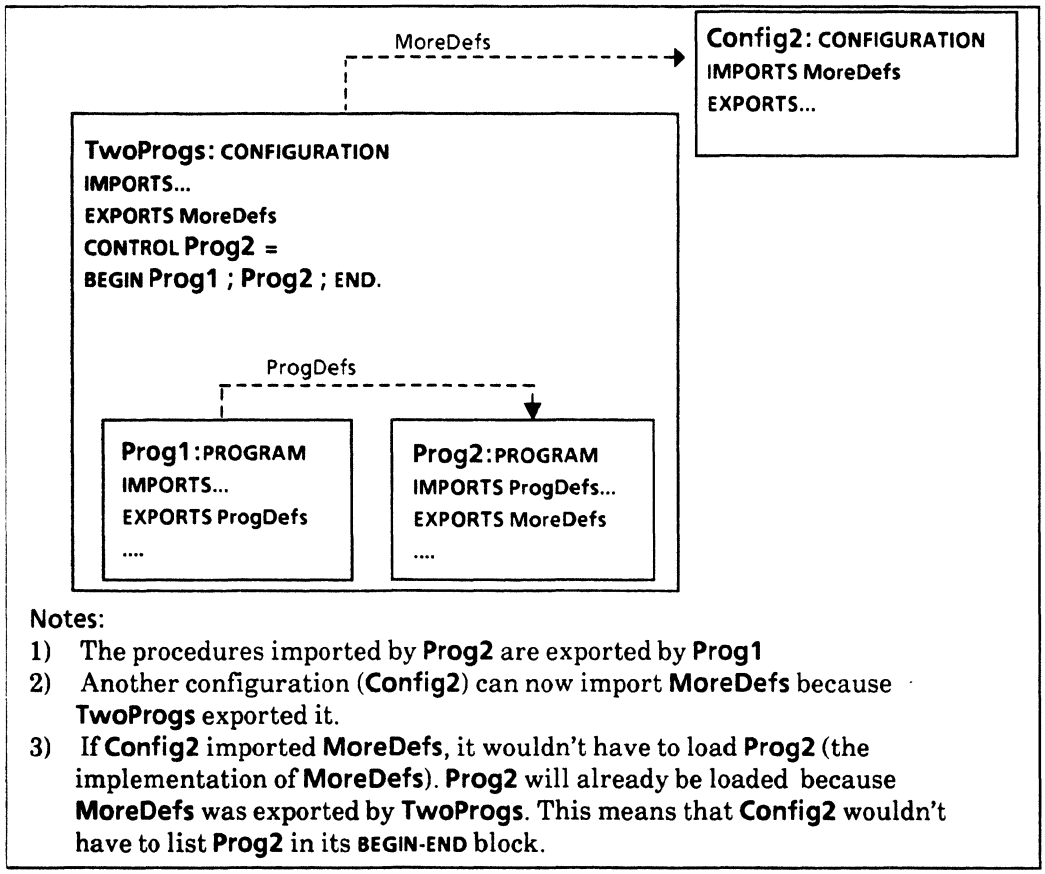
The module **AverageClient** imports **GetAverage** from the interface **AverageDefs**, and the module **AverageImpl** supplies **GetAverage**. Thus, all the necessary information is available; **GetAverage** need not be imported into the configuration. The implementations for **Exec**, **String**, **Format**, and **Heap**, however, are not supplied by either of the modules being bound together, and must thus be imported into the configuration. (Recall from the last chapter that implementations for system interfaces are part of the bootfile, and are thus already loaded.)

3.2.1.3 Exporting from a configuration

Like the IMPORTS list, the EXPORTS list is not just a list of items exported by the components of the configuration. Putting an interface in the EXPORTS list of a configuration makes its symbols available to the world outside the configuration, just as putting an interface in the EXPORTS list of a module makes its symbols available outside the module. You can think of the bound configuration as a large module, composed of other, smaller modules. You get to choose which symbols you will make available to the outside world, and which you will

keep local to your configuration. You might want to keep all of your symbols local to your configuration, in which case you wouldn't even have an **EXPORTS** list.

One of the side effects of exporting an interface from a configuration is that the interface's implementation will remain loaded. (It thus has the same status as a system interface.) This means that the next configuration that imports the interface won't have to load the implementation module by listing it in the configuration file. Figure 3.1 illustrates exporting an interface from a configuration.



Notes:

1) The procedures imported by **Prog2** are exported by **Prog1**
2) Another configuration (**Config2**) can now import **MoreDefs** because **TwoProgs** exported it.
3) If **Config2** imported **MoreDefs**, it wouldn't have to load **Prog2** (the implementation of **MoreDefs**). **Prog2** will already be loaded because **MoreDefs** was exported by **TwoProgs**. This means that **Config2** wouldn't have to list **Prog2** in its **BEGIN-END** block.

Figure 3.1 Exporting from a configuration

3.2.1.4 Template for a configuration file

Figure 3.2 is a general template for a configuration file.

Configuration
ConfigName: CONFIGURATION IMPORTS ***InterfaceA, InterfaceB,...*** EXPORTS ***InterfaceX, InterfaceY, InterfaceZ,...*** CONTROL ***Module1, ... =*** BEGIN ***Module1; Module2***; ... END.
Notes: 1) This is a configuration because of the key word **CONFIGURATION**. The name of the source file should be ConfigName.config. 2) The configuration contains **Module1**, **Module2**, etc. **ModuleK** can be a program or a configuration. Order of module names within the **BEGIN**...**END** block is not important. 3) The **CONTROL** statement specifies the module that is to receive control when the configuration is started. (Also list there any modules that require explicit starting, but this is rarely necessary.) 4) ***ConfigName*** will import the interfaces listed in the **IMPORTS** statement. These interfaces should be all those imported within any **ModuleM** and not exported by another **ModuleN**. 5) ***ConfigName*** will export the interfaces listed in the **EXPORTS** statement. These interfaces must be exported by some **ModuleJ**. (You never have to export anything from a configuration, unless you want to make it available to others.)

Figure 3.2 Template for a configuration file

3.2.2 Unbound procedures

In XDE, a configuration can be run even if some of the procedures are not available, as when the exporting module has not yet been loaded. If a missing procedure is not called, everything runs without incident. However, when a missing procedure is called, a software interrupt named **UnboundProcedure** is generated. The program will not be able to continue and control will transfer to the debugger. If this happens, you should make sure that *all* of the modules necessary to run your program are listed in your configuration file, and add them if they're not there. Such errors are generally easy to debug.

3.2.3 Naming conventions

The *file name* is the name of the file in which you store modules, as in XYZ.mesa. The *module name* is the name that appears before the word **PROGRAM**, **DEFINITIONS**, or **CONFIGURATION**. It is *highly* recommended that you keep the file name the same as the module name (and remember that capitalization is significant.)

The name of a configuration file should be different from the names of the modules that it binds together. The reason is this: if you compile a module called XYZ.mesa, you get an object file called XYZ.bcd. If you bind this module to other modules using a configuration

file called XYZ.config, you get a bound configuration called XYZ.bcd, which overwrites the old XYZ.bcd. Consequently, you lose your compiled implementation of XYZ.mesa. By convention, implementation modules should have the suffix Impl, as in XYZImpl.mesa, to avoid this problem. Figure 3.3 illustrates this problem and its solution.

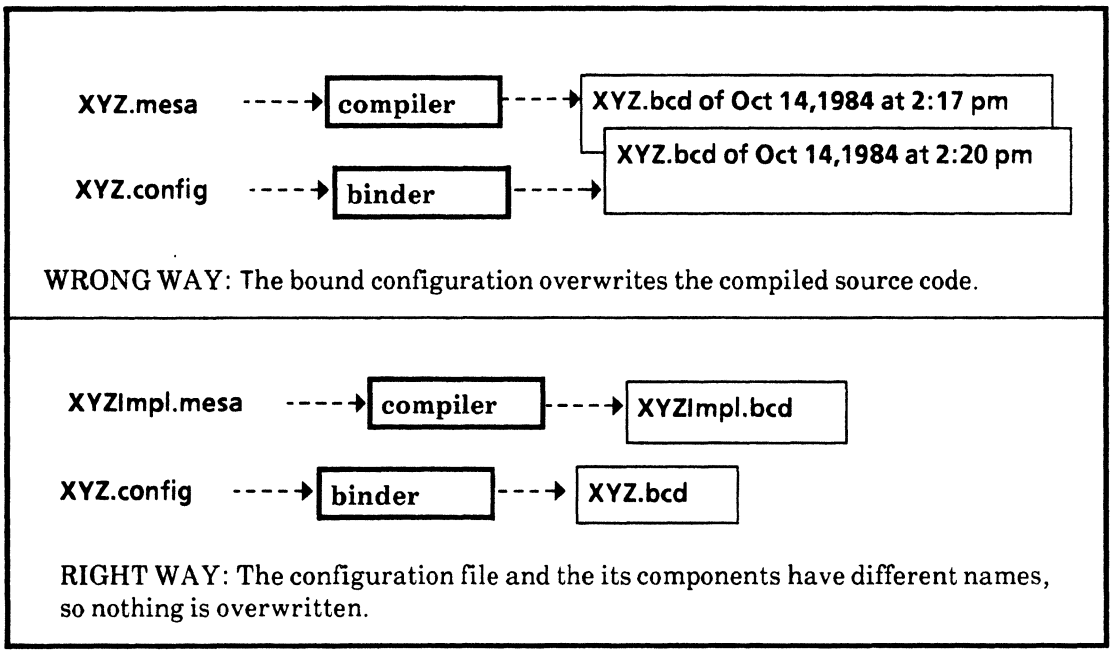


Figure 3.3 Naming conventions

3.2.4 System interfaces

As discussed in the last chapter, system interfaces are interfaces whose implementations are included in the bootfile. Thus, when you import a system interface, you do not have to include its implementation in your config file. The implementation is already bound into the bootfile, and will be available when you run your program. You do have to import the interface, but you do not have to include its implementation in your configuration, and you do need to have the copmiled version of the interface on your local disk.

3.3 Summary

This chapter discussed using the binder to produce bound configurations from a list of object modules. From the information in the "config" file and in each "bcd" file being bound, the binder can:

(1) resolve requests from modules for imported items
(2) combine a group of object modules into one larger object module
(3) control which interfaces are to be exported.
(4) determine which module is to be started first.
(5) maintain version control

Figure 3.4 gives a summary of the source file used by the binder, and its relationship to the modules that it binds together. This diagram also includes the use of system interfaces in program modules and in the configuration file.

Implementation Module

```
--this text stored in a file called ProgramNameImpl.mesa
DIRECTORY
 InterfaceName ;
ProgramNameImpl: PROGRAM
 EXPORTS InterfaceName =
 BEGIN
  ProcedureName: PROCEDURE ... = BEGIN ... END.
 END
```

Client Module

```
--this text stored in a file called ClientName.mesa
DIRECTORY
 InterfaceName USING [ProcedureName] ,
  SystemInterfaceName USING [SystemProcedure];
ClientName: PROGRAM
 IMPORTS InterfaceName, SystemInterfaceName =
 BEGIN ...
  InterfaceName.ProcedureName[] ;
  SystemInterfaceName.SystemProcedure[] ...
 END
```

Notes:
1) System interfaces are imported just like any other interface.
2) The module name should be the same as the program name, but not the same as any of the procedure names.

Configuration File

```
--this text stored in a file called ProgramName.config
ProgramName:CONFIGURATION
IMPORTS SystemInterfaceName
CONTROL ClientName =
BEGIN
 ProgramNameImpl ;
 ClientName ;
END.
```

Notes:
1) The name of the configuration file is not the same as the name of any of the modules that it binds together.
2) Implementation modules for the system interfaces are not listed.
3) There are no imports other than system interfaces because all of the imported interfaces are implemented by modules within the configuration.
4) Control goes to the module that has the mainline code, generally the client module.

Figure 3.4 Configuration file and Naming Conventions

3.4 References

Chapter 7 of the *Mesa Language Manual*, Modules, Programs, and Configurations, discusses configuration files and C/Mesa.

Chapter 17 of the *Xerox Development Environment User's Guide* discusses the binder and how to use it. This chapter also describes the binder's switches and error messages.

The *Mesa Programmer's Manual* and the *Pilot Programmer's Manual* give the details of the various system interfaces.

3.5 Exercises

3.5.1 Writing a configuration file and binding

For your first exercise, we have supplied a client program and two interfaces. Your job is to write a configuration file to bind the client with the implementations of the interfaces.

You will need the following files:

- ReverseWordsImpl.mesa -- the client program. It takes a string of input words (separated by spaces) from the user and reverses the order of the words.
- PrivateStorage.mesa -- an interface defining storage allocation procedures
- BasicIODefs.mesa -- another interface
- BasicIOImpl.mesa -- the implementation for some of the procedures defined in the interfaces **BasicIODefs** and **PrivateStorage**.

The scenario looks like this: **ReverseWordsImpl** gets the definitions of the procedures it needs from the interfaces **PrivateStorage** and **BasicIODefs**. These interfaces in turn get the actual code for the procedures from the implementation module **BasicIOImpl**. Therefore, you need to write a configuration file that binds together the client program and the implementation module. The name of your configuration file should be Reverser.config. You will then run the entire program under the name "**Reverser**".

Remember, if you are binding two modules together and one of them exports the symbols that the other imports, you don't need to list the interface in the **IMPORTS** or **EXPORTS** list of the configuration file. You only need to list interfaces that are **IMPORT**ed from outside the configuration file (such as system interfaces).

3.5.2 Writing an interface

We're going to re-visit the combinatorics exercise. This time, instead of using **CombineDefs** to export **Combine**, you will write your own interface to define this procedure. Modify your implementation of **Combine** so that it exports the interface **MoreCombineDefs**, and write this interface so that it defines **Combine**.

You still need to import **CombineDefs** to use **Fact** and **CombineType**. However, you should now export **Combine** to **MoreCombineDefs**.

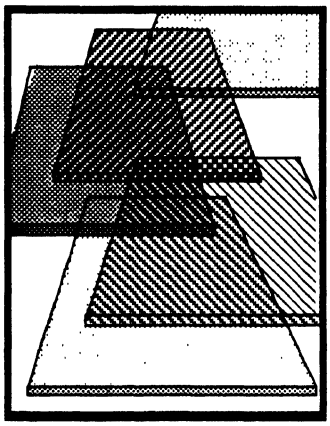
You must also modify the client module to import **Combine** from **MoreCombineDefs**.

Compile the following 3 modules:

- your interface (**MoreCombineDefs**)
- the modified client module (**CombineClient**)
- your modified implementation module (**CombineImpl**)

Write a configuration file, bind the necessary modules together, and run your configuration. Remember, you need all the same implementation modules that you needed last time you ran this program.

Notes:



Pointers

This chapter is an introduction to using pointers in Mesa. It covers what pointers are, how to perform common operations such as initialization and assignment on them, and how to pass them as procedure parameters. The next chapter, Dynamic Allocation, discusses how to allocate storage for the data that pointers reference.

There are a number of graphs throughout this chapter. They depict the memory in a hypothetical machine by representing each location in memory as a box. The number above the box is the memory location. The number in the box is the value stored in the location. The name below the box is the symbol in the example that has the associated value stored in the memory location.

4.1 Definition of terms

Pointer — A *pointer* is a reference to the location of a value. Mesa has *pointer types*, for pointers to specific types of values, and *pointer variables*, which contain the addresses of values rather than the values themselves. In Figure 4.1 below, **c** is a variable of type **INTEGER** containing the value 5. The variable **b**, a **LONG POINTER**, contains the address of **c**, and therefore **b** is a pointer to **c** and is said to reference **c**.

@ — @ is the prefix "address of" operator. @x generates a reference to the expression x. In Figure 4.1, **b** contains the value **@c**, and so **b** is a pointer to **c**. Similarly, **a** contains **@b**, and so is a pointer to **b**.

Dereference — To *dereference* a pointer is to follow the pointer through one level of indirection toward the value it is referencing. Dereferencing a variable is the opposite of generating a reference to a variable. In other words, if **b** is a pointer to **c** then dereferencing **b** produces **c**. In Figure 4.1, dereferencing **a** once produces **b**, and dereferencing **a** twice produces **c**.

↑ — In Mesa, ↑ is the postfix dereferencing operator. ↑ is the inverse of @, and is found at the opposite end of the expression. In Figure 4.1, **a** is **@b**, while **a**↑ is **b**, and **a**↑↑ is the same as **b**↑, which is **c**.

Dangling pointer — A *dangling pointer* is a pointer to an invalid memory location. A dangling pointer is usually caused by deallocating storage while a

pointer to it remains. Dereferencing a dangling pointer leads to unpredictable results.

Address fault

An *address fault* occurs when an attempt is made to reference an illegal address. For example, suppose that pointer **b** were not initialized to point to **c**, but instead left to be whatever value was in that location when **b** was allocated. If the value in the location is not a legal address, then dereferencing **b** causes an address fault. If, on the other hand, the address is legal, then you will *not* get an address fault. Rather, your pointer will be referencing some arbitrary location in memory, and you will be working with invalid data.

Frame

A *frame* is a Mesa processor data structure allocated while a module or procedure is executing to contain the variables and internal data structures for that module or procedure. Program frames are called *global frames*, and procedure frames are called l*ocal frames*. Since Mesa supports recursion, there may be several frames for a particular program or procedure.

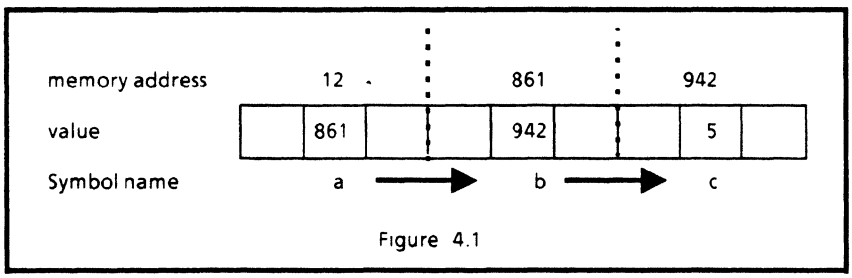


Figure 4.1

4.2 Discussion

Pointers are essential for good programming.

4.2.1 Declaring pointers

The Mesa architecture defines a uniform, paged virtual memory of 16-bit words. (A page is 256 words.) The entire virtual memory can be accessed by **LONG POINTER**s, which are two words long and can therefore address all 2^{32} locations.

Within this uniform virtual memory there is a distinguished region called the Main Data Space (**MDS**). Within the **MDS**, words may be addressed by **POINTER**s, which are one word long. The **MDS** is used internally to hold global and local frames. Therefore, all the pointers to storage that you allocate should be **LONG POINTER**s.

Pointers in Mesa are declared as references to types so that the Compiler can type-check their usage. The following example declares a pointer to an object of type **INTEGER**:

```
intPtr: LONG POINTER TO INTEGER;
```

4.2.2 Initializing pointers

Pointers allow indirect access to objects. In order for a pointer to be meaningful, the object it points to must exist. This means that storage has been allocated for the object, and has

been appropriately initialized. In the exercises in this chapter, the storage is allocated from the program's frame. Once an object is allocated and initialized, the @ operator is used to generate the pointer.

You can also allocate storage dynamically using the system's storage allocator; we will discuss this in the next chapter.

To initialize a pointer called **intPtr** to point to an **INTEGER** variable whose value is 5 you would write:

```
int: INTEGER ← 5;
intPtr: LONG POINTER TO INTEGER ← @int;
```

The first line allocates a space in the global frame and initializes it to 5. The second line initializes the pointer to the address of the storage location that contains the integer, as depicted in Figure 4.2 below.

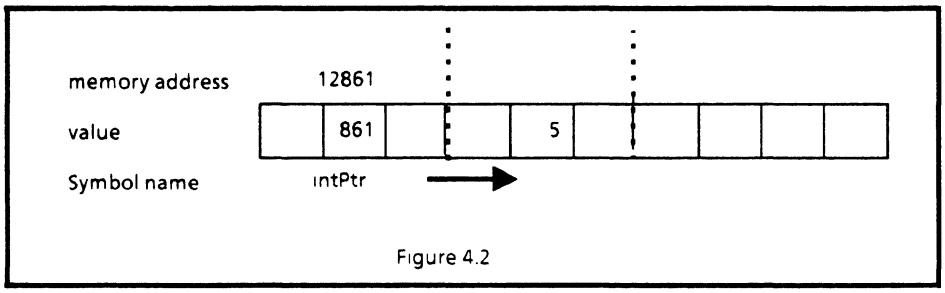


Figure 4.2

What if **intPtr** were initialized and **int** were not? As shown in Figure 4.3, the value for **int** would be meaningless, even though **int** is allocated. Pointing **intptr** to this location is valid, but not very useful.

```
int: INTEGER;
intPtr: LONG POINTER TO INTEGER ← @int;
```

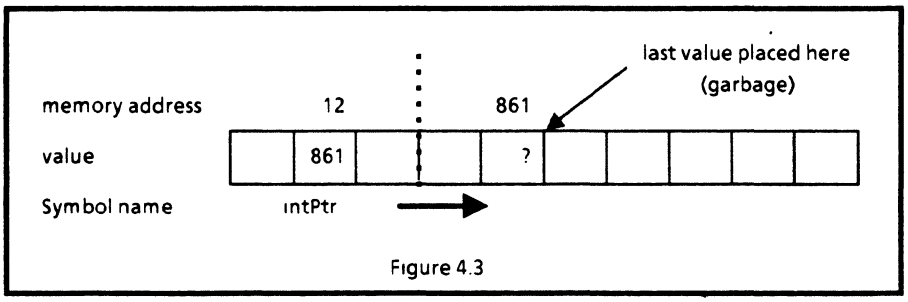


Figure 4.3

It is a good idea to avoid having pointers to uninitialized objects, lest you forget that the object is uninitialized and try to use the pointer. This would cause strange errors that are hard to debug. Instead, keep a pointer "uninitialized" until the object it will point to is initialized. Consider:

```
int: INTEGER;
intPtr: LONG POINTER TO INTEGER;
```

This recoding is one way of keeping your pointer uninitialized, but it suffers from the same problem as before. Now there are two uninitialized variables instead of just one, as illustrated in Figure 4.4.

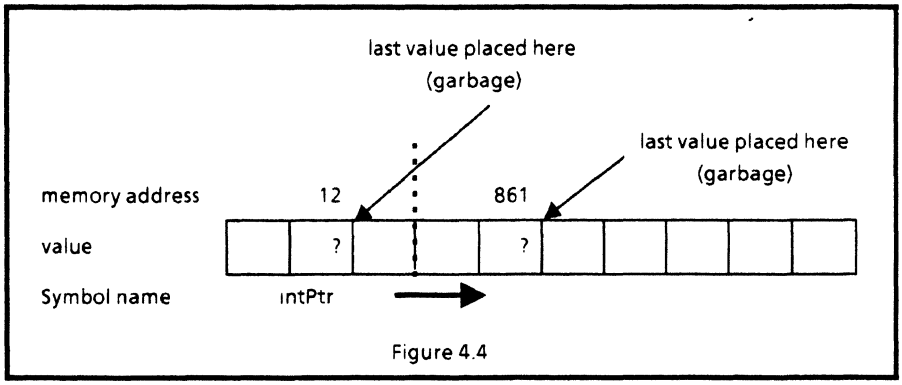


Figure 4.4

We have already discussed what might happen if you have a pointer to an uninitialized variable (such as **int**). If you try to dereference an uninitialized pointer, on the other hand, the value stored in the pointer's location would be interpreted as the address of a location. As shown in Figure 4.5 this pointer's value might point to a valid memory location in the address space. Dereferencing **intPtr** would therefore yield the garbage value 212 stored in memory location 942.

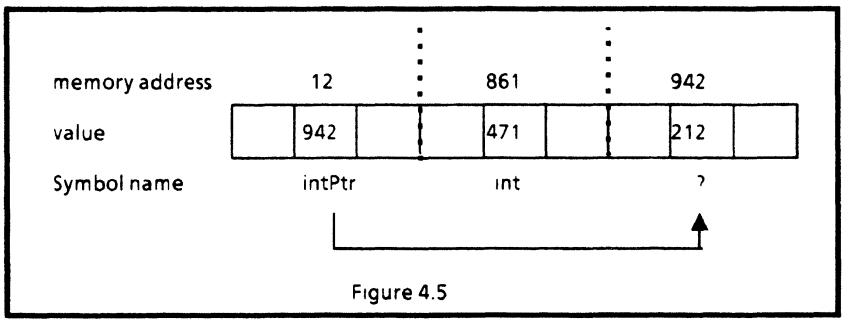


Figure 4.5

If, on the other hand, the value of **intPtr** pointed outside of the address space, to unavailable memory, then your program would address fault and the debugger would be called. In an environment that uses real memory addresses in code, this means that any address that points beyond the end of available memory would cause an address fault. However, the Pilot environment provides virtual memory. Addresses (that appear in code) are virtual and must be dynamically translated into real memory address at runtime.

During address translation, Pilot determines whether the page containing the reference is in real memory. If it is not, a page fault occurs and the page is swapped in from its backing file using available mapping information. An address fault occurs if the page to be swapped in is not mapped (has no associated backing store). Thus, in a virtual memory system, addresses that lie in the address space of a process can still cause address faults if they reference sections of the address space that are not mapped, as shown in figure 4.6.

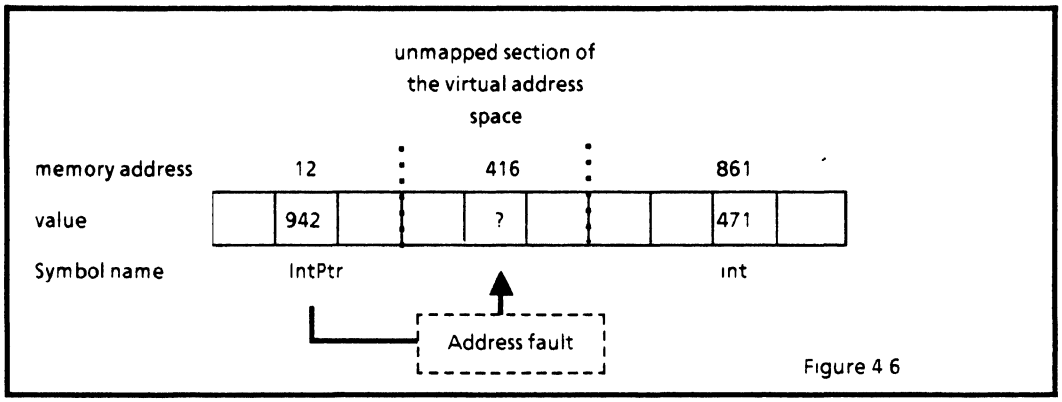


Figure 4 6

It is important to initialize all pointers, even those that have no referent. Mesa provides the special value NIL for this purpose. NIL signifies that a pointer does not point to anything valid and should not be dereferenced. Dereferencing a NIL pointer is undefined and will cause an address fault. When you are debugging, getting an immediate address fault is far better than having your program continue to execute with invalid data. In the latter case, your program may not malfunction until far from the scene of the crime.

```
int: INTEGER;
intPtr: LONG POINTER TO INTEGER ← NIL;
```

4.2.3 Assigning pointers

There are two common uses of pointers in assignment statements: assigning the address of a location to a pointer, as in the initialization of intPtr; and changing the contents of one pointers's referent to be a copy of another pointer's referent.

4.2.3.1 Assigning pointer values

In Mesa, pointers are type checked to the object they reference. This means that only pointers pointing to the same type of object can be assigned, as in this example:

```
int: INTEGER ← 5;
intPtr: LONG POINTER TO INTEGER ← @int;
anotherPtr: LONG POINTER TO INTEGER ← NIL;
anotherPtr ← intPtr;
```

The assignment of intPtr to anotherPtr is valid because they both point to an object of type INTEGER. After the assignment is complete, both intPtr and anotherPtr point to the same memory location. This has the same effect as if both pointers were individually assigned the address of int, like this:

```
int: INTEGER ← 5;
intPtr: LONG POINTER TO INTEGER ← @int;
anotherPtr: LONG POINTER TO INTEGER ← @int;
```

Figure 4.7 shows a before-and-after view of this assignment.

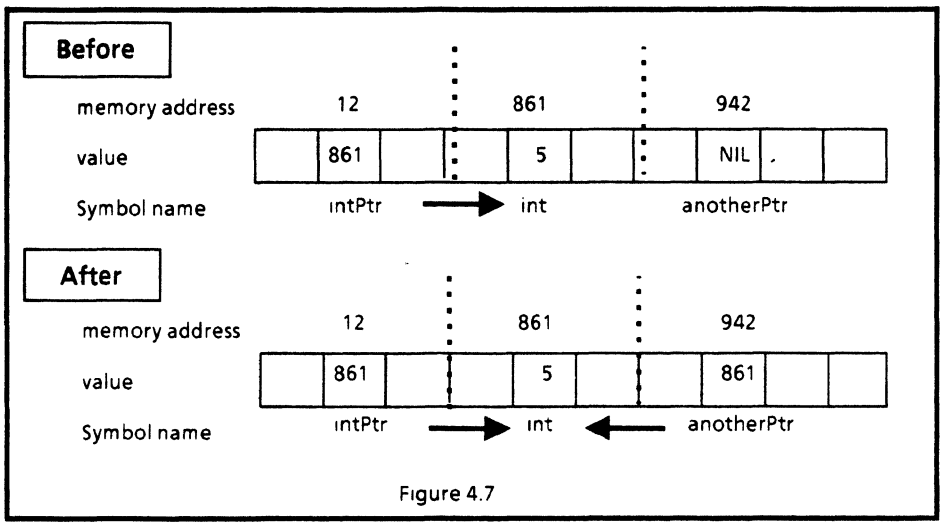


Figure 4.7

Now both **intPtr** and **anotherPtr** reference **int**. When **int**'s value changes, dereferencing either pointer will yield the changed value.

4.2.3.2 Assigning the contents of pointer references

Often, you do not want to share the value of an object, but you want to have two pointers that reference identical copies of one object. To do this, you dereference the pointers in the assignment statement:

```
int: INTEGER ← 5;
anotherInt: INTEGER ← 0;
intPtr: LONG POINTER TO INTEGER ← @int;
anotherPtr: LONG POINTER TO INTEGER ← @anotherInt;
anotherPtr ↑ ← intPtr ↑ ;
```

This assignment copies the value referenced by **intPtr** into the memory location referenced by **anotherPtr**. Changing the value in either of these two locations has no effect on the value pointed to by the other pointer. Figure 4.8 shows this situation.

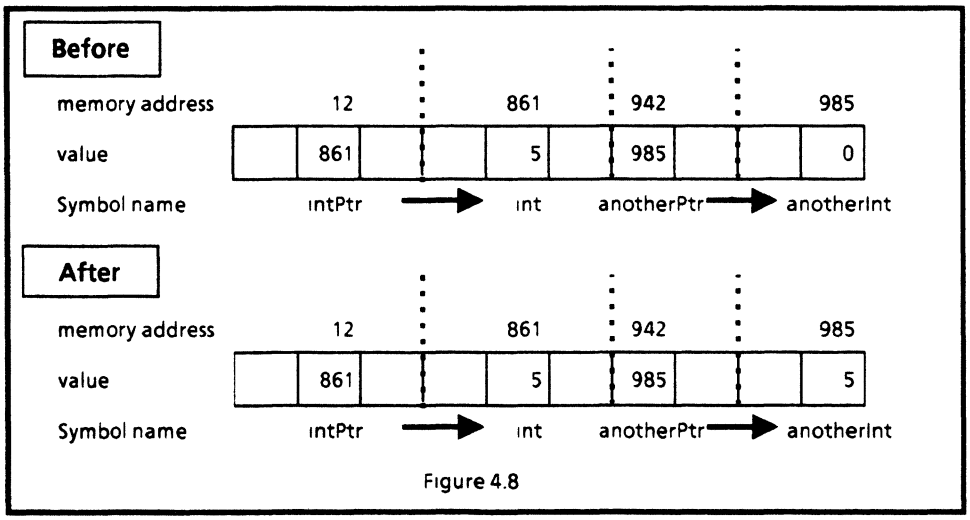


Figure 4.8

When you use pointers, be sure to think about the type of assignments you want your program to perform. If you accidentally share data between two or more pointers when you intend to copy the values, you will undoubtedly find some surprises when one pointer's referent is unexpectedly changed through another pointer. Conversely, copying data when you intend to share it will result in expected changes not taking effect.

4.2.4 Using pointers for parameter passing

There are two basic techniques of parameter passing: *call by reference* and *call by value*. In Mesa, all parameter passing is done as call by value. In other words, the variables passed as parameters to a procedure are not changed by what happens inside that procedure's body. For example, consider the procedure **DoNothing**:

```
DoNothing: PROCEDURE [a: INTEGER] =
  BEGIN a ← a + 1; END;
```

Assume that an INTEGER **int** has the value 5. When a program calls **DoNothing [int]**, the *value* of **int** is copied into **DoNothing**'s local variable **a**. When **DoNothing** changes the value of **a**, nothing happens to the value of **int**. Once **int**'s value has been copied into **a**, **int** is isolated from whatever goes on inside of **DoNothing**. Upon exit from **DoNothing**, **a** has the value 6 but **int** still has the value 5, as illustrated in Figure 4.9.

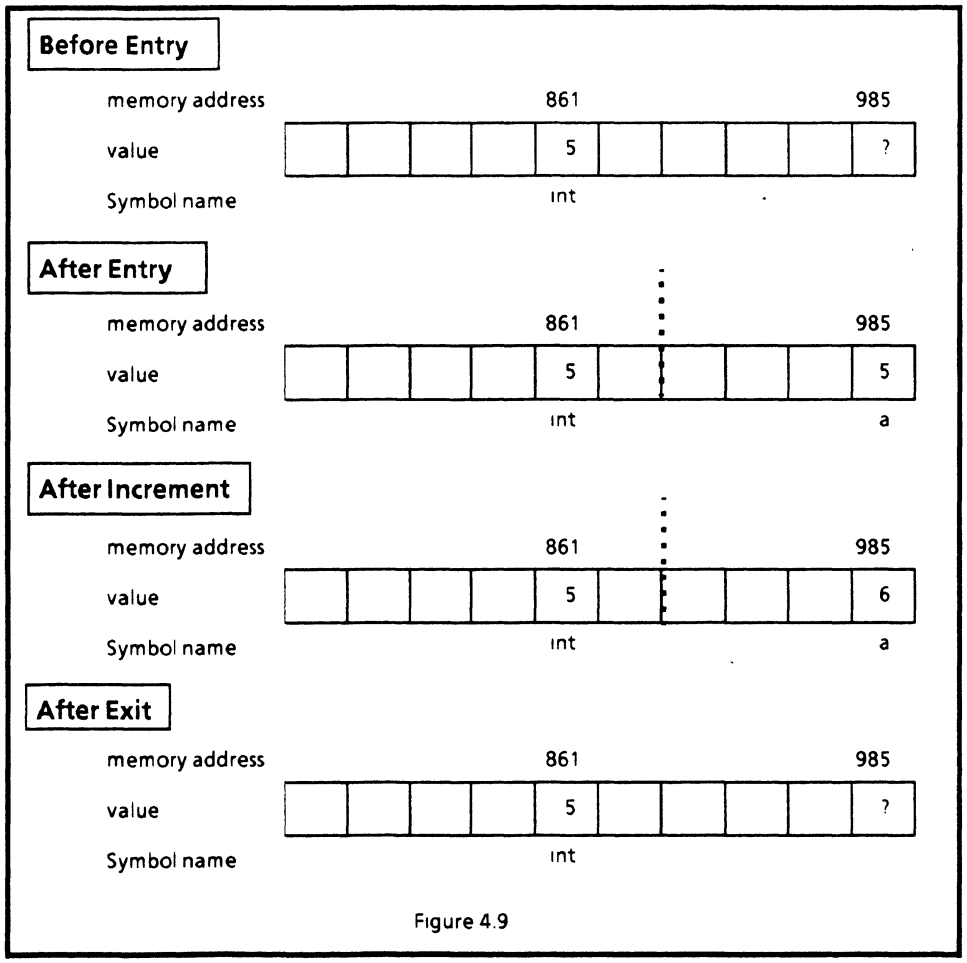


Figure 4.9

If Mesa did support call by reference and **DoNothing** was called so that its parameter, **a**, was a reference to the actual parameter, **int**, then **DoNothing** would have the desired effect of incrementing **int**. This manner of programming, where an argument to a procedure is changed as a side effect of the call, is considered bad form and discouraged in favor of having the procedure return the new value, as in:

```
DoSomething: PROCEDURE [a: INTEGER] RETURNS [INTEGER] =
  BEGIN RETURN [a + 1]; END;
```

Nevertheless, it is sometimes desirable for a procedure to modify one of its arguments. For example, a procedure may be called with a large array, several components of which need to be changed. If the array is so large that returning a copy of it would consume significant processor time and memory, then efficiency considerations may outweigh model programming, and the procedure might be designed to accomplish its end through side effects on its input.

When a procedure needs to have a side effect on one of its input variables, it takes as an argument not the variable itself but a pointer to that variable. After all, a pointer is a reference to where the value of the variable is stored. Given this reference (the address of the variable), a procedure can freely manipulate the contents of a variable by storing values into the location in memory where the variable's value resides. For example, a procedure **Increment** could look like this in Mesa:

```
Increment: PROCEDURE [a: LONG POINTER TO INTEGER] =
  BEGIN a↑ ← a↑ + 1; END;
```

To change the value of **int** by calling **Increment**, a program has to pass the procedure a pointer to **int**. When it makes the call **Increment[@int]**, the program makes the local variable **a** inside **Increment** point to **int**. Given such a call, **Increment** can change the value of the variable **int** by dereferencing the pointer **a**. Figure 4.10 illustrates the situation upon entry to the **Increment** procedure. The local variable **a** contains the address of the global variable **int**. When the assignment statement **a↑ ← a↑ + 1** is executed inside of **Increment**, the value of **int** is incremented. If **int** held the value 5 before the call **Increment[@int]**, then it will contain the value 6 immediately after the statement **a↑ ← a↑ + 1** is executed, as illustrated in Figure 4.10.

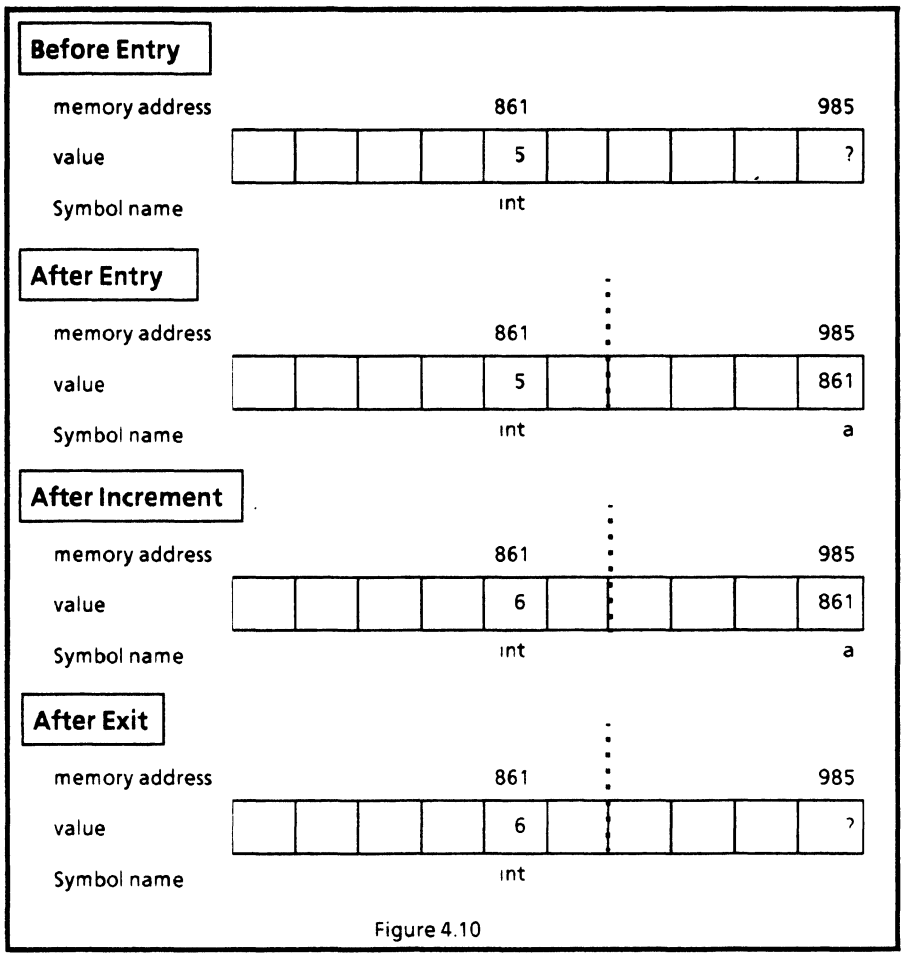


Figure 4.10

4.2.5 A common mistake: dangling pointers to local storage

When you asssign pointers to local values in procedures, you must not reference these values after exiting the procedure. Dereferencing a dangling pointer that used to point to a value allocated in a local procedure is undefined. The following example illustrates this.

`SimplePointer1.mesa` contains an instance of the **Increment** procedure discussed above. This program, when run, will work perfectly. Take a look at the code:

```
SimplePointer1: PROGRAM =
  BEGIN
  c: CARDINAL ← 0;
  worked: BOOLEAN ← FALSE;

  Increment: PROCEDURE [a: LONG POINTER TO CARDINAL] =
    BEGIN a↑ ← a↑ + 1; END; --Increment

  Unity: PROCEDURE RETURNS [b: CARDINAL] = BEGIN b ← 1; END; --Unity

  --Mainline Code
  c ← Unity[];
  Increment[@c];
  worked ← c = 2;
  END.
```

`SimplePointer2.mesa` tries to accomplish the same thing as **SimplePointer1**, but it takes a more devious approach. The code for **SimplePointer2** is slightly confusing, but looks like it will work when run. Unfortunately, the code is faulty. See if you can find the problem:

```
SimplePointer2: PROGRAM =
  BEGIN
  c: CARDINAL ← 0;
  worked: BOOLEAN ← FALSE;

  Increment: PROCEDURE [a: LONG POINTER TO CARDINAL] =
    BEGIN a↑ ← a↑ + 1; END; --Increment

  PointerToUnity: PROCEDURE RETURNS [b: LONG POINTER TO CARDINAL] =
    BEGIN d: CARDINAL ← 1; RETURN[@d]; END; --Unity

  --Mainline Code
  c ← PointerToUnity[]↑;
  Increment[@c];
  worked ← c = 2;
  END.
```

Look at the first assignment statement in the main body of **SimplePointer2**, the line: **c ← PointerToUnity[]↑;**. The intent is to dereference the pointer returned by the call to **PointerToUnity** in order to get the value 1. While **PointerToUnity** is executing, the situation is as depicted in the "Before Exit" part of Figure 4.11. The pointer **b** to be returned by **PointerToUnity** contains the address of the variable **d**, a variable local to **PointerToUnity**.

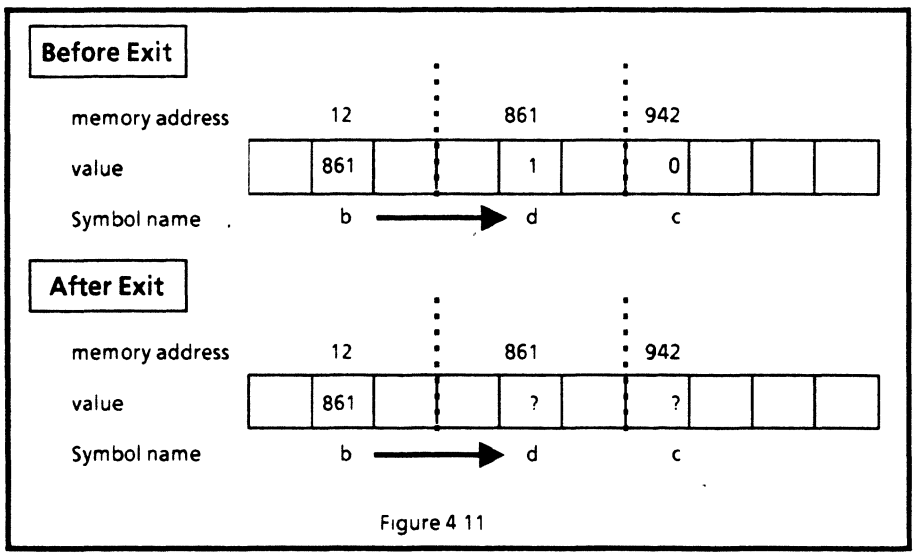


Figure 4 11

"After Exit" shows the situation after returning from **PointerToUnity**. The variable **c** should be assigned the value contained in the variable pointed to by **b**. But, now that **PointerToUnity** has been exited, the space used by **PointerToUnity** is considered by the system to be free space, ready to be overwritten as space is needed. Since **d** is local to **PointerToUnity**, it may already be overwritten now that **PointerToUnity** has been exited. The pointer returned by **PointerToUnity** points to where the value of **d** used to be. But **d** may be overwritten now, and so the pointer is worthless. When the program tries to assign

the value **@d↑** to **c**, it will be assigning a value that *might not* be the value that **d** had when **PointerToUnity** finished execution.

This procedure demonstrates the mistake of returning a dangling pointer to a local variable. When assigning pointers to values in local frames, be sure that the referents will still exist after the procedure has returned. One way to ensure this is to dynamically allocate space that outlives the local frame; this is the subject of the next chapter.

4.3 Summary

This chapter briefly discussed how pointers are used in Mesa programs. It presented a set of do's and don't's to keep in mind when programming with pointers, most notably:

- Do declare pointers as pointers to objects. This keeps you inside of the Mesa type checking system, which will go a long way in preventing pointer errors.
- Do initialize all variables including pointers. Having initialized variables will save you the trouble of worrying about whether or not a variable's value is valid. When you cannot initialize a pointer to an allocated and initialized piece of storage, signify this by initializing the pointer to **NIL**.
- Do be aware, when using pointers in assignment statements, whether you want the value shared between the two pointers (and therefore alterable by either pointer), or copied. To share the value between two pointers, assign the pointers (**ptr2 ← ptr1**); to copy the value, assign the dereferenced pointers (**ptr2↑ ← ptr1↑**).
- Do use pointers as arguments to procedures when you want the value of the caller's variable changed by the called procedure.
- Do **not** return pointers that point to a procedure's local variables.

4.4 References

Sections 3.3 and 3.4 of the *Mesa Language Manual* cover the syntax of record and pointer declarations, as well as detailing the operations that can be performed on pointers and records.

4.5 Questions

1) Assume that you are calling a procedure from an interface in order to get the next piece of input data from a file of **CARDINAL**s. Let's say that the **DataIn** interface contains three procedures, declared as follows, that can each get the next **CARDINAL** from the file.

```
GetNextValue1: PROCEDURE [nextValue: CARDINAL];
GetNextValue2: PROCEDURE [nextValue: LONG POINTER TO CARDINAL];
GetNextValue3: PROCEDURE RETURNS [nextValue: CARDINAL];
```

From looking at those declarations, determine which of the following calls will actually get the next piece of data from the file, and decide which call would be the best one to use in a Mesa program from a stylistic point of view.

```
i: CARDINAL ← 0;
DataIn.GetNextValue1[@i];
DataIn.GetNextValue1[i];
DataIn.GetNextValue2[@i];
DataIn.GetNextValue2[i];
@i ← DataIn.GetNextValue3[];
i ← DataIn.GetNextValue3[];
```

2) Given the type declarations below, explain what the differences between calling **AverageData1** and **AverageData2** are.

```
DataHandle: TYPE = LONG POINTER TO Data;
Data: TYPE = RECORD [
  interval, scale, length, maxlength: CARDINAL,
  data: ARRAY [0..0) OF CARDINAL];

AverageData1: PROCEDURE [dataToAverage: Data] =
  BEGIN
  FOR i: CARDINAL IN [0..dataToAverage.length - 1) DO
    BEGIN
    dataToAverage.data[i] ← (dataToAverage.data[i] + dataToAverage.data[i + 1] )/2;
    END;
  END;

AverageData2: PROCEDURE [dataToAverage: DataHandle] =
  BEGIN
  FOR i: CARDINAL IN [0..dataToAverage.length - 1) DO
    BEGIN
    dataToAverage.data[i] ← (dataToAverage.data[i] + dataToAverage.data[i + 1] )/2;
    END;
  END;
```

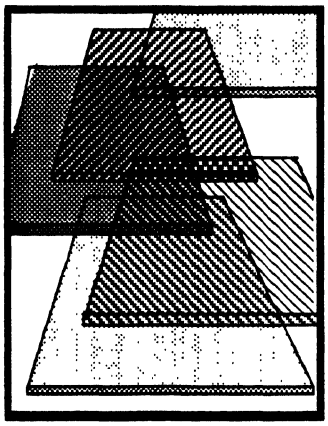
4.6 Exercises

1) Study Appendix D, which appears at the end of this course. It discusses how to debug address faults.

Write two procedures: **Compare**, which compares the values referenced by two pointers, and **Exchange**, which exchanges the value referenced by two pointers. You should declare your procedures to be of type **PointerDefs.CompareProcType** and **PointerDefs.ExchangeProcType**. Store your procedures in a file called CompareAndExchangeImpl.mesa.

To test your procedures, have your program call **PointerDefs.CreateCompareAndExchangeTool** passing the names of the two procedures. We have provided a config file (CompareAndExchangeTool.config) and the implementation for the tool (MesaCourseImplForCompareAndExchangeTool.bcd). Thus, you need to write your implementation, bind the config file, and run CompareAndExchangeTool.bcd.

5



Dynamic storage allocation and management

After reading the last chapter, you undoubtedly realized that pointers were not invented to point at just INTEGERs, when there're so many more interesting data structures in the world. Pointers can point at just about anything, including objects of undeterminable size at compile-time. Of course, constructs such as CARDINALs, with their fixed known length at compile-time, can reside in a local or global frame, but what about a dynamic array or a string of characters? To allocate storage for constructs whose length or usage is not known at compile-time, you need dynamic allocation.

This chapter discusses how you allocate and deallocate storage dynamically, and suggests some ways for managing that storage effectively. We also discuss heaps, which are the storage allocators used for dynamic allocation.

5.1 Preliminary readings

Read the Pilot Memory Managment section (§ 4.6) in the *Pilot Programmer's Manual 11.0*. This section discusses zones and heaps.

Read § 6.6 in the *Mesa Language Manual 11.0*, entitled "Dynamic Storage Allocation." It discusses the Mesa operators NEW and FREE, which are used to allocate and deallocate storage.

5.2 Definition of terms

Dynamic allocation — *Dynamic allocation* acquires storage during program execution.

Dynamic deallocation — *Dynamic deallocation* releases space acquired through dynamic allocation.

Node — A *storage node*, or *node* for short, is a block of allocated storage, often with a record structure.

Storage Leak — A *storage leak* occurs when a program neglects to free all the storage nodes it has allocated, thus reducing the total amount of space available for the system. Leaked storage

degrades the system performance and in extreme cases can cause the system to crash.

Heap — A *heap* is a system-designated area of virtual memory used for dynamic allocation of storage. Heaps, which provide more automatic management of storage than zones, are designed to support the Mesa language operators **NEW** and **FREE**, which allocate and deallocate storage dynamically.

Valid memory location — A location is *valid* if it is currently allocated. A location that has been freed is invalid and should not be referenced.

Zone — A *zone* is a client-designated area of virtual memory used to acquire and manage arbitrarily sized storage nodes.

5.3 Discussion

Heaps are the primary storage allocators in Mesa. They are designed to allocate and free blocks of storage (nodes) of arbitrary size. A heap begins as one large free (unallocated) node somewhere in virtual memory. When a program requests storage, a node is allocated and a pointer to its location is returned to the requesting program. The program then moves values in and out of this node by indirect reference through the pointer. When the program no longer needs the storage, it returns the node to the heap's pool of available (free) nodes.

Clients interact directly with a heap by using Mesa's **NEW** and **FREE** operators and the facilities of the **Heap** interface. Clients use the **Heap** interface to obtain a heap (by either creating one or using one provided by the system) and to destroy a heap. Clients allocate storage from a heap with the **NEW** operator, and return storage to the heap when it is no longer needed with the **FREE** operator.

5.3.1 The system heap

Tajo provides a system-wide heap, called the **systemZone**, for all programs to share. If you need to share storage with other programs, the system heap is a good place to allocate the common storage. You should also use the system heap for programs that only allocate a small amount of storage. You will see an example of using the **systemZone** a little later in the chapter.

You access the **systemZone** through the **Heap** interface. For a program to allocate and deallocate nodes from the **systemZone**, it must **IMPORT** it from the **Heap** interface. Take a look at Section 4.6.2 of the *Pilot Programmer's Manual*, which describes this interface. **Heap.systemZone** is declared as an **UNCOUNTED ZONE**. (Think of this name as historic, not mnemonic.) The size of the **systemZone**, initially 40 pages, is bounded only by the amount of available virtual memory; it expands automatically when a request for storage is larger than the largest free node. The **systemZone** is created when a volume is booted and not destroyed unless the volume is rebooted. Misuse of this heap can be costly, since there is no garbage collection mechanism to free nodes that are no longer in use.

5.3.2 Private heaps

A program can create a private heap. Private heaps exist separately from the system heap, and only programs that have access to a private heap can allocate nodes from it. Like the system heap, private heaps can be grown to unlimited size, although they are typically bounded at 64K pages. The growth of an unbounded heap is limited only by available virtual memory.

Heap.**Create** is declared as follows:

```
Heap.Create: PROCEDURE[initial: Space.PageCount,
  maxSize: Space.PageCount ← Heap.unlimitedSize,
  increment: Space.PageCount ← 4,
  swapUnit: Heap.SwapUnitSize ← Heap.defaultSwapUnitSize
  threshold: NWords ← Heap.minimumNodeSize,
  largeNodeThreshold: NWords ← Space.wordsPerPage/2,
  ownerChecking: BOOLEAN ← FALSE, checking: BOOLEAN ← FALSE]
  RETURNS [UNCOUNTED ZONE];
```

Except for **initial**, the parameters have default values, which you will not (at this point) need to change. **initial** specifies the initial size of the heap, in pages. The system will automatically grow the heap as needed, in steps of **increment** up to **maxSize**.

You should destroy a private heap when you are finished with it. To destroy a private heap, call **Delete**, passing the zone returned by **Create**, like this:

```
Heap.Delete: PROCEDURE[z: UNCOUNTED ZONE, checkEmpty: BOOLEAN ← FALSE];
```

Delete has a second parameter to check if all the allocated nodes have been deallocated. This parameter, defaulted to false, prevents the accidental deletion of a heap still in use.

Space leaks are not as important in private heaps as they are in the **systemZone**, since deleting a private heap frees the entire space occupied by the heap and thereby reclaims any unfreed nodes. Any space leaks would be a potential problem only during the life of the private heap.

5.3.3 Allocating nodes: Using the NEW operator

A conventional way to allocate a node is to determine the amount of storage needed, and then ask the heap for a chunk of that size. The **NEW** operator does this, but it adds the protection of type checking for the allocated node by taking the type of the object as a parameter. It determines the size of the node that needs to be allocated, allocates it, and then returns a pointer to the allocated node.

Mesa enforces type checking on the returned value (the pointer). For example, if you were allocating a record of 3 CARDINALS, your code would look something like this:

```
ptrToRecord: LONG POINTER TO Record ← NIL;
Record : TYPE = [a: CARDINAL ← 0,
  b: CARDINAL ← 1,
  c: CARDINAL ← 2];
  . . .
ptrToRecord ← Heap.systemZone.NEW[Record];
```

The node allocated by the NEW operator (from Heap.systemZone) is of type **Record**. The pointer returned by NEW is thus a LONG POINTER TO **Record**. The variable on the left side of this assignment statement must conform to that type.

You can also initialize a node while allocating it with the NEW operator. To get the default initialization for **Record**, you could change the assignment to be:

```
ptrToRecord ← Heap.systemZone.NEW[Record ← []];
```

To override the default values, to set c ← 10, for example, you could write:

```
ptrToRecord ← Heap.systemZone.NEW[Record ← [c:10]];
```

5.3.4 Deallocating nodes: Using the FREE operator

The FREE operator takes a pointer to a node pointer as its parameter. It frees the node and sets the value of the node pointer to NIL, as in

```
Heap.systemZone.FREE[@ptrToRecord];
```

Setting the pointer to NIL reduces the chances of creating a dangling reference. Figure 5.1 illustrates how FREE works. Without the extra level of indirection in **@ptrToRecord**, the system would not be able to change the value in **ptrToRecord** to NIL.

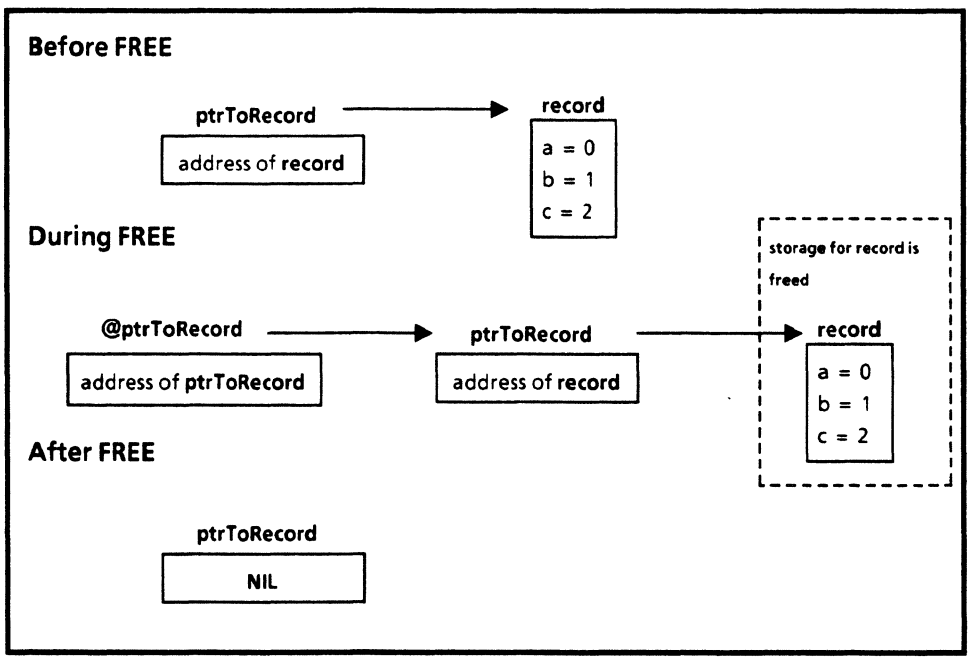


Figure 5.1 Using FREE

5.3.5 The systemMDSZone

The Mesa environment also provides a second system-wide heap. This second heap is called the **systemMDSZone**, and is used for allocating storage pointed to by POINTERs (whereas the **systemZone** is used for allocating storage pointed to by LONG POINTERs). The **systemMDSZone** exists inside a 256-page space called the Main Data Space (MDS), and is limited to that size. Since you will not ordinarily be using the **systemMDSZone**, this chapter discussed only the **systemZone**. However, the two heaps are functionally identical, and all observations about the **systemZone** apply also to the **systemMDSZone**.

5.4 Basic rules for storage management

So far, you've learned the definition of dynamic storage allocation and the procedures to manipulate storage dynamically. However, we haven't covered the best ways to supervise and manipulate space allocation and deallocation. If you had an infinite amount of resources (time and space), then management of those resources would be unnecessary, but since resources are limited and therefore considered to be precious, taking the time to understand storage management can improve your program's (and system's) performance. The following list represents general guidelines for efficient storage management. The rest of this chapter will discuss each item on the list in detail.

1. Hold onto storage only while you are using it.
2. Minimize the number of times you allocate any one item.
3. Keep global frames small.
4. Allocate temporary variables from local frames.
5. Avoid allocating string literals from the global frame.
6. Pass a pointer to an object as an argument rather than the object itself.
7. Use the **systemZone** when the total amount of allocated storage is small, and when use is over a short period of time.
8. Use a private heap when your program (or set of programs) require a lot of storage.
9. Avoid allocation from the **systemMDSZone**.

5.4.1 Hold onto storage only while you are using it

The actual space taken up by dynamically allocated objects is a precious resource, so you should only use it when absolutely necessary. Avoid allocating storage until you need it, and release that storage when you are no longer using it.

5.4.2 Minimize the number of times you allocate any one item

This rule really asks you to think about how a particular item is to be used in your program. When you learn about SEQUENCEs in the next chapter, you'll find that a dynamic array is implemented by copying different-sized arrays back and forth and changing the pointers to create the illusion of a dynamic array. The problem is that repeated allocations and deallocations take time and cause fragmentation within the heap. If you can determine the approximate use of the SEQUENCE in the program, then you can allocate a SEQUENCE that is, for example, four elements larger than what is currently needed, because you know that the SEQUENCE will need space for four more elements in the near future.

You might have noticed that this rule can conflict with the first rule of holding onto storage only while you are using it. You walk a fine line between the time issue and the

space issue and must make tradeoffs between the two to "optimize" your program. When making decisions about tradeoffs, keep in mind such issues as the size of the allocations, the use of the allocated space, and the length of use of the space.

5.4.3 Keep global frames small

Again, you are trying to conserve a precious resource. Global frames reside in the Main Data Space (MDS), a 256-page segment of virtual memory that can be directly addressed by short (16-bit) POINTERs. The MDS is heavily used by the run-time system, so you should avoid placing non-essential demands on it. As you may know, once a program is loaded it stays loaded until it is explicitly unloaded or until the system is rebooted. As a result, many global frames can exist in the MDS; thus the amount of free pages available for other programs to use decreases. Keeping global frames small helps to free the MDS for other tasks.

5.4.4 Allocate temporary variables from local frames

Besides the global frame, you can allocate space from a local frame and from heaps. Storage for local frames also comes from the MDS (see above). The difference between local and global frames (in terms of their burden on the MDS) is that a local frame remains allocated only as long as it is executing. When the procedure returns, the space for the local frame is released. Therefore, when you have fixed-size variables that are not needed for the life of the program, you should allocate them from local frames.

5.4.5 Avoid allocating string literals from the global frame

Suppose you need a string literal in the mainline code. If you allocate a string literal in the mainline code (with or without the L suffix), that literal will take up space in your global frame for the life of the program. To work around this problem, you should have the mainline code call a procedure that includes the code using the string literal. That way, the space for the string literal is released when the procedure finishes.

5.4.6 Pass a pointer to an object as an argument rather than the object itself

In Mesa, procedures pass arguments by value. In a procedure call, the parameters are copied into the local frame of the called procedure. Thus, passing a large object wastes both space and time. Avoid copying large objects in procedure calls by passing a pointer to an object instead.

5.4.7 Use the systemZone when the total amount of allocated storage is small, and when use is over a short period of time

The **systemZone** is created when the system is booted; a private heap, however, is created when your program makes a call to **Heap.Create**. The time needed to make this call can be significant when all you need is a small block of storage for a short period of time. For transient storage, the low overhead of using the **systemZone** is quite attractive.

5.4.8 Use a private heap when your program (or set of programs) requires a lot of storage

Private heaps have several advantages over public heaps. You can restrict the number of clients using a private heap, allowing faster access and minimizing fragmentation. You have potentially faster access because requests for storage must be monitored; thus, the fewer the clients, the less you have to wait in line for storage. Having a small number of clients reduces the amount that allocated nodes are spread around the heap. Since you have no control over where a block of storage is allocated from, the degree of dispersion of nodes will be large if the heap is large. The result of this is that a large heap will have very little of it mapped into real memory at any one time, and accessing the blocks of storage will cause more swapping than if they were allocated within a smaller heap.

5.4.9 Avoid allocation from the systemMDSZone

Since the **systemMDSZone** is contained within the MDS, allocations from this public heap compete with local and global frames for the bounded 256-page resource. The **systemZone** and private heaps, by comparison, are bigger and less congested.

5.5 Summary

This chapter discussed why you need dynamic allocation, and introduced heaps as the most common storage allocator for dynamically allocating nodes. To access the heap facility, you use the **Heap** interface (described in the *Pilot Programmer's Manual*). This interface provides two system heaps, as well as the mechanisms for creating and deleting private heaps.

You use the **NEW** operator to allocate nodes from a heap. When using **NEW**, you specify the heap the node should be allocated from and the type of the node to be allocated. The **NEW** operator calculates the size of storage needed, causes the allocation to occur, and returns a pointer to the node.

When your program is through with a node it must return the storage to the storage allocator. You do this with the **FREE** operator, passing a pointer to the pointer to the node. **FREE** deallocates the node and sets your pointer to **NIL**.

This chapter also presented some guidelines to help you manage storage allocation in a manner that will help your programs' performance. Most of the guidelines are common sense maxims that will help you use the system's time and space efficiently. The guidelines can be boiled down to two basic themes: don't waste time and space, and make a careful tradeoff when time and space issues conflict.

5.6 Questions

Assume that you are using an interface named **Node** that has procedures to allocate and free nodes of type **NodeType**, as defined below:

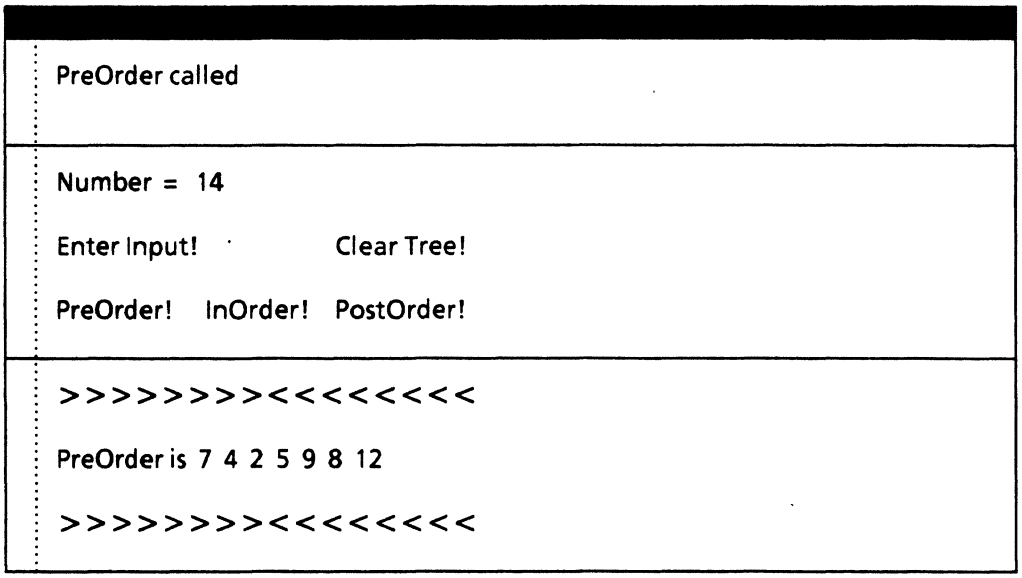
```
NodePtr: TYPE = LONG POINTER TO NodeType;
NodeType: TYPE = RECORD [
  start, end, size: LONG CARDINAL,
  duration: CARDINAL];
AllocateNode: PROCEDURE RETURNS [newNode: NodePtr];
FreeNode: PROCEDURE [nodeToFree: NodePtr];
```

Because the **FreeNode** procedure does not return NIL, you must set the **NodePtr**s to NIL with an assignment statment after you call **FreeNode**. Since the code frees nodes in many places, the following procedure was written to help free nodes. Does this procedure work as intended?

```
OurFreeNode: PROCEDURE [nodeToFree: NodePtr] =
  BEGIN
  Node.FreeNode[nodeToFree];
  nodeToFree ← NIL;
  END;
```

5.7 Exercises

The Tree Traversal Tool allows you to enter numbers into a sorted binary tree. At any point, you can make a preorder, inorder, or postorder traversal of the tree,with the order of traversal displayed in the tool. Your assignment is to complete the tool by writing the procedures **Init**, **EnterNumber**, and **ClearTree** in the module TreeTraversalProblem.mesa. The comments in this

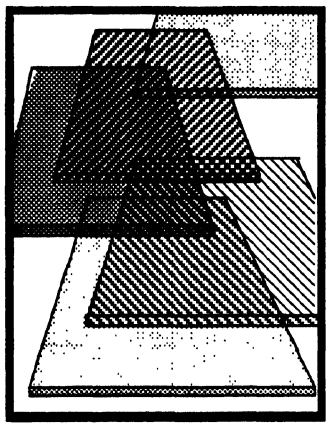


Tree Traversal Tool

module provide a more complete explanation of the procedures that you are expected to write.

You will also need the modules TreeProblem.config, TreeTraversalTool.mesa, and TreeTraversalDefs.mesa.

6



Sequences

Now that you know about heaps, it's time to look at one of the most common heap-dependent Mesa constructs: SEQUENCEs, the Mesa implementation of dynamic arrays. This construct allows you to defer specifying the size of an array until run-time. Because you don't know the size of a sequence until run-time, you have to allocate that sequence from a heap rather than in a local or global frame. This chapter discusses how to allocate, deallocate, and use sequences.

6.1 Discussion

One of the main advantages of using a dynamic array rather than a static array is that you don't have to commit your program to consuming storage before it uses that storage. A program does not allocate storage until it is actually ready to use that storage. You can also change the size of a dynamic array after it allocating it; this comes in handy when you find out sometime in the middle of your program that your sequence is too short. However, a corresponding drawback of using dynamic arrays is the amount of time it takes to allocate a dynamic array during run-time. Static arrays avoid this overhead since they're allocated when the program is loaded.

6.1.1 Declaring a Sequence

Sequences are always declared as the last field in a record. For example, the following declares a record structure that contains a sequence of LONG INTEGERS:

```
ptrToRecord: LONG POINTER TO Record ← NIL;
Record : TYPE = RECORD[
  a: BOOLEAN ← TRUE,
  b: BOOLEAN ← FALSE,
  c: BOOLEAN ← TRUE,
  seq: SEQUENCE length: CARDINAL OF LONG INTEGER];
```

The declaration of a sequence has a *variant tag* part (the length: CARDINAL) and an *element type* part (the LONG INTEGER). The type specification in the variant part determines the type of the indices used to select a sequence element. The range of valid indices is not specified when the sequence is declared but will be computed by the FIRST and SUCC functions when the sequence is allocated. This computation requires that the variant tag specify a valid

IndexType, as defined in the *Mesa Language Manual*. The element type defines the type of object that is being sorted in the sequence, thereby making sequences type-safe.

6.1.2 Allocating a Sequence

To allocate the record to contain a sequence of 10 elements, you could encode:

```
ptrToRecord ← Heap.systemZone.NEW[Record[10]];
```

Record[10] is a type specification describing a RECORD with a sequence part, **seq**, containing 10 LONG INTEGERS. The effect of **Heap.systemZone.NEW[Record[10]]** is to allocate **SIZE[Record[10]]** words of storage from the **systemZone** and return a **LONG POINTER TO Record** to this storage. All fields in the common part of the RECORD (the BOOLEAN fields **a**,**b**, and **c** in the example) are initialized to their default values if default values have been specified (**TRUE**, **FALSE**, and **TRUE** in the example). The sequence tag field, **length**, is set to 10, a value computed automatically using the formula:

```
length ← SUCC^10 [FIRST[CARDINAL]]
```

If the variant tag type uses an enumerated type or a subrange type whose first element is not 0, the value of **length** would still be the value of the tenth successor of the first element of the index set.

The index will range over **[0..10)**, a set of values computed using the formula:

```
[FIRST[CARDINAL]..SUCC^10 [CARDINAL] )
```

The elements of the sequence part are not initialized when the sequence is allocated. Initializing the sequence is your responsibility. However, you can use a constructor of type **Record** in the call to NEW to provide different initial values for the common part of the RECORD , as in:

```
ptrToRecord ← Heap.systemZone.NEW[Record[10] ← [a: FALSE]];
```

6.1.3 Using a Sequence

You can index individual elements of a sequence directly. For example, if **var** is of type LONG INTEGER ,then all of the following are equivalent:

```
var ← ptrToRecord ↑ .seq[3];
var ← ptrToRecord.seq[3];
var ← ptrToRecord[3];
```

Once you have allocated a sequence, you can use it as you would an array:

```
IF ptrToRecord.length > 5 THEN ptrToRecord[5] ← 13;
```

6.1.4 Deallocating a Sequence

You deallocate the record containing the sequence as you would any other node, by using the FREE operator:

```
Heap.systemZone.FREE[@ptrToRecord];
```

6.1.5 VowelSeparatorWithPublicHeap

VowelSeparatorWithPublicHeap is an example of dynamically allocating records with sequences in them. The program, which runs from the Executive, separates user input into vowels and consonants. A sample input would be:

```
VowelSeparator.~ separate the letters in these words by vowels and
consonants
```

Try running the program now.

6.1.5.1 TextSeqBody: the data structure used for storing text

The input is stored in the **TextSeqBody** data structure, which is defined in the **SequenceDefs** interface as:

```
TextSeqBody: TYPE = RECORD [
  length: CARDINAL,
  text: SEQUENCE maxlength: CARDINAL OF CHARACTER];
```

The **length** field specifies the number of elements currently stored in the sequence. The **text** field defines the sequence of characters where the input is stored. The **maxlength** tag field specifies the maximum number of characters that can be stored in the sequence.

TextSeq is a pointer type to this record object, defined as:

```
TextSeq: TYPE = LONG POINTER TO TextSeqBody;
```

6.1.5.2 The procedure Main

In **VowelSeparatorWithPublicHeapImpl**, the procedure **Main** controls translating the input into a **TextSeqBody** and separating the characters into vowels and consonants. However, since the program runs from the Executive, no call to **Main** appears in the program. Instead, the mainline code calls **Init**, which subsequently calls **InitializeVowelSeparator** (from the **SequenceDefs** interface). **InitializeVowelSeparator** registers the program with the Executive, telling it that **Main** is the procedure to call when a user types the `VowelSeparator.~` command. It is important to remember that the procedure, not the whole program, is executed when the command is invoked.

Let's assume a user types into the Executive

```
VowelSeparator.~ separate the characters in these words
```

The Executive recognizes the command and calls **Main**. **Main** declares three variables, **input**, **vowels**, and **consonants**, of type **TextSeq**. These variables will point to **TextSeqBodys** containing the input, the vowels in the input and the consonants in the input. The variables **vowels** and **consonants** are initialized to **NIL**.

SequenceDefs.GetText stores the user's input in **input** and then translates it into a **TextSeqBody**. Because **GetText** must allocate the **TextSeqBody**, we pass the **systemZone** as a parameter to **GetText**. Passing the zone ensures that all nodes are allocated from the same heap. Figure 6.1 depicts the situation at this point.

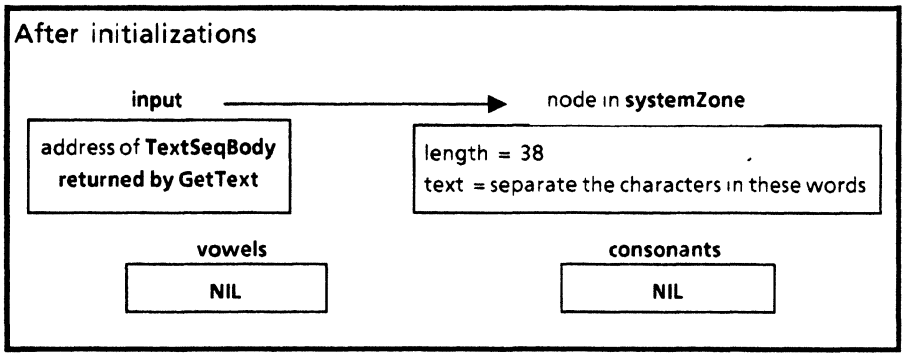


Figure 6.1

Following these initializations, **Main** calls **Separate** to sort the input line into vowels and consonants. **Separate** creates (allocates) two **TextSeqBody**s and returns a pointer to each of these **TextSeqBody**s. Figure 6.2 represents the situation after **Separate** has returned.

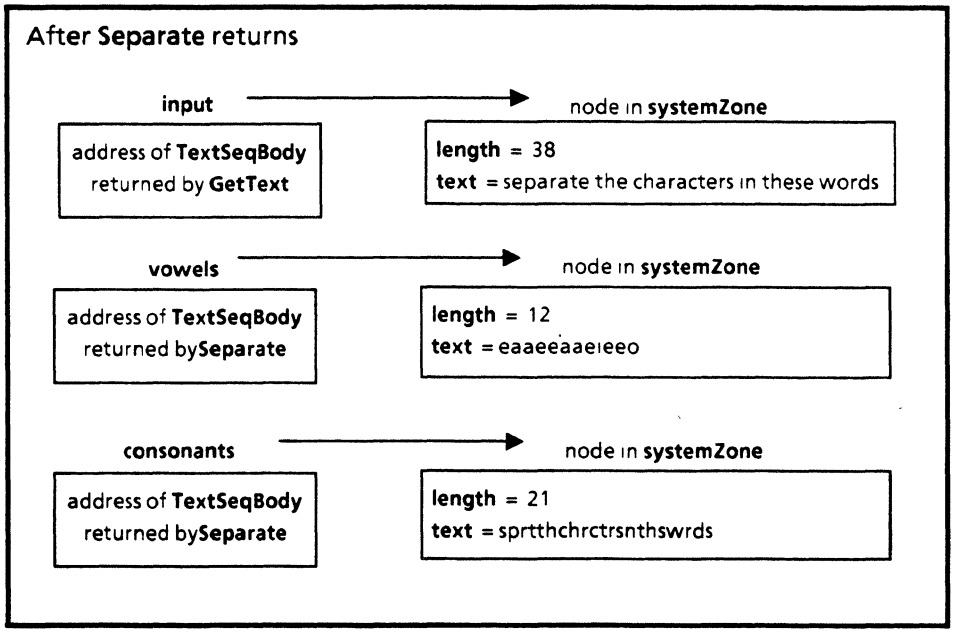


Figure 6.2

Main now outputs the separated characters, first checking to see if there is anything to print. It uses **SequenceDefs.PutComments** and **SequenceDefs.PutText** to print to the Executive. (**PutComments** outputs string literals; **PutText** outputs a **TextSeqBody**.)

Next, **Main** frees the **TextSeqBody**s that were allocated and passed to it:

```
FreeTextSeq[@input];
FreeTextSeq[@vowels];
FreeTextSeq[@consonants];
```

Figure 6.3 shows that all allocated storage is freed before **Main** returns.

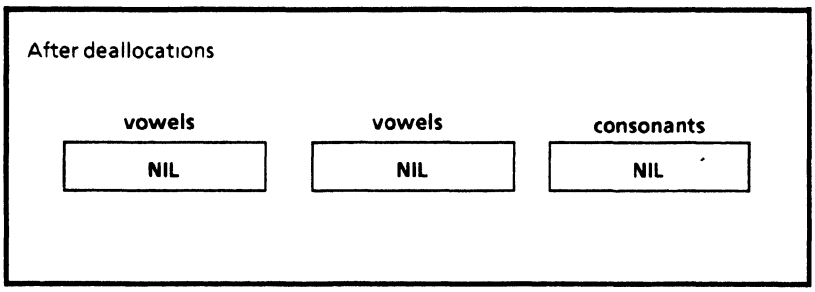


Figure 6 3

Note: Use the information presented in the last chapter (Dynamic Storage Allocation and Management) to figure out the reason for freeing the **TextSeqBody** nodes in this procedure as well as in **AppendChar**

6.1.5.3 How the input is separated

Separate and **AppendChar** are the procedures primarily responsible for separating the characters. **Separate** defines the algorithm for separating the characters; **AppendChar** adds a character into a **TextBodySeq** object.

Separate takes a parameter of type **TextSeq** and separates the characters into two sequences, one containing vowels and the other containing consonants, and returns pointers to each of these **TextSeqBody**s. We use the following algorithm: check if the next character in the input line is alphabetic; if it is, check the alphabetic character to see if it is a vowel. If the character is a vowel, we append it to the **vowels TextSeqBody**. Otherwise, we append it to the **consonants TextSeqBody**.

Note: In the implementation of this algorithm, **Separate** allocates storage for **vowels** and **consonants** from a reasonable guess of vowel and consonant distribution. We did this to minimize the number of allocations done by **AppendChar**.

AppendChar builds the vowel and consonant sequences by adding a character to the end of a **text** sequence. If the **text** sequence is not full (i.e., **length** is less than **maxLength**), then the character can just be appended (by entering it as the next element in the sequence and incrementing **length**).

However, if the **text** sequence is full, the situation is more complicated. **AppendChar** cannot add the next element because there is no room left in **text**. Trying to store into the sequence will cause a run-time error if you compiled with the b switch (bounds checking). If there is no bounds checking, the append will be done, but the element will not be stored into a properly allocated memory location. Instead, it will be stored just beyond the end of the allocated storage. This location could be undefined (causing an address fault), currently allocated for another node (smashing memory by writing over other data), or unallocated (with no assurances on how long the location will stay unallocated and its contents unchanged).

To avoid this situation, you must allocate a new **TextSeqBody** when the sequence is full. (This is how to "grow" a sequence.) You must then copy the contents from the old sequence into the new one. This is what **AppendChar** does; take a look at the code for this procedure.

The series of graphs in Figure 6.4 illustrates the expansion of the sequence when **AppendChar** is asked to append the letter **e** to a full **TextSeqBody**.

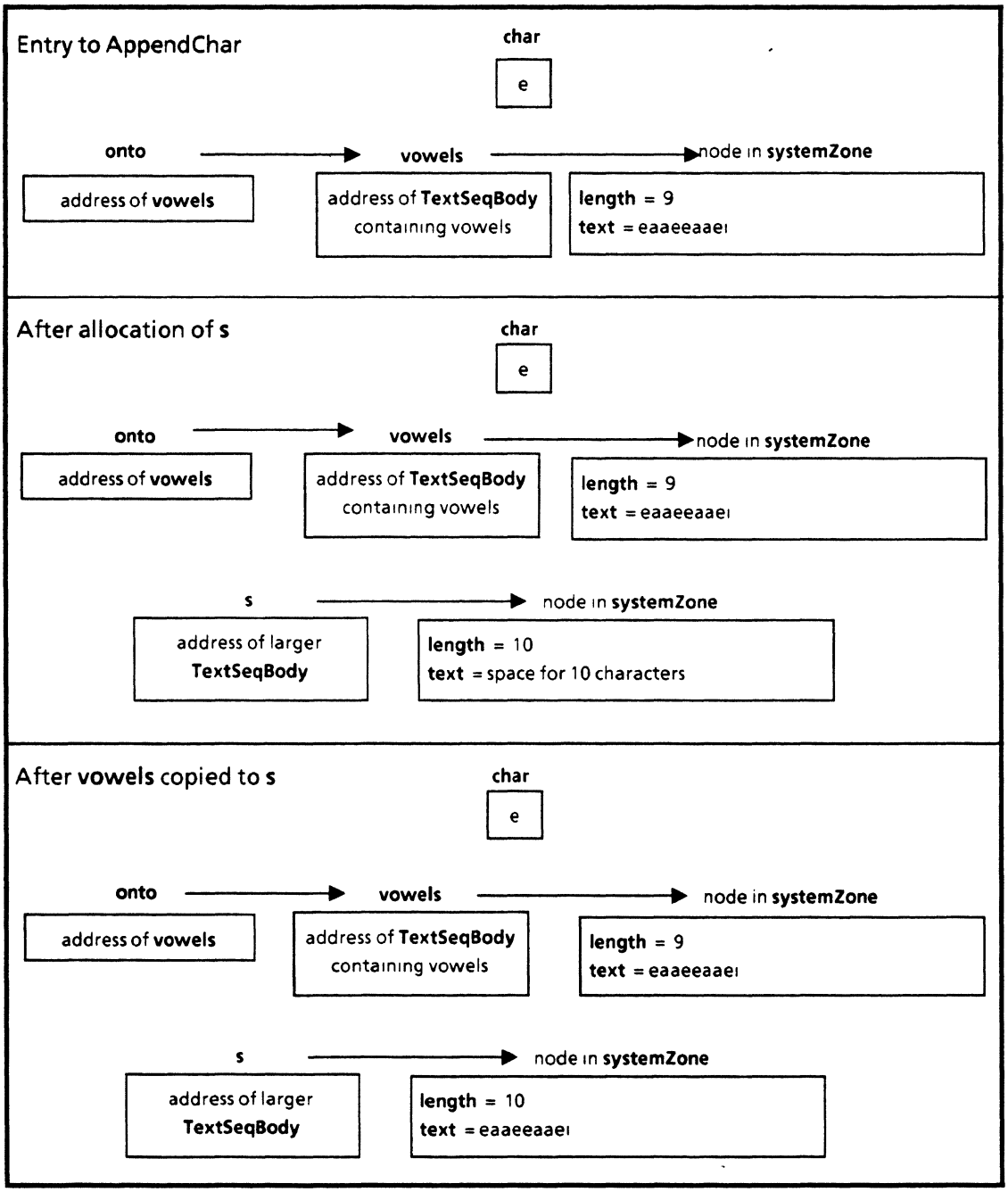


Figure 6.4

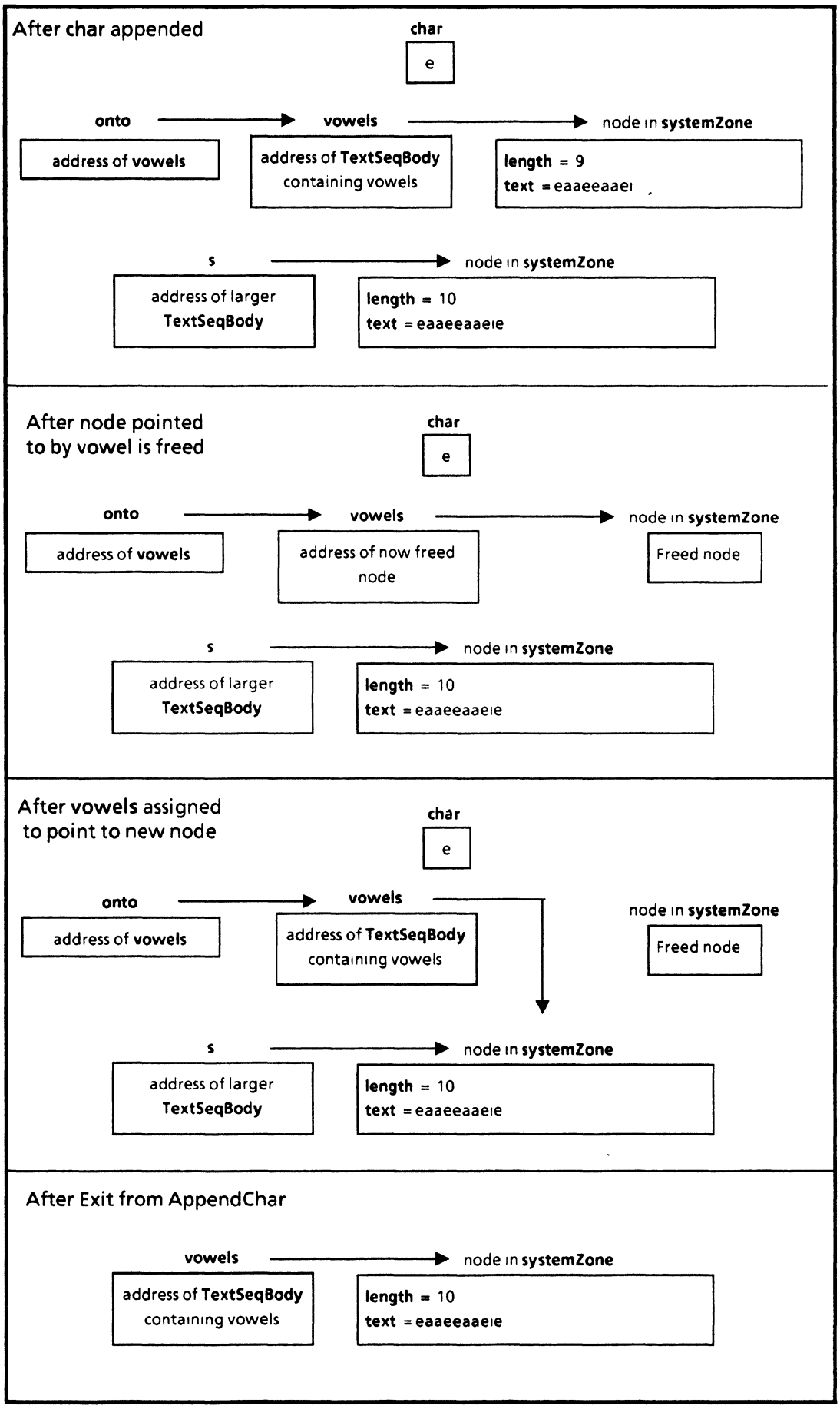

After char appended
char
e
onto
vowels
node in systemZone
address of vowels
address of TextSeqBody containing vowels
length = 9
text = eaaeeaaei
s
node in systemZone
address of larger TextSeqBody
length = 10
text = eaaeeaaeie
After node pointed to by vowel is freed
char
e
onto
vowels
node in systemZone
address of vowels
address of now freed node
Freed node
s
node in systemZone
address of larger TextSeqBody
length = 10
text = eaaeeaaeie
After vowels assigned to point to new node
char
e
onto
vowels
node in systemZone
address of vowels
address of TextSeqBody containing vowels
Freed node
s
node in systemZone
address of larger TextSeqBody
length = 10
text = eaaeeaaeie
After Exit from AppendChar
vowels
node in systemZone
address of TextSeqBody containing vowels
length = 10
text = eaaeeaaeie

Figure 6.5

6.1.6 VowelSeparatorWithPrivateHeap

VowelSeparatorWithPrivateHeapImpl differs from **VowelSeparatorWithPublicHeapImpl** only in that it uses a private heap instead of the **systemZone** to allocate **TextSeqBody**. This module is part of the configuration called `VowelSeparatorWithPrivate-Heap.bcd`. It runs from the Executive command `VowelSeparator.~`. Run the program to verify that it acts like **VowelSeparatorWithPublicHeap**, and then study `VowelSeparatorWithPrivateHeapImpl.mesa`. Pay particular attention to the creation and deletion of the private heap, and to the allocation and deallocation of nodes.

6.2 Summary

A sequence appears as the last field in a record. It contains a variant index field in its declaration, which becomes fixed at the time of allocation. To enlarge a sequence, therefore, you must:

1) allocate a new, larger one,
2) copy the data from the full sequence into the new one,
3) free the old sequence , and
4) adjust the pointers so the new sequence is referenced by the pointer that referenced the original sequence.

6.3 Reference

The *Mesa Language Manual 11.0* section entitled "Sequences" is a thorough reference.

6.4 Exercises

Complete a program that takes a string of characters as input and stores the characters alphabetically in queues according to the number of queues that the user specifies. For example, if the input were ***James! Where are you?!***, and the user wanted four groups of characters, the result would look like this:

For Group 0 (A-G):
a e e e a e

For Group 1 (H-N):
J m h

For Group 2 (O-T):
s r r o

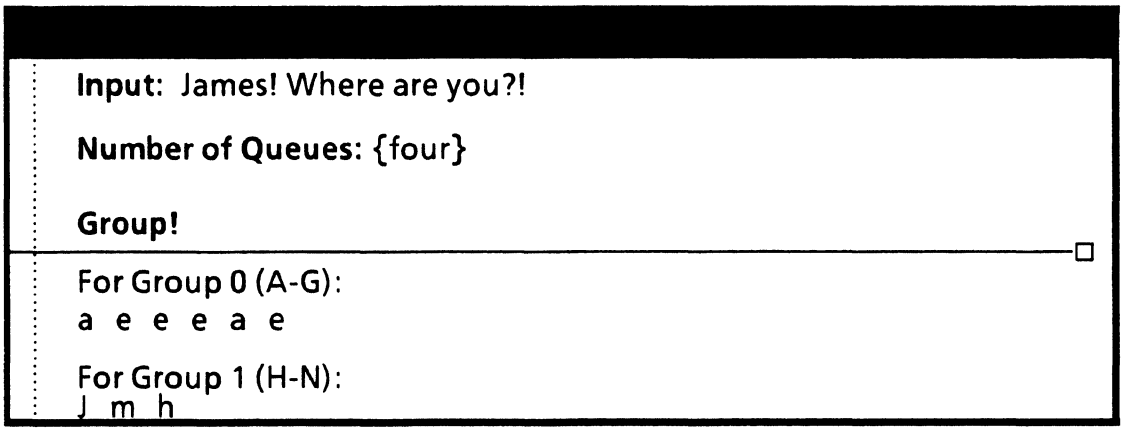
For Group 3 (U-Z):
W y u

For Last Group (non-alphabetic characters):
! SP SP SP ? !

Done.

The program runs from a tool, which consists of the following modules:

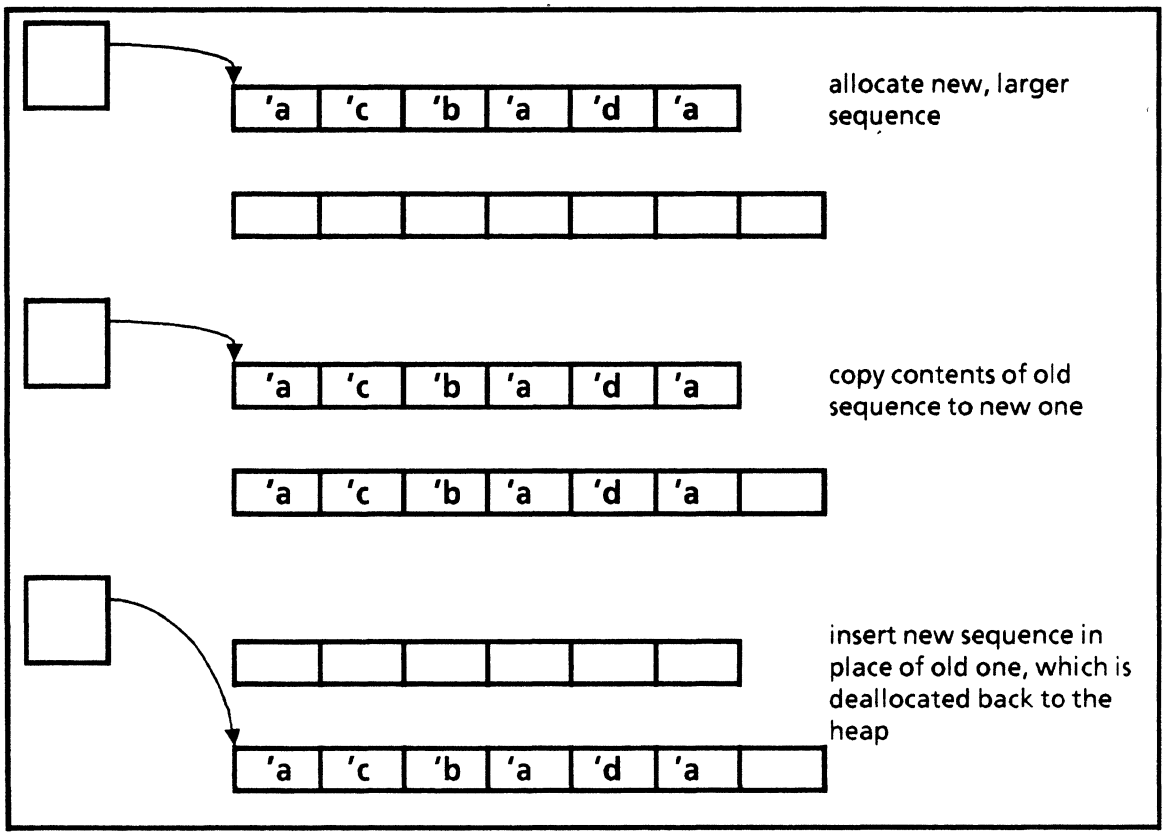
LetterTool.mesa: contains tool-related code (I/O);
LetterImpl.mesa: contains the implementation code that actually processes the input;
LetterDefs.mesa: is the interface for these modules;
LetterConfig.config: is the configuration module for the above.



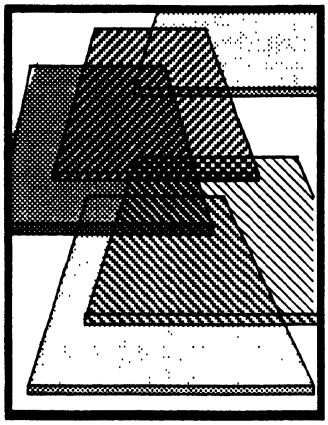
The tool as it appears when *LetterConfig.bcd* is executed.

When **Group!** is invoked, the CommandItem procedure **Group** (in **LetterTool**) passes the input string and the number of desired queues to procedure **ProcessInput** (in **LetterImpl**). **ProcessInput** calls **InitQueues** to create and initialize the queues. It then calls **CutUpAlphabet** to determine which characters each queue will handle. **ProcessInput** then calls **StoreLetters** to actually put the characters into the queues. Finally, **PrintResults** (in **LetterTool**) is called to display the results of the user-requested action.

There are two instances where you must consider dynamic storage allocation. First, there is the initial allocation from a heap, where two factors are variable: the number of queues and the size of each queue. Secondly, there is the expansion of a queue when the sequence that represents the queue is full. The "expansion" really consists of allocating a new sequence that is larger than the original one, copying over the original sequence into the new one, inserting the new sequence in place of the original one, and freeing the space that the original sequence occupied (see diagram on next page).



"Expansion" of a sequence



Strings

In this chapter we introduce Mesa strings. Although you may not have realized it, the classic implementation of a string as an array of characters with an associated length actually involves a pointer. In languages such as Pascal, these string pointers are hidden from you. Mesa, on the other hand, makes this string pointer explicit and puts it under program control.

This chapter will show how string pointers differ from standard pointers, and how string use is facilitated by using public interfaces.

7.1 Definition of terms

String A *string* is conceptually a sequence of characters, such as "that". A string is represented in Mesa as a pointer to a record that contains an array of characters and a length.

7.2 Discussion

The structure of a STRING is very similar to the structure of the **TextSeqBody** in the last chapter. As described in the *Mesa Language Manual* (§6.1), the type LONG STRING is:

```
LONG STRING: TYPE = LONG POINTER TO StringBody;
StringBody: TYPE = MACHINE DEPENDENT RECORD [
  length: CARDINAL,
  maxlength: CARDINAL,
  text: PACKED ARRAY[0..0) OF CHARACTER];
```

The **length** field of the string is, by convention, the current length of the string in the **text** array. The **maxlength** field specifies the maximum length of the string. This field is read-only because the size of a string is fixed when it is allocated.

The **text** field is a special form of array, which used to be the primary way for providing dynamic arrays in Mesa, before SEQUENCEs were added to the language. It declares an array (as the last field in a record) to have an undetermined length (indices from **[0..0)**). The compiler, however, interprets this field as an array with zero length. This has interesting

effects on string pointer manipulations in assignment and comparisons, as discussed below.

7.2.1 Allocating a STRING

There are four ways to allocate a **STRING**:

- Allocate fixed-sized storage from the local or global frame of a program.
- Assign a string literal to a string variable. String literals are automatically allocated in the local or global frames of your program.
- Use the **NEW** operator to allocate storage from a heap.
- Use procedures provided by the **String** interface (discussed in the *Pilot Programmer's Manual*, §7.3) to allocate storage from a heap.

STRINGs are the only Mesa construct that can be allocated by an explicit request for space from a local or global frame. For example, the following declares a variable **string** and allocates space for up to 256 characters from the same local or global frame as the statement itself:

```
string: LONG STRING ← [256];
```

Sometimes, however, you may want to use known text as a string, for example, to print a message, prompt the user for input, or explain how to use the program. Mesa provides string literals for these uses, such as:

```
globalString: LONG STRING ← "Hi There";
localString: LONG STRING ← "Hi There"L;
```

Both of these strings are initialized to point to a record whose **length** and **maxlength** fields are 8 and whose **text** field contains the characters H, i, , T, h, e, r, e. **globalString** is allocated out of the program's global frame; **localString** is allocated from the local frame (denoted by the suffixed **L**.)

When a string literal is inappropriate, you will often allocate the string from a heap (or it will be allocated for you). As a pointer, a **STRING** is well suited for the **NEW** and **FREE** operators. The following example accomplishes what our first example did, except it gets its storage from the heap instead of the local or global frame of the program. It declares a **LONG STRING** and initializes it to **NIL**. When space is needed, it uses the **NEW** operator on the **StringBody** type to allocate a space for 256 characters:

```
string: LONG STRING ← NIL;
...
string ← Heap.systemZone.NEW[StringBody[256]];
```

To deallocate the string, you use the **FREE** operation:

```
Heap.systemZone.FREE[@string];
```

Because strings are very common in Mesa programs, there is a system interface (called **String**) that implements primitive string operations such as allocating, copying, and

comparing strings. The **MakeString** and **FreeString** procedures in this interface work much like **NEW** and **FREE** for allocating and deallocating a string.

String.MakeString takes two parameters: the heap from which the node is to be allocated, and the maximum size of the string:

```
String.MakeString: PROCEDURE[z: UNCOUNTED ZONE, maxlength: CARDINAL];
```

Thus, the following code is equivalent to calling **Heap.systemZone.NEW[StringBody[256]]**:

```
string: LONG STRING ← NIL;
...
string ←String.MakeString[z: Heap.systemZone, maxlength: 256];
```

FreeString takes as parameters a string and the heap from which the string was allocated:

```
String.FreeString[z: UNCOUNTED ZONE, s: LONG STRING];
```

FreeString frees the space occupied by the **StringBody**; you are responsible for setting the string to **NIL**.

7.2.2 Caveats in using strings

Besides the usual pointer considerations, there are a few peculiarities related to the structure of strings that you should be aware of. The following examples demonstrate common **STRING** misuse. Try to figure out the effect of each group (and the error) before looking at the explanations.

7.2.2.1 Initializing strings from the current frame

```
string1, string2: LONG STRING ← [256];
```

This is analogous to

```
number: CARDINAL ← 5;
ptrToNumber1, ptrToNumber2: LONG POINTER TO CARDINAL ← @number;
```

It points both strings to the same 256-character space, which is most likely not what was intended. To point each string to its own space of 256 characters, you would code:

```
string1: LONG STRING ← [256];
string2: LONG STRING ← [256];
```

7.2.2.2 Comparing strings

Consider the following attempts to compare **string1** and **string2**:

```
string1: LONG STRING = "Hi There"L;
string2: LONG STRING = "Hi There"L;
  1) IF string1 = string2 THEN ...
  2) IF string1↑ = string2↑ THEN ...
  3) IF string1.text = string2.text THEN...
```

All three string comparisons are incorrect. The first compares the value of the pointers, and not the objects which these pointers reference. This comparison asks if the two

pointers point to the same object, not if the two objects pointed to are equal. For this example, the result is FALSE, even though the two strings contain the same text.

The second comparison seems like it should work: it compares the objects referenced by the two pointers. Unfortunately, when the compiler generates code for the comparison, it treats strings as having text fields with zero length without taking run-time sizes into account. Since the sizes are zero, the statement only compares the **length** and **maxlength** fields of the two strings (equivalent to **string1.length = string2.length AND string1.maxlength = string2.maxlength**). For this example, the result is TRUE. However, this comparison does not really compare the two strings.

The final statement fails for the same reason as the second comparison. When the compiler generates the comparison code, it treats the text field as an empty array [0..0). The compiler thinks it is comparing two empty objects. (The result of this is left for you to determine. The value is definitely a constant, but is it TRUE or FALSE?)

To compare two strings properly, you need to compare each element in their arrays. This is simple to encode, and you may want to try it as a short exercise. However, the **String** interface provides **String.Equal** and **String.Compare** to perform these primitive STRING operations; take a look at their descriptions in the **String** section of the *Pilot Programmer's Manual.*

7.2.2.3 Assigning strings

```
string1: LONG STRING ← [256];
string1 ← "Copy this into the string, please"L;
```

This set of statements does not, in fact, copy the string literal into the space allocated from the current frame. The first statement declares the variable **string1** and initializes it to point at a **StringBody** with a 256-character text field. The second statement assigns **string1** to point to a new **StringBody**, one which contains the literal "Copy this into the string, please", making the original 256-character text field leaked storage that can no longer be referenced.

To correctly copy this literal into **string1** you could use either **AppendString** or **Copy** from the **String** interface.

7.2.3 Using the String interface.

The **String** interface provides routines for doing common string operations: comparing, appending, copying, and allocating. A number of the appending and copying routines also involve allocation. You will need to be familiar with these routines to complete the exercises at the end of this chapter.

7.3 Summary

This chapter has not really presented anything new. All string use involves pointers, and you have already learned the intricacies of pointer usage. However, STRINGs do cause problems, often because programmers are used to strings as arrays of characters. Just remember that in Mesa, the pointer has been put under program control. The structure of Mesa STRINGs is another potential source of difficulty. Because the **text** field is seen by the

compiler as having zero length, comparisons among **StringBodies** are not as straightforward as among other pointer objects. However, the **String** interface supplies most common string routines, so you will not have to worry about writing them yourself.

7.4 References

Section 6.1 of the *Mesa Language Manual* briefly describes the record structure of a **STRING** and discusses how to declare and use string variables.

Section 7.3 of the *Pilot Programmer's Manual* describes the **String** interface, including many procedures for manipulating **STRING**s.

7.5 Exercises

In this exercise, you will modify a line editor that runs in a tool window. The line editor currently calls several string manipulation procedures defined in the **String** interface. These procedures allocate and deallocate strings from a heap, free strings, copy strings, and replace strings. In addition, the tool implements some more advanced string features such as substring operations. Your assignment is to implement the same procedures through another interface called **String2**. You will write the implementations to this new interface and bind the modules together into a configuration.

You will need the following modules for this assignment:

EditorDefs.mesa
EditorImpl.mesa
EditorTool.mesa
String2.mesa
Editor2.config

Notice that none of the modules currently use **String2**. You should:

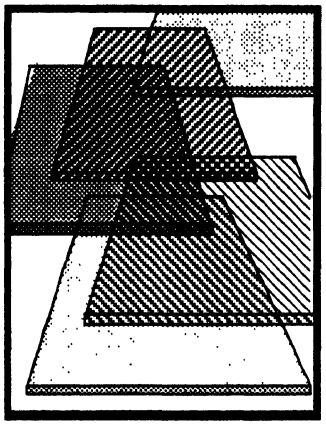
1) Change all **String** references in the module EditorImpl to **String2**.
2) Create an implementation module for **String2**.
(Name it String2Impl.mesa.)
3) Move the procedure **InsertString** from the module EditorImpl to String2Impl.mesa.
4) Change all **InsertString** references to **String2.InsertString**.
5) Write implementations for the procedures listed in **String2**.
6) Change the configuration Editor2.config to reflect the new program modules.

All of the procedures in **String2** are taken directly from the Pilot **String** interface. You should take a look at the **String** documentation in the *Pilot Programmer's Manual* to get an idea of what each of these procedures is supposed to do.

This might also be a good time for you to familiarize yourself with a tool called DebugHeap. This tool allows you to check for storage leaks in your programs. To find out how to use this tool, check your *XDE User's Guide*.

Notes:

8



Signals

Signals are a software interrupt facility used when exceptional conditions occur during the execution of a program. Mesa's signal mechanism is more flexible and powerful than the exception handling facilities provided by most other languages or systems.

This chapter provides several examples that illustrate how to suspend program execution to handle an exception, how to provide code to handle the exception, and how to continue program execution afterwards. At the end of the chapter, you will apply your understanding of signals to write a program that both generates and handles signals.

8.1 Definition of terms

Exception — An *exception* is an unusual event that programs must be prepared to handle, such as end-of-file or an invalid input.

Signal — A *signal* is a Mesa language construct used to help handle exceptional conditions encountered during program execution. Signals are like procedures except that the code to be executed for a signal call is determined at run-time.

Error — An *error* is a Mesa language construct similar to a signal, except that program execution can be resumed after a signal, but not after an error. The word "signal" is used to refer to both signals and errors, except where explicitly noted.

Catch Phrase — A *catch phrase* is a Mesa construct that establishes code to catch one or more signals. The catch phrase contains the code to be executed when the exception occurs.

Signaller — The *Signaller* is the program that receives control when a signal is raised, attempts to find an associated catch phrase, and executes the code in the catch phrase.

Call Stack — The *call stack* is a Mesa processor data structure containing a frame for each procedure invocation that has not yet returned. The call stack is ordered by most recent invocation, and is referred to as growing

downward. Therefore, going "up" the call stack means going from the most recently called procedure record toward the oldest.

Raise — To *raise* a signal is to instruct the Signaller to look in each procedure on the call stack until it finds a procedure with a catch phrase for that signal. The Signaller searches up the call stack.

Reject — A catch phrase *rejects* a signal when it is not prepared to handle it (the Signaller continues searching up the call stack for another catch phrase for the same signal). A catch phrase rejects a signal either by explicitly placing a **REJECT** statement in the code or by not specifying how to resolve the signal.

Resume — To *resume* a signal is to tell the Signaller to resume program execution immediately after the statement that raised the signal. As when returning from a procedure call, any values returned by the signal are passed back to the statement that raised the signal. An **ERROR** cannot be resumed.

Continue — To *continue* a signal is to tell the Signaller to resume program execution at the statement following the one to which the catch phrase belongs. Thus, control is resumed in the procedure where the signal was caught, not the procedure that raised the signal.

Retry — To *retry* a signal is to tell the Signaller to re-execute the statement to which the catch phrase belongs.

Goto, Exit, Loop — These are Mesa statements that can be used, in addition to **REJECT**, **RESUME**, **CONTINUE**, and **RETRY** to indicate where execution is to occur after the signal handling mechanism is finished.

Unwind — *Unwind* is a special signal raised by the Signaller to allow procedures about to be deleted from the call stack to clean up their data structures (e.g. deallocate storage and close files). When there is an unconditional branch out of the catch phrase (**GOTO**, **EXIT**, **LOOP**, **CONTINUE**, **RETRY**) the Signaller raises the unwind signal at the point where the original signal was raised.

8.2 Discussion

Generally speaking, there are two methods for detecting an event at which you are not present. You can continuously *poll* an observer or participant of the event, or you can have the observer or participant *notify* you. If the event you are checking for is reasonably predictable and you have time, polling may be convenient. However, if the event is unlikely to occur or happens intermittently, notification may be more convenient. The choice of method always involves a trade-off between the inefficiency of polling when nothing has happened and the inconvenience of being interrupted for notification.

Most computer languages do not implement a notification system for errors or exceptions. Since computers execute so quickly, the inefficiency of polling can often be tolerated, particularly when compared with the expense of providing a notification capability.

However, there are cases, such as device time-out, when notification is an easier, more logical, and more efficient way to communicate the information that an exception has occurred. For example, while you are transferring files from a file server, it is a rare event for the connection to time out, and notification is preferable to polling. Mesa provides the *signal* facility for cases such as this.

Signals also make it easier for someone who is reading a program to see the exceptions that are being handled and to identify the code that handles them. A signal always indicates the occurrence of a rare event. Status polling doesn't have this feature: since it is usually implemented by boolean checking, it is not always obvious which of the two is the rare case.

8.2.1 How signals work

The declaration of a signal is similar to that of a procedure: there may be a parameter list and a returns list. But instead of being initialized to an actual body of code, a signal is initialized by the symbol CODE. Here's a sample signal declaration:

```
StringBoundsFault: SIGNAL[s: LONG STRING]
  RETURNS [ns: LONG STRING] = CODE;
```

A signal is raised when a SIGNAL (or ERROR) statement is executed, as in:

```
SIGNAL StringBoundsFault [string];
```

The body of code to be executed for a signal is determined at run-time (dynamic binding). When a signal is raised, normal execution is suspended and control is passed to the Signaller, which is part of Mesa's run-time support. It is the Signaller's responsibility to find and execute the bodies of code to handle the signal.

These bodies of code are called *catch phrases*. Each catch phrase can have code for one or more signals, in a structure similar to a SELECT statement. For example:

```
StringBoundsFault => BEGIN
  ns ← AllocNewString [s: length + 10];
  CopyString [from: s, to: ns];
  DeallocateString [s];
  RESUME [ns];
  END;
String2 => BEGIN...END;
```

A catch phrase can occur in one of two places: explicitly on a procedure call (denoted by "!"), or after the word ENABLE in a BEGIN-END block. A !-defined catch phrase will catch a signal raised while the called procedure is executing, or while procedures called by that procedure are executing. An ENABLE-defined catch phrase does the same thing for every procedure call in the surrounding BEGIN-END block, and in addition will catch any signal raised directly in the BEGIN-END block. In the code fragment below, **Signal1** would be caught only if it is raised while **Procedure1** is executing. **Signal2**, on the other hand, would be caught if it is raised through **Procedure1**, through another procedure call in the block, or directly, as in the SIGNAL **Signal2** statement.

```
BEGIN
ENABLE Signal2 = > BEGIN ... END;
...
Procedure1[...!Signal1 = > BEGIN ... END];
SIGNAL Signal2;
...
END;
```

Catch phrases form a dynamic list that is ordered by the call stack, and by BEGIN-END blocks within each procedure call. In the example above, the catch phrase for **Signal1** in the call to **Procedure1** is nested below the ENABLE-defined catch phrase for **Signal2**. These two catch phrases are followed by any ENABLE-defined catch phrases in enclosing BEGIN-END blocks and then any catch phrase on the procedure one higher on the call stack, etc. This list of catch phrases is terminated at the root of the call stack, where there is an implicit catch phrase that catches any signal that has not been otherwise dealt with and raises the error **UncaughtSignal**.

When a signal is raised, the Signaller goes up the program's call stack looking in the BEGIN-END blocks of each procedure on the stack for a catch phrase that recognizes the signal. When an appropriate catch phrase is found, the Signaller executes a call to it. The parameters (if any) are passed and the catch phrase is entered. As with procedures, the signal's parameters can be referenced inside the body of the catch phrase. (The signal's parameters have precedence over any other symbols of the same name. Within a **StringBoundsFault** catch phrase, for example, **s** and **ns** refer to the signal's parameters.)

After the catch phrase is entered one of three things can happen:

- **Resume** A RESUME statement tells the Signaller to conclude processing of this signal and resume execution of the program at the point where the signal was raised. Its syntax is just like RETURN, and the signal can return values if it is defined that way. RESUME is not legal if the signal is an ERROR.
- **Exit** EXIT, CONTINUE, RETRY, LOOP, and GOTO are the statements used to conclude processing a signal by jumping to a point outside the catch phrase. When a jump occurs, the Signaller raises the special signal UNWIND to inform procedures more deeply nested on the call stack that they are about to be deleted. (UNWIND is discussed in §8.2.5.)
- **Reject** This tells the Signaller to continue processing this signal and to pass it to the next higher catch phrase. There are three ways that a catch phrase can reject a signal: explicitly (with a REJECT statement), implicitly (by not catching the signal), or by first catching the signal, and then "falling off the end" without executing a RESUME, EXIT, CONTINUE, RETRY, LOOP, or GOTO.

8.2.2 Resume

After handling an exception, it's possible to return to the code that raised the signal. This is desirable if the code executed in the catch phrase has eliminated the source of the exception.

For example,

```
Node: TYPE = RECORD[
  index: CARDINAL,
  sequence: SEQUENCE length: CARDINAL OF SeqType];
PtrToNode: TYPE = LONG POINTER TO Node;
seq: PtrToNode;
...

GrowSequence: PROCEDURE [seqNeedsLengthening: PtrToNode]
  RETURNS[lengthenedSeq: Ptr ToNode] = { . . . };
--If seqNeedsLengthening is NIL then GrowSequence allocates a new sequence and
--returns a pointer, lengthenedSeq, to it. Otherwise, GrowSequence allocates a
--new sequence longer than seqNeedsLengthening.length, copies the data from
-- seqNeedsLengthening↑ to lengthenedSeq↑, frees seqNeedsLengthening↑,
--and returns a pointer, lengthenedSeq, to the new sequence.

InsertNode: PROCEDURE [object: SeqType] =
BEGIN
  IF (seq = NIL) OR (seq.index = seq.length) THEN seq ← GrowSequence[seq];
  seq[seq.index] ← object;
  seq.index ← seq.index + 1;

ProcessNextObject PROCEDURE[object: SeqType];
BEGIN
  IF DuplicateObject[object] THEN TakeAppropriateAction
  ELSE InsertNode[object];
END;
```

If the sequence is full, **InsertNode** calls **GrowSequence[seq]** to lengthen the sequence. It would improve modularity if **InsertNode** knew only how to add data to the sequence, and did not attempt to handle the exception. Instead, when the sequence is full, **InsertNode** would raise a signal to inform a catch phrase on the call stack (presumably one that knows how to grow the sequence) to take care of the problem. Once the sequence has been lengthened, the signal can be RESUMEd, returning control to **InsertNode**, which can then continue to add data to the sequence.

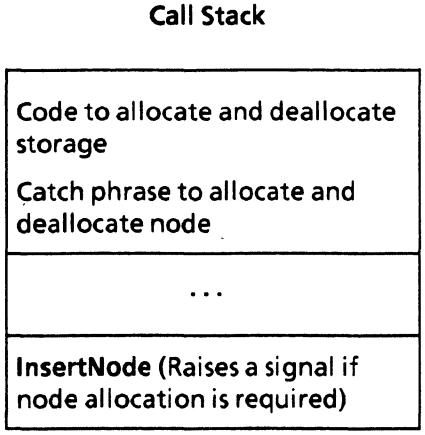


Figure 8.1

Figure 8.1 illustrates this scheme. It shows a box for a procedure that knows how to allocate and deallocate storage, and, lower on the stack, a box for the procedure **InsertNode**, which communicates with the previous procedure by raising a signal when it is necessary to allocate a new node.

Let's look at how to add the appropriate signal-raising and signal-handling code to the above fragment to accomplish this design.

First, we declare the following signal:

```
SequenceBoundsFault: SIGNAL[oldSeq: PtrToNode]
  RETURNS [newSeq: PtrToNode] = CODE;
```

We want to raise this signal when the sequence needs more space. This can occur either when the sequence needs to be initialized for the first time, or when the sequence needs to be extended beyond its present boundaries. We have modified **InsertNode** as follows:

```
InsertNode: PROCEDURE [object: SeqType] =
BEGIN
  IF seq = NIL THEN seq ← SIGNAL SequenceBoundsFault[seq]; --raise signal
  UNTIL seq.index < seq.length DO
    seq ← SIGNAL SequenceBoundsFault[seq];                --raise signal
  ENDLOOP;
  seq[seq.index] ← object;
  seq.index ← seq.index + 1;
END;
```

The first line of code checks to see if the sequence is **NIL**. If it is, it raises **SequenceBoundsFault**, passing **seq** as the sequence to be extended. When the signal is raised, normal program execution is suspended. The Signaller takes over and begins to examine catch phrases on the call stack. An appropriate one is found in the call to **InsertNode** in the revised **ProcessNextObject**:

```
ProcessNextObject PROCEDURE[object: SeqType];
BEGIN
  IF DuplicateObject[object] THEN TakeAppropriateAction
  ELSE InsertNode[object! SequenceBoundsFault = > --catch signal
                    RESUME[GrowSequence[oldSeq]]];
END;
```

The body of the catch phrase is dynamically bound to the signal call and is executed after passing in the parameter, **oldSeq**, of **SequenceBoundsFault**. This catch phrase only contains one line of code, the **RESUME** statement, which calls **GrowSequence[oldSeq]**. **GrowSequence** takes **oldSeq**, allocates a larger one (copying the data from **oldSeq** ↑), and returns the new sequence. The signal is then resumed, which passes control back to **InsertNode**, in the statement that raised the signal. At this point, **seq** is assigned the newly allocated sequence returned by the **RESUME**. **InsertNode** now has a freshly allocated sequence into which it can insert data.

The **UNTIL** loop handles the case of no space for new data in the existing sequence. **SequenceBoundsFault** works in the same way as just described. (The raising of the signal appears in a loop for robustness, in case the catch phrase does not allocate enough new space to cover **InsertNode**'s needs in a single call. The copying operation described above is

performed each time the signal **SequenceBoundsFault** is raised in the UNTIL loop of **InsertNode**.)

Figure 8.2 shows the state of the call stack when a full sequence is encountered. **ProcessNextObject** has called **InsertNode**, which has raised **SequenceBoundsFault[seq]** to signify the need for a larger sequence. This resulted in a run-time system call to the Signaller, which created a call to the catch phrase for **SequenceBoundsFault** (labelled **CatchFrame: ProcessNextObject** in the figure). The catch phrase has then called **GrowSequence**, which will allocate a new sequence and deallocate the old one. When **GrowSequence** returns, the catch phrase will execute a RESUME, and return the longer sequence to **InsertNode**.

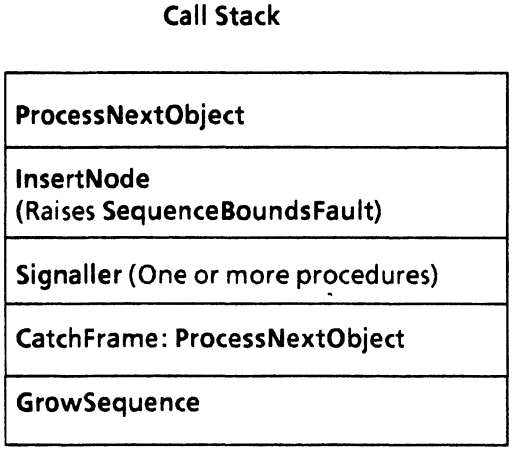


Figure 8.2

Signals do not automatically return after execution of a catch phrase; you must indicate where control is to continue if you do not want the Signaller to continue up the call stack looking for catch phrases. In this case we wanted to return to the point where the signal was raised, so we used RESUME. Allowing a signal to "fall off the end" of a catch phrase, is not a RESUME, but rather an implicit REJECT.

8.2.3 Retry and continue

There are times when an unsuccessful action raises a signal and it is appropriate to repeat the action until it is successful. For instance, if the File Tool is unable to open a connection to a specified service on the first try, you might want it to keep trying until it was successful or until you told it to stop. **RetryExample** provides an example of this. Run the program by typing **RetryExample** in the Executive, followed by the name of a server. (You should move the program to the Tajo volume via Command Central, etc.) The program simulates a failure to open a connection to the specified server. (Notice the message to that effect.) On the second attempt the simulated connection is made.

Take a look at the source listing to see how this retry was accomplished. **RetryExampleImpl** primarily consists of one procedure, **RetryProc**, which gets the server name from the user's input and then tries to open a connection. Inside **OpenConnection**

the signal **TimeOut** can be raised if the connection is not established within a certain time period. This signal is defined in the **SignalsDefs** interface as

```
TimeOut: ERROR;
```

OpenConnection has been rigged for this example to raise the signal **TimeOut** the first time it is called. We catch this signal in the call to **OpenConnection**, print a message to the user to explain the problem and **RETRY**. This causes the program to make the procedure call to **OpenConnection** again. The second call succeeds and we post a message indicating the open connection. Figure 8.3 shows the situation after the signal is caught.

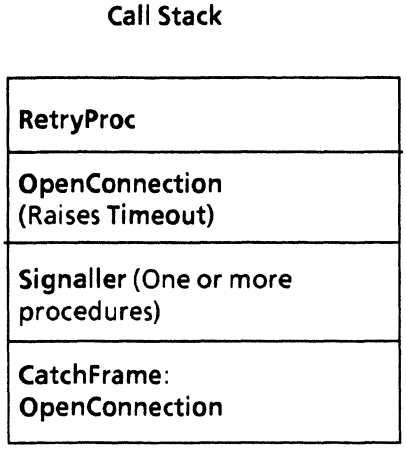


Figure 8.3

When the catch phrase executes the **RETRY**, *there is a jump to the beginning of the statement that contains the catch phrase*, in this case, the call to **OpenConnection**:

```
OpenConnection[server! Timeout => BEGIN ...RETRY END]
```

When an **ENABLE** clause is used to define the catch phrase, the **BEGIN-END** block surrounding the **ENABLE** clause is the "statement that contains the catch phrase." For example, if **RetryProc** had been coded this way:

```
...
BEGIN
ENABLE Timeout => BEGIN ... RETRY END;
...
OpenConnection[server];
END;
```

then the **RETRY** would jump to the beginning of the outermost **BEGIN-END** block.

CONTINUE is similar to **RETRY**, except that the jump is to the statement following the one that contains the catch phrase, or for an **ENABLE** clause, the statement following the **BEGIN-END** block surrounding the clause. **CONTINUE** is used when the catch phrase determines that it is desirable to skip the signal-raising statement rather than retry it.

8.2.4 Exit, loop and goto

The Mesa statements EXIT, LOOP, and GOTO can be used within a catch phrase just as they are used in BEGIN-END blocks and loops. These statements are legal within a catch phrase whenever the catch phrase is enclosed within a loop or BEGIN-END block in which they would normally be legal.

As an example, consider a program fragment that reads data from a file and inserts it into a linked list in sorted order. (We use the system interface **Stream**, discussed later in the course, to read the file. **Stream** raises the signal **Stream.EndOfStream** at end of file.)

```
DIRECTORY
   Heap USING [Create, Delete],
   MStream USING [Handle, . . .],
   Stream USING [EndOfStream, GetWord, . . .],
. . .;

ExitExample: PROGRAM
   IMPORTS Heap, MStream, Stream, . . . =

BEGIN
--TYPES
Node: TYPE = RECORD[
   data: CARDINAL ← 0,
   nextNode: PtrToNode ← NIL];
PtrToNode: TYPE = LONG POINTER TO Node;
PtrToPtrToNode: TYPE = LONG POINTER TO PtrToNode;

--Variables
z: UNCOUNTED ZONE ← NIL;
headOfList: PtrToNode ← NIL;

--Heap allocation / deallocation procedures
CreateStorageArea: PROCEDURE = BEGIN z ← Heap.Create[initial: 20]; END;

DestroyStorageArea: PROCECURE = {. . .};

MakeNode: PROCEDURE[nextNode: PtrToNode]
   RETURNS[nodePtr: PtrToNode] = {. . .};

FreeOneNode: PROCEDURE[freeThisNode: PtrToPtrToNode]
   RETURNS[nodePtr: PtrToNode] = {. . .};

FreeAllNodes: PROCEDURE =
BEGIN
   tempNodePtr: PtrToNode ← headOfList;
   UNTIL tempNodePtr = NIL DO
      tempNodePtr ← FreeOneNode[@tempNodePtr];
   ENDLOOP;
   headOfList ← NIL;
END;
```

```
--File Management Procedures
OpenDataFile: PROCEDURE [fileName: LONG STRING]
  RETURNS[sh: MStream.Handle] = {...};

CloseDataFile: PROCEDURE[sh: MStream.Handle]
  RETURNS[default: MStream.Handle ← NIL] = {...};

GetNextData: PROCEDURE[sh: MStream.Handle]
  RETURNS[n: CARDINAL] =
  BEGIN
    RETURN[Stream.GetWord[sh]]; --raises Stream.EndOfStream
  END;                          -- at "end of file"

--Linked List Management
ProcessData: PROCEDURE =
  BEGIN
    insertHere: PtrToPtrToNode ← NIL;
    sh: MStream.Handle ← OpenDataFile[MyFile];
    n: CARDINAL ← 0;
    DO
      n ← GetNextData[sh! Stream.EndOfStream = > EXIT];
      insertHere ← SearchLinkedList[n];
      InsertNode[insertHere, n];
    ENDLOOP;
    sh ← CloseDataFile[sh];
  END;

SearchLinkedList: PROCEDURE[n: CARDINAL]
  RETURNS [ insertionPoint: PtrToPtrToNode] = {...};

InsertNode: PROCEDURE[insertionPoint: PtrToPtrToNode, n: CARDINAL] = {...};

...
END.
```

The loop in **ProcessData** gets the next data item from the file, searches the list to see where it belongs and inserts it. Execution of the loop ends at the end of the file. The procedure **Stream.GetWord**, which is called in **GetNextData**, raises the signal **Stream.EndOfStream** when there is no more data to be transferred. The signal is caught in the call to **GetNextData** in **ProcessData**. The loop is then EXITed and control is transfered to

```
sh ← CloseDataFile[sh];
```

which closes the file before returning.

8.2.5 Unwind

A **GOTO**, **EXIT**, **RETRY**, **LOOP** or **CONTINUE** statement can cause a jump out of a catch phrase into the surrounding code. When a jump of this sort occurs, there may be several procedure calls on the stack below the target of the jump that will be prematurely exited when the jump is accomplished. (The signal was necessarily raised by the procedure on the bottom of the call stack, so neither that procedure nor any of the procedures between it and the procedure with the catch phrase will be completed when the jump is executed.) Since these

procedures may have been in the midst of doing something when the signal was raised, Mesa provides a facility for them to wrap up any unfinished operations.

Before executing the jump, the Signaller raises a special signal called **UNWIND** to tell all catch phrases that had previously rejected the signal that they are about to be removed. **UNWIND** propagates along the same path as the original signal: from the **BEGIN-END** block in which the original signal was raised to the **BEGIN-END** block containing the catch phrase executing the jump. It is the responsibility of each of these blocks to catch **UNWIND** and clean up its operations. The Signaller stops **UNWIND** when it reaches the catch phrase that is making the jump. The jump is then executed and control returns to the program.

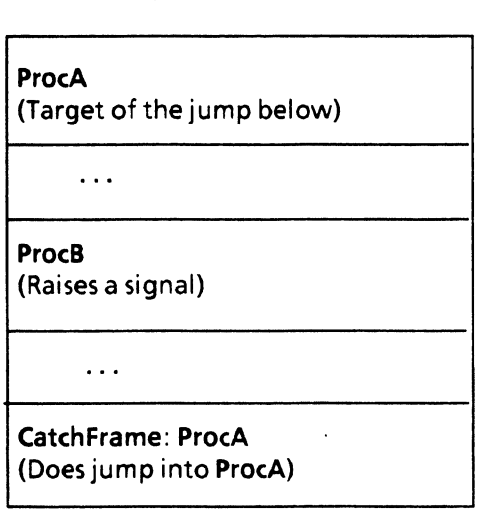


Figure 8.4

In Figure 8.4, **ProcB** has raised a signal which was caught by a catch phrase in **ProcA**. When that catch phrase does a jump, all the procedures below **ProcA** will be removed from the call stack and all **BEGIN-END** blocks within **ProcA** below the target of the jump will be exited. All of the catch phrases more deeply nested than the one executing have (necessarily) rejected the signal, so **UNWIND** propagates through this set of catch phrases. Because **UNWIND** stops after going through the catch phrases that rejected the original signal, it never results in an uncaught signal.

When doing a **GOTO**, **EXIT**, **RETRY**, **LOOP** or **CONTINUE** from a catch phrase, you must be aware that the **UNWIND** signal is going to be raised and that you need to clean up any work in progress in the procedures and **BEGIN-END** blocks lower on the call stack. If you forget, your programs may have space leaks from storage that should have been deallocated, or they may develop strange bugs from things such as files that should have been closed.

As an example, let's modify the previous fragment to allow the user to cancel the operation of inserting data from MyFile into the linked list. If the user hits the **ABORT** key (detected

by the call to the system interface **UserInput**) then the file transfer and insertion operation will be terminated.

```
DIRECTORY
  ...
UserInput USING [UserAbort],
FormSW USING[ProcType, ...],
Put USING[ Line, ...],
  ...;

UnwindExample: PROGRAM
  IMPORTS Heap, MStream, Stream, UserInput, ... =

BEGIN
  ...
--Signal declaration
UserAbort: ERROR = CODE;
  ...

CheckForAbort: FormSW.ProcType =
--Later chapters discuss sending text to a tool message subwindow
  BEGIN
  ENABLE
    UserAbort = > BEGIN GOTO abort; END;
  Put.Line[PtrToSomeToolsDataStructure.msgSW, "Processing File "L];
  ProcessData[];
  Put.Line[PtrToSomeToolsDataStructure.msgSW, " ... done" L];
  EXITS
    abort = >Put.Line[PtrToSomeToolsDataStructure.msgSW, " ...aborted" L];
  END;

ProcessData: PROCEDURE =
  BEGIN
    insertHere: PtrToPtrToNode ← NIL;
    sh: MStream.Handle ← OpenDataFile[MyFile];
    n: CARDINAL ← 0;
    BEGIN
    ENABLE
      UNWIND = >
        BEGIN
            IF sh # NIL THEN sh ← CloseDataFile[sh];
            IF headOfList # NIL THEN FreeAllNodes;
        END;
    DO
      IF UserInput.UserAbort[PtrToInputWindow] THEN ERROR UserAbort;
      --If the user has pressed the abort key raise the global signal UserAbort
      n ← GetNextData[sh! Stream.EndOfStream = > EXIT];
      insertHere ← SearchLinkedList[n];
      InsertNode[insertHere, n];
    ENDLOOP;
    sh ← CloseDataFile[sh];
    END;
  END;
```

```
...
--mainline code
...
CheckForAbort;
...
```

On each pass through the DO loop of **ProcessData**, we check to see if the user has hit the ABORT key. If so, the error **UserAbort** is raised. (See the Style section for a discussion of when to use ERROR and when to use SIGNAL.)

We catch the signal and print a message to the user that the action has been aborted. Since this signal has been declared as an ERROR, the catch phrase cannot RESUME. It must remove **ProcessData** from the stack, but at this point **ProcessData** has an open file and a linked list filled with nodes allocated from a heap. By providing a catch phrase for UNWIND in **ProcessData**, we get the chance to deallocate the nodes in the linked list and close the file before the procedure is removed. (See the Style section for a discussion on why the ENABLE clause is in an embedded BEGIN-END block.)

Note: It is common to recognize an exception condition (either by boolean checking or by catching a signal), and then raise a signal to pass this information on to a higher level procedure. This is often done to hide the lower level's implementation from the higher level's implementation. When debugging an uncaught signal, it is important to remember to check on the call stack for nested signals. For example, the apparent signal may have been raised in a catch phrase for some other signal. The root of the problem may be more apparent from the original signal than the one being debugged.

8.3 Summary

Signals and errors are an alternative to status polling. They are best at handling rare events, since raising a signal requires fewer checks than status polling within a loop, but processing a signal (with the Signaller) takes more time than processing a boolean statement. Using signals also helps the reader of a program to see which exceptions are being handled and to identify the code that handles them.

Though raising a signal is similar to calling a procedure, there are several differences:

- The code for a signal is dynamically bound to the signal at run-time, whereas the code for procedures is specified at compile-time.
- Normal execution halts during the processing of a signal, and the Signaller takes control.
- Execution can proceed at several places after a signal is raised, whereas after a procedure call execution must proceed after the statement that made the call.

The code for processing a signal is contained in a catch phrase. Catch phrases can occur either after an ENABLE, or after an ! in a procedure call. Catch phrases after an ENABLE can catch signals from any procedure calls nested within the BEGIN-END block, but catch phrases in procedure calls can only catch signals nested within that procedure call.

When the Signaller takes control, it does the following:

1. Looks up the call stack for a catch phrase that recognizes the signal, starting with the **BEGIN-END** blocks in the code that raised the signal.
2. Executes any catch phrases found for the signal, branching as indicated in the catch phrase. If no jump is indicated, it continues looking up the call stack.
3. If it can't find a catch phrase in any of the procedures on the call stack, the signal is uncaught, and the debugger is called via the special signal **UncaughtSignal**.

There are several ways to tell the Signaller how to continue execution after a catch phrase. You can use the Mesa statements **GOTO**, **EXIT**, or **LOOP**, with their normal effects. There are also several signal-specific jump statements. Doing a **RESUME** is similar to returning from a procedure call: control returns to the statement that raised the signal. However, you cannot **RESUME** an error. (This is the only difference between signals and errors.) **CONTINUE** causes execution to be transferred to the first statement after the one containing the the catch phrase. **RETRY** retries the statement that contains the catch phrase. (If the catch phrase is in an **ENABLE** clause, then the "containing statement" means the **BEGIN-END** block that contains the **ENABLE**.) **REJECT** tells the Signaller to continue looking up the call stack for another catch phrase that recognizes the signal. If you don't specify any jump statement the catch phrase performs an implicit reject.

GOTO, **EXIT**, **LOOP**, **CONTINUE**, and **RETRY** each cause a jump into the procedure containing the catch phrase. This means that the procedure and **BEGIN-END** blocks below it will be removed from the call stack. The Signaller generates the special signal **UNWIND** to allow catch phrases that have previously rejected the signal to do clean up, such as closing files and deallocating storage.

8.4 Style

8.4.1 Scope

The scope of an **ENABLE** clause places it outside the scope of variables declared in the same **BEGIN-END** block, since the **ENABLE** clause must precede any declarations. (See page 8.5 of the *Mesa Language Manual* for a diagram of clause scopes.) To permit the catch phrase in the **ENABLE** clause to have access to local variables, the **ENABLE** clause must be more deeply nested than the local variables. To accomplish this, declare the **ENABLE** clause and the executable statements within an extra **BEGIN-END** block. The **ENABLE** clause will then know about the variables since they are declared in a surrounding block:

```
BEGIN
  Declarations
  BEGIN
    ENABLE
    Statements
  END
END
```

8.4.2 Errors vs. signals

An **ERROR** is used instead of a signal when a **RESUME** cannot be handled, since it is illegal to **RESUME** an **ERROR**. You don't want a catch phrase to do a **RESUME** if you do not want to return to the procedure that generated the **ERROR**, either because it would be inappropriate, or

because something catastrophic has happened. In the program **UnwindExample**, we used the ERROR **UserAbort**. We made **UserAbort** an ERROR since the user wants the procedure to stop. This is a case where it would be inappropriate to resume execution.

8.4.3 A caution

In the RESUME example in §8.2.2, the catch phrase returned a pointer for use by the RESUMEd procedure. If some intermediate procedure held the value of the old pointer it would not have been informed of the new value, and presumably an error situation would arise when control returned to it. When you code a catch phrase to replace a node out from under a pointer, make sure that any code that used the old node will use the revised pointer.

8.5 Questions

1) In the following code fragment, to which statement will the CONTINUE branch?

```
commands ← 0;
BEGIN
ENABLE
  AlreadyDone => CONTINUE;
  GetToken[token];
  DoCommand[token]; -- where AlreadyDone would get raised
  commands ← commands + 1;
  ResetStatus[];
END
Write["Commands completed."L];
```

In the following code fragments, list the order that the statements labeled <statement n> will be executed.

2)

```
Sig1: SIGNAL = CODE;
x: CARDINAL ← 0;
...
FOR counter: INTEGER IN [1..3] DO
  ENABLE
    Sig1 => RETRY;
  <statement 1>
  IF counter = 2 THEN
    BEGIN
    ENABLE
      BEGIN
      Sig1 => <statement 2>;
      UNWIND => x ← 1;
      END;
    <statement 3>;
    IF x = 0 THEN
      SIGNAL Sig1;
    <statement 4>;
    END;
  <statement 5>
ENDLOOP; ...
```

3)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
      BEGIN
      ENABLE
        Sig1 = > LOOP;
      <statement 1>;
      IF counter = 1 THEN
        SIGNAL Sig1;
      <statement 2>;
      END;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

4)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
      BEGIN
      ENABLE
        Sig1 = > CONTINUE;
      <statement 1>;
      IF counter = 1 THEN
        SIGNAL Sig1;
      <statement 2>;
      END;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

5)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
      BEGIN
      ENABLE
        Sig1 = > EXIT;
      <statement 1>;
      IF counter = 1 THEN
        SIGNAL Sig1;
      <statement 2>;
      END;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

6)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
   ENABLE
      Sig1 = > LOOP;
   <statement 1>;
   IF counter = 1 THEN
      SIGNAL Sig1;
   <statement 2>;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

7)

```
Sig1: SIGNAL = CODE;
...
FOR counter : INTEGER IN [1..2] DO
   ENABLE
      Sig1 = > CONTINUE;
   <statement 1>;
   IF counter = 1 THEN
      SIGNAL Sig1;
   <statement 2>;
   <statement 3>;
ENDLOOP;
<statement 4>;
...
```

8)

```
Sig1: SIGNAL = CODE;
...
Proc1: PROCEDURE =
   BEGIN
      SIGNAL Sig1;
   END;
...
IF TRUE THEN
   BEGIN
   ENABLE
      Sig1 = > RESUME;
   <statement 1>;
   Proc1[!Sig1 = > CONTINUE];
   <statement 2>;
   Proc1;
   <statement 3>;
   END;
<statement 4>;
```

9)

```
Sig1: SIGNAL = CODE;
...
BEGIN
ENABLE
  Sig1 = > RESUME;
<statement 1>;
IF TRUE THEN
  BEGIN
  ENABLE
    Sig1 = > GOTO TheEnd:
  <statement 2>;
  SIGNAL Sig1;
  <statement 3>;
  EXITS
    TheEnd = > <statement 4>;
<statement 5>;
EXITS
  TheEnd = > <statement 6>;
END;
...
```

10) In the following pseudo-Mesa code, what happens when the call **Proc1[0]** is made? (Assume that catch-cases 4 and 7 reject **Sig1**.) Which catch-cases are executed, and in what order?

```
Proc1: PROC [x: CARDINAL] =
  BEGIN -- block A
  ENABLE { -- Catch phrase-1
    Sig1 = > GOTO punt; -- Catch-case-1
    Sig2 = > <Catch-case-2>;
    UNWIND = > <Catch-case-3>};
  Stmt1;
  Stmt2;
    BEGIN -- block B
    ENABLE -- Catch phrase-2
      Sig1 = > <Catch-case-4>;
    Stmt3;
    Stmt4;
    OtherProc[x ! -- Catch phrase-3
      Sig2 = > <Catch-case-5>;
      UNWIND = > <Catch-case-6>];
    END; -- block B, and scope of Catch phrase-2
  Stmt5;
  EXITS
    punt = > Stmt6;
  END; -- Proc1, and scope of Catch phrase-1
...
OtherProc: PROC [x: CARDINAL] = {stillOtherProc[x ! -- Catch phrase-4
    Sig1 = > <Catch-case-7>;
    Sig2 = > <Catch-case-8>;
    UNWIND = > <Catch-case-9>]};
```

```
StillOtherProc: PROC [x: CARDINAL] = {
  IF x = 0 THEN ERROR Sig1 ELSE ERROR Sig2};
```

11) In the program below, what value does b get?

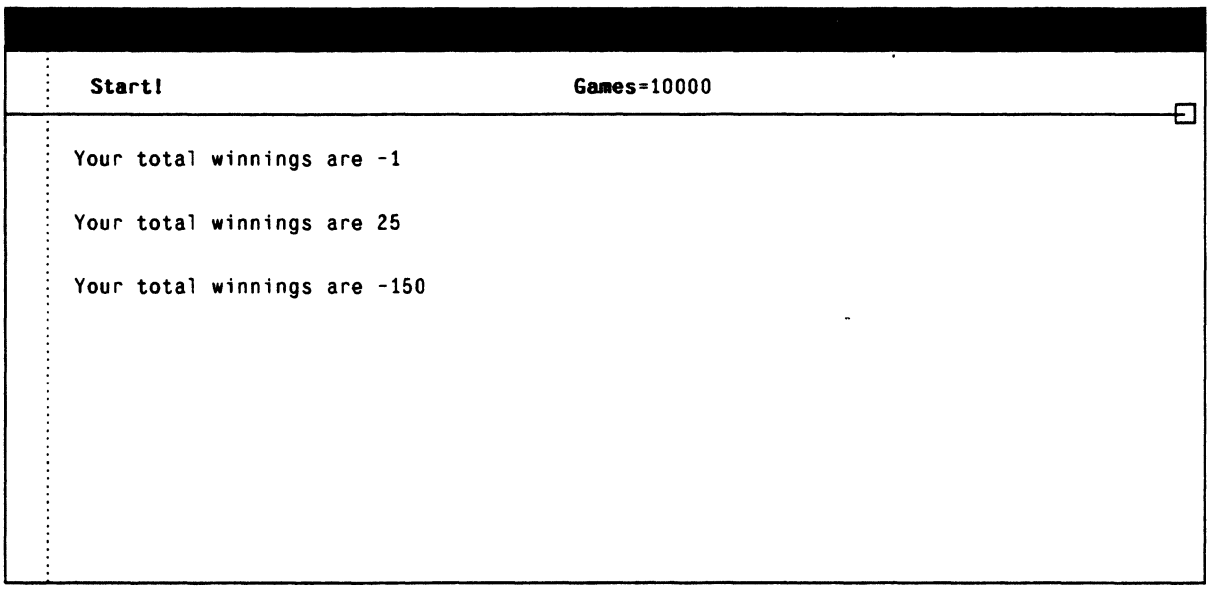
```
Question3: PROGRAM =
  BEGIN
  Sig: SIGNAL [c1: CARDINAL] RETURNS [c2: CARDINAL] = CODE;
  Proc: PROCEDURE [c1, c2: CARD] RETURNS [BOOLEAN] =
    BEGIN
      ENABLE Sig => {c2 ← c1; RESUME};
      If c2 # c1 THEN c2 ← SIGNAL Sig[c2];
      RETURN [c1 = c2]
    END;
  c1, c2: CARDINAL;
  b: BOOLEAN;

  --Mainline code
  b ← Proc[1,2];

  END.
```

8.6 Exercise

In this programming assignment, you will alter a program that has been written to play the game of blackjack. The user initially specifies the number of games the program will play with itself. There will only be 2 players in the game: the dealer and the player. When the user clicks **Start!**, the program will play out all of the games; the player's winnings will be output to a file sub-window when all of the games are finished:



In this game of blackjack, the player bets 1 dollar on every hand. If he gets blackjack (a total of 21 in exactly two cards), then he wins 2 dollars. If the dealer gets blackjack, the player loses. If the game continues, the player receives hits (additional cards) according a conservative strategy based on his hand, and the dealer's face card. If he busts (exceeds 21), he loses. Otherwise, the dealer receives hits until his total is a hard 17 (a hand in which an ace is counted as 1 rather than 11) or above. If the dealer busts, the player wins 1 dollar. Finally, if the game has reached this stage, the 2 hands are compared. The players wins 1 dollar if his hand is greater; his winnings remain the same if the hands tie; and he loses if the dealer's hand is greater. There is no double-down, splitting, or insurance in this version of blackjack.

When the user invokes **Start!**, the following procedure in the implementation module is called:

```
PlayBlackJack: PUBLIC PROCEDURE[output: Window.Handle ← NIL, gamesToBePlayed:
  CARDINAL ← 0] =
 --This procedure will play Blackjack as many times as specified in gamesToBePlayed.
 --After the games have been played, results are written out to the window handle
 --output.
BEGIN
 playerTotal: CARDINAL;
 dealerTotal: CARDINAL;
 playerHasAce: BOOLEAN;
 dealerHasAce: BOOLEAN;
 dealerHole: CardType;
 dealerFace: CardType;
 winnings: INTEGER ← 0;

 THROUGH [1..gamesToBePlayed] DO
   IntializeDeckForNewGame;
   [playerTotal,dealerTotal,playerHasAce,dealerHasAce,dealerHole,dealerFace] ←
     Deal[];
   IF playerHasAce AND (playerTotal = 11) THEN
    BEGIN
     winnings ← winnings + 2; --Player has Blackjack
     LOOP;
     END;
   IF dealerHasAce AND (dealerTotal = 11) THEN
    BEGIN
     winnings ← winnings - 1; --Dealer has Blackjack
     LOOP;
     END;
  [playerTotal] ← HitPlayer[playerHasAce, playerTotal, dealerFace];
   IF playerTotal > 21 THEN
     BEGIN
     winnings ← winnings - 1; --Player busted
     LOOP;
     END;
   dealerTotal ← HitDealer[dealerHasAce, dealerTotal];
   IF dealerTotal > 21 THEN
     BEGIN
     winnings ← winnings + 1; --Dealer busted
     LOOP;
     END;
   SELECT playerTotal FROM
     < dealerTotal = > winnings ← winnings - 1;
     > dealerTotal = > winnings ← winnings + 1;
     ENDCASE = >    NULL; -- Push
  ENDLOOP;
 Put.CR[output];
 Put.Text[output,"Your total winnings are "L];
 Put.LongDecimal[output, winnings];
 Put.CR[output];
END;
```

The procedures **Deal**, **HitPlayer**, and **HitDealer** all call the following procedure when they need a card:

```
NewCard: PROCEDURE RETURNS [card: CardType] =
  --This procedure returns the next card in the deck. If at any point, the last card in
  -- the deck is used, the non-used cards in the deck are shuffled, and play continues
  --where it left off
  BEGIN
  IF freeCard = 53 THEN
    [deck, firstCard, freeCard] ← Shuffled[deck, firstCard];
  card ← deck[freeCard];
  freeCard ← freeCard + 1;
  RETURN;
  END;
```

In the procedure **NewCard**, **deck** is an array of 52 records with each record representing one card. Dealing is accomplished by stepping through the deck one card at a time. At any point during a game of blackjack, **firstCard** is an index indicating the first card that was dealt for that hand. **freeCard** is an index indicating the top card on the remaining deck (the next card to be dealt). Thus, when **freeCard** is 53, **deck**, **firstCard**, and **freeCard** are reinitialized by calling the procedure **Shuffled,** which makes sure that the cards on the table are not included in the shuffle. To complete this assignment, you don't have to know how **Shuffled** works, just that it does the right thing when passed the right arguments.

Currently, if the dealer runs out of cards at any point in the game, the cards are in use are shuffled, and the game continues where it left off. So if only 1 card remains in the deck, that card will be dealt, the rest of the deck will be shuffled, and the dealing will continue.

Modify this program (using a signal) so that if the dealer runs out of cards while dealing the initial hand (the first 4 cards), that game is started over with a shuffled full deck of 52 cards. If the dealer runs out of cards while hitting the player, the unused cards in the deck should be shuffled, and the game continued where it had paused (as before). If the dealer runs out of cards while hitting himself, then the dealer loses the game and the next game is started with a shuffled full deck of 52 cards. The file that you will be altering is **BlackjackImpl.mesa**. Other files you will need are **BlackjackDefs.mesa**, **BlackjackControl. mesa**, and **Blackjack.config**. Once you have the new version of **BlackjackImpl.mesa,** answer the following questions:

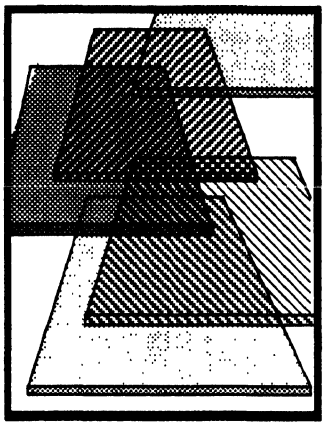
1. Briefly describe how you could have completed the the assignment without using a signal.
2. Signals could have been used to indicate **DealerBlackjack**, **DealerBusted**,... From an efficiency point of view, why isn't this a good idea?

8.7 References

Chapter 8 of the *Mesa Language Manual* describes the syntax of signals and some reasons for using them.

Section 4 of *Mesa: A Designer's User Perspective* gives some background information on signals.

9



Variant records

Programmers often find it convenient to aggregate information of different types. For example, suppose you want a data base of statistics for individual softball players. For each player, you want to know things like name (**LONG STRING**), position (enumerated **TYPE**), times at bat (**INTEGER**), hits (**INTEGER**), etc. When the information is the same for all players, you can use the Mesa **RECORD** type to group the data for each player. However, some players have additional pieces of information that are relevant only to the position they play. For example, if a player is a pitcher, you want to keep track of the number of walks given up, and the number of strikeouts pitched, in addition to the common information that you keep track of for all players. Or, if a player is an infielder, you might want to know the number of errors committed. In cases where members of a class have information that is relevant only to their subclass, you should use the variant **RECORD** construct.

In this chapter, we discuss how to declare variant **RECORD** types, how to declare, allocate and initialize variant **RECORD** variables, how to use constructors to assign values to variant **RECORD**s, and how to access the fields of variant **RECORD**s.

9.1 Definition of terms

adjective — An *adjective* is an identifier constant from an enumerated **TYPE** used to select one of the alternatives in a variant **RECORD** template.

tag — The *tag* is a field of a variant **RECORD**; tag is used to select one of the alternative "arms" of the variant part by matching one of the adjectives.

discrimination — A *discrimination* statement provides access to the fields in the variant part of a variant **RECORD** variable, based on the value of the tag.

9.2 Discussion

9.2.1 Declaring variant RECORDs

There are basically two parts to declaring a record variable. Step one is to declare a **TYPE** that provides a "template" – that is, the **TYPE** declaration shows all the fields that a variable of that **TYPE** will have. Step two is to declare variables of the newly defined **RECORD**

type. Variant RECORDs are done the same way. The only difference is that the TYPE declaration must show the fields for all possible alternative variants of the TYPE.

It is worth taking some time to study the syntax of variant RECORDs to make your use of them less error-prone. We declare the TYPE as follows:

```
identifier: TYPE = RecordTC
```

The syntax for **RecordTC** is shown in Fig. 9.1. Refer to it as you read this discussion.

```
RecordTC            ::= MachineDependent RECORD [VariantFieldList]
MachineDependent    ::= empty | MACHINE DEPENDENT

VariantFieldList    ::= CommonPart identifier : Access VariantPart |
                        VariantPart |
                        NamedFieldList |
                        UnnamedFieldList

CommonPart          ::= empty |
                        NamedFieldList ,

VariantPart         ::= SELECT Tag FROM
                        VariantList
                        ENDCASE

Access              ::= empty |
                        PUBLIC |
                        PRIVATE

Tag                 ::= identifier : Access TagType |
                        COMPUTED TagType |
                        OVERLAID TagType

TagType             ::= TypeSpecification | *
VariantList         ::= Variant | VariantList Variant
Variant             ::= IdList ⇒ [VariantFieldList] , |
                        IdList ⇒ []

    ...
NamedFieldList      ::= IdList : Access TypeSpecification DefaultOption |
                        NamedFieldList, idList: Access TypeSpecification
                        DefaultOption
```

Figure 9.1 RecordTC Syntax

Obviously, the syntax presents a lot of possibilities for declaring a variant RECORD type. The main things to notice are the syntax for the variant field list, for the variant part and

for the tag within the variant part. If a RECORD has a common part and a variant part, there will be an identifier for the variant part and a second identifier for the tag.

Let's look at a simple example. There is a variant RECORD type declared in the program **SoftballDataTool**. (You should retrieve the files `SoftballDataTool.mesa` and `SoftballDataTool.bcd` from the course directory, if you don't already have them on your local disk.) This program is designed to solve the problem of keeping track of information for people on a softball team. Let's look first at the TYPE declarations.

The declaration for **SoftballPlayerData** is a variant RECORD:

```
SoftballPlayerData: TYPE = RECORD[
  name: LONG STRING ← NIL,
  timesAtBat: INTEGER ← 0,
  hits: INTEGER ← 0,
  otherInfo: SELECT position: Position FROM
    outfielder = > [
      bestPosition: OutfieldPosition,
      errors: INTEGER ← 0],
    infielder = > [
      bestPosition: InfieldPosition,
      doublePlays: INTEGER ← 0,
      errors: INTEGER ← 0],
    pitcher = > [strikeouts, walks: INTEGER ← 0],
    catcher = > [],
  ENDCASE];
```

The fields in the common part include **name**, **timesAtBat** and **hits**. We want these three pieces of information about every player. Notice that the syntax requires that you declare all fields of the common part before you declare the variant part. The identifier for the variant part, **otherInfo**, comes just after the fields for the common part.

Each player has a **position**, which is the tag identifier. The TYPE of this field is enumerated: **Position: TYPE = {outfielder, infielder, pitcher, catcher};**. The constants of the enumerated TYPE are used as adjectives in the variant part of the variant RECORD. In our example, the value of **position** for any given player may be either **outfielder**, **infielder**, **pitcher**, or **catcher**. The remaining fields in the RECORD representing any individual player will depend on the value in the tag field. If a player's **position** is **outfielder**, for example, the RECORD representing that player will have two fields (**bestPosition** and **errors**) in addition to the fields in the common part of the RECORD. So, a RECORD representing an **outfielder** has a total of five fields, while the RECORD of an **infielder** has a total of six fields. Notice that a **catcher**'s RECORD only has three fields, because

```
catcher ⇒ []
```

is the way to express the fact that this variant has no additional fields.

This is a relatively simple example. The syntax for RECORD types provides many possibilities, such as bound variant types, implicit tags and computed tags.

9.2.2 Allocation of variant RECORDs

Now that we have declared a variant RECORD type, we can declare variables of that TYPE. You declare and initialize variant RECORD variables in the usual way. For example, notice

```
noPlayer: SoftballPlayerData ← [NIL, 0, 0, catcher[]];
```

in `SoftballDataTool.mesa`. This is the declaration and initialization of a variant RECORD variable. You may be wondering how the Compiler can allocate space for a variable whose size may change during the course of execution of the program; after all, we may assign some other variant to **noPlayer** at some point. The answer is that when a variable is declared to be of TYPE **SoftballPlayerData**, the Compiler allocates enough space for the largest variant.

This program also illustrates allocation from a heap. Instead, the space for the **dataSeq** is dynamically allocated from the system heap by the following statement:

```
IF dataPtr = NIL THEN
    dataPtr ← Heap.systemZone.NEW[Data[numberOfPlayers]];
```

in the procedure **ClientTransition**. Here the run-time system allocates enough space for each member of the sequence to hold the largest possible variant.

9.2.3 Initialization of and assignment to variant RECORD variables

Variant RECORDs are initialized and assigned values like regular RECORDs, except that you must supply appropriate information about the variant part. Here's a helpful way to look at variant record initialization: the variant part is another, embedded record, whose type is determined by the tag, and the syntax for constructing this embedded record is exactly the same as for a regular record.

The RECORD constructor that you use to initialize a variant RECORD variable must specify a value for the tag field, and values for the appropriate fields for that variant. In the above example, the value **catcher** is assigned to the tag field of **noPlayer**. Recall that the **catcher** variant had no additional fields, so no additional values are given in the above constructor. We see other examples of initialization of variant RECORD variables in the procedure **InitDataBase**. For example

```
dataPtr[0] ← [String.CopyToNewString[s: "Ralph"L, z: Heap.systemZone],
  140, 128, pitcher[133, 1]];
```

assigns **"Ralph"** to the **name** field, 140 to the **timesAtBat** field, and 128 to the **hits** field of the RECORD. The **position** field is assigned the value **pitcher**, 133 is assigned to the **strikeouts** field in the variant part, and 1 is assigned to the **walks** field of the variant part of the RECORD.

An alternate way of stating this assignment is:

```
dataPtr[0] ← SoftballPlayerData[
  name: String.copyToNewString[s: "Ralph"L, z: Heap.systemZone],
  timesAtBat: 140,
  hits: 128,
  otherInfo: pitcher[
    strikeOuts: 133,
    walks: 1]];
```

9.2.4 Accessing the fields of a variant RECORD variable

Finally, now that we have declared a variant RECORD type and variant RECORD variables, we are ready to use these variables. A typical situation is when a procedure accepts a

parameter that is of some variant RECORD type, and processes the information contained in the RECORD variable. For example, take a look at the procedure **DisplayData**. This procedure displays the information about each player in the data base in the tool's message subwindow. Notice that it expects a parameter of TYPE **SoftballPlayerData**.

The "discrimination statement" solves the problem of making sure the procedure knows which variant it is dealing with. The common fields of the actual parameter can be accessed normally, but the fields in the variant part can be accessed *only* inside the discrimination statement, which is

```
WITH player: playerData SELECT FROM
  outfielder ⇒ {...};
  infielder ⇒ {...};
  pitcher ⇒ {...};
ENDCASE;
```

Notice how the structure of the discrimination statement mirrors the structure of the TYPE declaration of **SoftballPlayerData**.

Inside the discrimination statement, an "alternate name" is given to the actual parameter by

```
WITH player: playerData SELECT FROM
```

The fields of the variant part of **player** (but not **playerData**) become accessible inside whichever arm is selected, based on the value in the tag of **playerData**. This construct allows the compiler to detect any attempt to access an "incorrect" field within a given arm. For example, if you write

```
Put.Decimal[toolData.msgSW, player.strikeouts];
```

inside the **outfielder** arm of this discrimination statement, the compiler will tell you that "strikeouts is not valid as a field selector. . . ." This prevents you from trying to access a field in an incorrect variant at run time.

Since the discrimination statement relies on the value in the tag field of the RECORD, suppose you just change that value in the tag field. That is, what if you add

```
playerData.position ← pitcher
```

as the first statement in **DisplayData**? Would the discrimination statement always select the **pitcher** arm of the discrimination statement, and try to use the value **strikeouts** for every kind of player? No, Mesa won't allow you to selectively access the tag field of a variant RECORD. In fact, if you try to write the above statement, the Compiler will tell you that "playerData.position cannot be updated. . . ." The only way you can change the variant tag is to assign a new value to the entire variant part using a constructor for that variant part. Variant RECORDs in Mesa are type-safe.

9.3 Summary

This chapter introduced the fundamentals of variant RECORDs. One important feature of Mesa's variant records is that they are type-safe. You can depend on the discrimination statement, in concert with the syntax, to prevent errors associated with accessing the fields in the variant parts of RECORDs.

Several topics related to variant RECORDs that we did not discuss include "bound" variant types, and "implicit" and "computed" tags. The built-in predicate ISTYPE, and the built-in operator NARROW are also available to assist you in your use of variant RECORDs. These features, along with a variation of the discrimination statement that is more efficient in certain cases than the one we looked at, are described in the *Mesa Language Manual*.

9.4 References

Section 6.4 of the *Mesa Language Manual* discusses variant RECORDs, including declaring variant RECORD types and variables, giving values to variant RECORD variables, and accessing the fields of variant RECORDs. This section also discusses several other points regarding particular uses of variant RECORDs that we did not discuss in this chapter.

9.5 Exercises

Modify the **SoftballDataTool** (used as an example in this chapter) to include the following information:

If a player is an infielder, has he been traded ?

If he has been traded:

-- how many times has he been traded ?

-- in what year was he last traded ?

If he has NOT been traded:

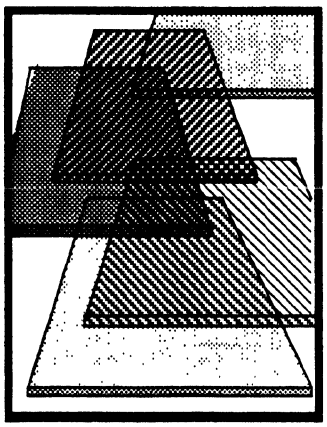
-- how many years has he played for the team ?

-- is he likely to be traded this season ?

You should include this information in a variant section, which is enclosed by the infielder section. Thus, you will create a variant within a variant record. You will have to add this new information for any infielders already existing in the database. Assume that existing infielders have never been traded.

Once you have added the new variant section, a new player will be joining the team. His name is Larry, he is an infielder who plays third base, and he has been traded 3 times, the last time in 1983. You will have to increase the **numberOfPlayers** in order to add him to the database, and print out his statistics along with those of the rest of the team. Obviously, you will also have to change the output routines to dispaly the new information.

10



Concurrency

Mesa provides language support for concurrent execution of multiple processes, as well as *monitors* and *condition variables* to help synchronize such processes.

In this chapter, we discuss how to use the **FORK** and **JOIN** operators to create new processes and later resynchronize them. We also illustrate how to monitor access to a module's global variables, and how to use condition variables to accomplish more complex forms of synchronization. We do not discuss how to monitor data implemented by a multi-module abstraction, or data that is encapsulated in an object rather than in a module; you will have to consult the *Mesa Language Manual* for information on these topics.

10.1 Definition of terms

Asynchronous call — An *asynchronous call* is a procedure call that initiates an operation and then returns control to its caller without waiting for the operation to complete.

Background process — A *background process* is a process that receives machine resources only if higher priority processes are idle or blocked.

Condition variable — A *condition variable* is a Mesa construct by which processes wait for or provide notification of an event. A condition variable is associated with a monitor.

Critical section — A *critical section* is a portion of a program in which only one process may be executing at a time. In Mesa, access to critical sections is arbitrated by monitors.

Hint — A *hint* is information that is usually accurate and is easy for a program to use. A program can detect when a hint is inaccurate and find the truth in some other (usually less efficient) way.

Monitor — A *monitor* module is a Mesa module that controls access to shared data.

Monitor invariant — A *monitor invariant* is a logical assertion about the state of monitored data whenever the monitor is unlocked (i.e., exited). Every monitor has a monitor invariant.

Monitor lock — A *monitor lock* is essentially a hidden data item associated with each monitored record or program that indicates when a process has entered and not yet exited a critical section.

Process — A *process* is effectively a procedure activation that runs concurrently with its caller, allowing asychronous activities.

Synchronous call — A *synchronous call* is a procedure call that returns control only after the operation completes.

10.2 Discussion

Mesa casts the creation of a new process as a special procedure call. You create a new process by FORKing a procedure rather than simply calling it; the new process then runs concurrently with its caller. The new process has a different call stack, with the forked procedure as the root of the activation. Mesa allows any procedure (except an internal procedure of a monitor; see section 10.2.3.1) to be invoked in this way.

10.2.1 JOINing processes

Once you have created concurrent processes, there are various levels of synchronization possible, depending on the role that your forked process is to perform. For example, you might fork a process when you have a long computation to perform, and you would like to allow other processing to take place concurrently. When you create such a process, you later need to synchronize that process with its parent so that it can return the result of the computation. You can accomplish this synchronization with the JOIN operation. JOIN establishes a rendezvous point: the first process to reach the rendezvous is blocked until the other arrives. When both processes have arrived, the forked process returns its results and is then terminated.

To illustrate this, here is an example that iteratively reads a large buffer of data and processes it. A sequential implementation might look like this:

```
Control: PROCEDURE =
  BEGIN
  buffer: LONG POINTER TO Buffer ← zone.NEW[Buffer];
  DO
    ENABLE
      NoMore => EXIT;
    ReadBuffer[buffer];
    ProcessBuffer[buffer];
    ENDLOOP;
  zone.FREE[@buffer];
  END;
```

ReadBuffer collects input data in **buffer**, and then **ProcessBuffer** manipulates the data. The signal **NoMore** is raised when there is no more data, causing the DO loop to terminate.

A problem with this code is that you can not read a buffer of data while processing one, nor process a buffer of data while reading one. Since these operations are distinct, it would be useful (and more efficient) to read the next buffer of data while processing the previous one. This double buffering scheme might look like this:

```
Control: PROCEDURE =
  BEGIN
  Status: TYPE = {normal, end};
  readBuffer: LONG POINTER TO Buffer ← zone.NEW[Buffer];
  processBuffer: LONG POINTER TO Buffer ← zone.NEW[Buffer];
  status: Status ← normal;
  p: PROCESS RETURNS[status: Status];          --declare the process

  status ← ReadBuffer[readBuffer];
  WHILE status = normal DO
    SwapBuffers[readBuffer, processBuffer];
<<points readBuffer to the buffer that has just been processed and points
processBuffer to the buffer that has just been read>>
    p ← FORK ReadBuffer[readBuffer];
    ProcessBuffer[processBuffer];
    status ← JOIN p;
    ENDLOOP;
  zone.FREE[@readBuffer];
  zone.FREE[@processBuffer];
  END;
```

Control now allocates two buffers, one of which can be processed while the other is being filled with the next block of data. **Control** reads in an initial buffer of data and then loops until the reading process returns a state other than normal. During the loop, we swap buffers and then we fork **ReadBuffer**. Thus, we can fill the new buffer while we process the old one. At the end of the loop, we synchronize the two processes with the **JOIN** operator.

Some things to notice from this example:

- **FORK** 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**, you cannot discard the value of the **FORK** by writing an empty extractor. Thus **FORK ReadBuffer[readBuffer]** is assigned to **p**.

- The **JOIN** appears as either a statement or an expression, depending upon whether or not the process being joined returns anything. 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.

- There is no *intrinsic* rule against multiple activations (calls and/or forks) of the same procedure coexisting at once. Of course, it is possible to write procedures that will work incorrectly if used in this way, but the mechanism itself does not prohibit such use.

10.2.2 Detached processes

Not all processes follow the FORK/JOIN paradigm; there are others whose role is better cast as continuing provision of services, rather than one-time calculation of results. Such processes are called "detached", since they never need to be resynchronized with their caller. If the lifetime of a detached process is bounded at all, its deletion is a private matter, since it involves neither synchronization nor delivery of results.

Pilot provides the facilities for detaching processes. The **Process** interface, documented in section 2.4.1 of the *Pilot Programmer's Manual*, includes operations to check on the state of a process, to set process timeouts, to set process priorities, to abort processes, and to detach processes.

Process.**Detach** takes a process and detaches it from its creator. If you use this procedure to create a detached process, the **Process** interface will take care of deleting the process when it returns from its root procedure.

Consider a tool with one command, which takes a long time to process. Typically this command runs in the notifier and therefore prevents concurrent user interactions. To avoid this, you can FORK the command as a new detached process:

```
Command: FormSW.ProcType =
  BEGIN
  Process.Detach[FORK RealCommand];
  END;
```

10.2.3 Monitors

FORK/JOIN enables very simple synchronization: you can synchronize two process when a computation has been completed. However, you need a more general mechanism to allow processes to communicate while work is in progress. Specifically, the FORK/JOIN construct does not provide access control (mutual exclusion) to shared data. Thus, we coded the double buffering example to ensure that **ReadBuffer** and **ProcessBuffer** never shared a buffer by executing the pointer swap while only one process existed (and thus there could be no contention to the data).

To enable more sophisticated interaction, Mesa provides an interprocess synchronization mechanism that is a variant of monitors adapted from the work of Hoare, Brinch Hansen, and Dijkstra. The underlying view is that processes share little, but when they do, the interaction reduces to carefully synchronized access to shared data.

10.2.3.1 Mutual exclusion to shared data

A monitor is a module instance. It thus has its own global frame, and its own procedures for accessing this (global) data. Unlike normal PROGRAM module instances, however, a monitor module has an associated monitor lock, which guarantees that only one process at a time can access the data. (The lock can also be associated with the object being shared; see section 9.4.5 of the *Mesa Language Manual*).

Monitor modules are declared much like program or definitions modules; for example:

```
M: MONITOR [arguments] =
  BEGIN
  ...
  END.
```

A call into the monitor implicitly acquires the lock; returning from the monitor releases the lock. When a process attempts to enter a monitor and the lock is already held, it must wait until the current process finishes and releases the lock. The monitor lock thus ensures that only one process at a time can change the data, thereby guaranteeing the integrity of the monitor invariant. (A monitor invariant is an assertion defining what constitutes a "good state" of the data for that particular monitor.)

It is important to realize that the mutual exclusion takes place at the entry and exit points of a monitor. In Mesa, these entry/exit points are encapsulated in procedures called **ENTRY** procedures. The code within an **ENTRY** procedure is a critical section: a call to an **ENTRY** procedure acquires the monitor lock, a return from an **ENTRY** procedure releases the monitor lock. Entry procedures are declared as:

```
P: ENTRY PROCEDURE [arguments] RETURNS [results] = ...
```

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, which are common routines shared among the several entry procedures. These execute with the monitor lock held, and may thus freely access the monitor data as necessary. Internal procedures should be private, since direct calls to them from outside the monitor would bypass the acquisition of the lock. You can only call internal procedures from an entry procedure or another internal procedure. They are declared as follows:

```
Q: INTERNAL PROCEDURE [arguments] RETURNS [results] = ...
```

The attributes **ENTRY** or **INTERNAL** may be specified only on a procedure in a **MONITOR** module (or on an **INLINE** procedure in a definitions module).

Some monitor modules may also wish to have *external* procedures. These are declared as normal non-monitor procedures:

```
R: PROCEDURE [arguments] RETURNS [results] = ...
```

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. Since it is outside the monitor, an external procedure must *not* reference any monitor data nor call any internal procedures. The compiler checks for calls to internal procedures within external procedures, but does not check for accesses to monitor data.

Generally speaking, a chain of procedure calls involving a monitor module has the 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 readability 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.
```

To illustrate mutual exclusion using monitors, consider the case where many processes may be capable of inspecting, incrementing, and decrementing a counter of active and inactive windows of a multiple instance tool. The operation **Activate** decrements the inactive counter by one and increments the active counter. The **Deactivate** operation does the reverse. To ensure consistent data (i.e. the number of active windows plus the number of inactive windows equals the number of instantiated windows) the increment/decrement to the active and inactive counters must occur atomically. Otherwise, it would be possible for an **Inspect** operation to return a counter that has only been partially updated.

```
KeepCount: MONITOR =
BEGIN
CounterType: TYPE = RECORD[active: INTEGER, inactive: INTEGER];
counter: CounterType ← [0,0];
Activate: ENTRY PROCEDURE =
  BEGIN
  ENABLE UNWIND => NULL;  --see section 10.5.3 for a discussion of this statement
  counter.active ← counter.active + 1;
  counter.inactive ← counter.inactive - 1;
  END;
Deactivate: ENTRY PROCEDURE =
  BEGIN
  ENABLE UNWIND => NULL;  --see section 10.5.3 for a discussion of this statement
  counter.active ← counter.active - 1;
  counter.inactive ← counter.inactive + 1;
  END;
Inspect: ENTRY PROCEDURE RETURNS[counter: CounterType] =
  BEGIN
  ENABLE UNWIND => NULL;  --see section 10.5.3 for a discussion of this statement
  RETURN[counter];
  END;
END.
```

10.2.4 Synchronization with condition variables

In addition to providing mutual exclusion; monitors also allow a sophisticated form of synchronization. For example, a process may only want to execute monitored code if certain conditions hold. If the conditions hold, the process continues as usual. If a condition is not satisfied, however, the process blocks and releases its hold of the monitor lock. A new process can then enter the monitor, eventually make the condition true, and notify the blocked process that it may continue. This kind of synchronization is provided by *condition variables*.

Condition variables are declared as:

```
c: CONDITION;
```

All the fields of a condition variable are private to the process mechanism; you can only access a condition variable via the condition variable operations **WAIT**, **NOTIFY**, and **BROADCAST**.

WAIT condition blocks the current process and releases the monitor lock. Since a **WAIT** always releases the monitor lock while waiting, you must restore the monitor invariant (i.e., return the shared data to a "good state") before waiting.

NOTIFY condition wakes up one process waiting on the condition. (Each condition variable has an associated queue.) If no process is waiting on the condition, the notification is discarded. Unlike **WAIT**, **NOTIFY** does not release the monitor lock. Therefore you can leave the monitored data in an arbitrary state, so long as you restore the invariant before the next time you release the lock (by exiting the entry procedure).

BROADCAST condition wakes up all processes waiting on the condition variable. If no processes are waiting on the condition, the broadcast is discarded. Like **NOTIFY**, the monitor lock is held during this operation.

10.2.4.1 Producer/Consumer problem

Consider the buffering scheme described in the beginning of this chapter. Because of the synchronization limitations imposed by **FORK/JOIN**, we could only use two buffers. A more general solution, however, would allow the two operations to share a buffer pool. This buffer pool would be bounded, as shown in the example on the next page:

```
DIRECTORY
  Heap USING [systemZone],
  MStream USING [Handle, ReadOnly, ReadWrite],
  Process USING [Detach],
  Stream USING [Delete, EndOfStream, GetChar, Handle, PutChar];

CircularBuffer: MONITOR IMPORTS Heap, MStream, Process, Stream =
BEGIN
maxElements: CARDINAL = 10;    --max number of buffers
bufferSize: CARDINAL = 128;
zone: UNCOUNTED ZONE ← Heap.systemZone;

Elmt: TYPE = LONG POINTER TO Buffer;
Buffer: TYPE = RECORD[
  length: CARDINAL ← 0,
  chars: ARRAY [0..bufferSize) OF CHARACTER ← ALL[' ]];
BufferArrayType: TYPE = ARRAY [0..maxElements) OF Elmt ← ALL[NIL];

get, put: CARDINAL [0..maxElements] ← 0; --which buffer being read/written
bufferArray: BufferArrayType;
notEmpty: CONDITION;
notFull: CONDITION;

-- The consumer gets a buffer from the monitored array of buffers and writes its
-- contents to another file. This process blocks if there are no buffers available.
Consumer: PROCEDURE[outStream: MStream.Handle] =
BEGIN
  DO
    myBuffer: Elmt ← ConsumeBuffer[];
    FOR i: CARDINAL IN [0..myBuffer.length) DO
      ch: CHARACTER ← myBuffer.chars[i];
      IF ch = '& THEN GOTO Exit;
      Stream.PutChar[outStream, ch];
    ENDLOOP;
    zone.FREE[@myBuffer];
  ENDLOOP;
  EXITS Exit => Stream.Delete[outStream];
END;

-- Producer produces buffers of information obtained from reading a file.
-- It blocks when there is no more room in the monitored array of buffers
Producer: PROCEDURE[inStream: MStream.Handle] =
BEGIN
  DO
    myBuffer: Elmt ← zone.NEW[Buffer];
    FOR i: CARDINAL IN [0..bufferSize) DO
      myBuffer.chars[i] ← Stream.GetChar[inStream! Stream.EndOfStream =>
       {myBuffer.length ← i; GOTO Exit}];
    ENDLOOP;
    ProduceBuffer[myBuffer]; -- put buffer in monitored buffer array
  ENDLOOP;
  EXITS Exit => Stream.Delete[inStream];
END;
```

```
-- Produce Buffer is called when the Producer needs a buffer.
ProduceBuffer: ENTRY PROCEDURE[element: Elmt] =
BEGIN
   ENABLE UNWIND => NULL;
   WHILE (put + 1) MOD maxElements = get DO WAIT notFull ENDLOOP;
   bufferArray[put] ← element;
   put ← (put + 1) MOD maxElements;
   NOTIFY notEmpty
END;

-- Consume Buffer returns a previously allocated buffer to the available buffer list
ConsumeBuffer: ENTRY PROCEDURE RETURNS[element: Elmt] =
BEGIN
ENABLE UNWIND => NULL;
   WHILE get = put DO WAIT notEmpty ENDLOOP;
   element ← bufferArray[get];
   get ← (get + 1) MOD maxElements;
   NOTIFY notFull;
END;

Init: PROCEDURE[] =
BEGIN
   inStream: MStream.Handle ← MStream.ReadOnly[
      name:"inFile"L,
      release: [NIL,NIL]];
   outStream: MStream.Handle ← MStream.ReadWrite[
      name:"outFile"L,
      type: text,
      release: [NIL,NIL]];
   Process.Detach[FORK Consumer[outStream]];
   Process.Detach[FORK Producer[inStream]];
END;

--mainline code
Init[];
END...
```

In this example, **bufferArray** is an array that can contain at most **maxElements** (10) elements (buffers). The **bufferArray** starts out empty. The **Producer** (the process reading input) allocates buffers, fills them with information, and adds them to the buffer pool via **ProduceBuffer**. If the buffer pool is full, **ProduceBuffer** waits until there is room. After adding the element to the buffer, **ProduceBuffer** notifies any waiting consumers that another element is available. Similarly, the **Consumer** (the process processing the input) receives its elements by calling **ConsumeBuffer**. If there are no elements in the buffer pool **ConsumeBuffer** waits. Once an element becomes available, **ConsumeBuffer** removes it and notifies any waiting producer processes that the buffer pool is not full.

Notice that a condition variable *c* is always associated with some boolean expression describing a desired state of the monitor data. Each **WAIT** must be embedded in a loop that checks the validity of the corresponding boolean. In Mesa, **NOTIFY** is regarded as a *hint* to a waiting process; it causes a process waiting on the condition variable to resume execution at some convenient time in the future. When the waiting process resumes, it will reacquire the monitor lock. But there is no guarantee that some other process will not enter the monitor before the waiting process. Therefore, the waiting process must

reevaluate the condition before continuing. The general pattern for condition variable code is therefore:

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;
```

When appropriate, the process mechanism always does a **NOTIFY**, even when there are no processes waiting to be notified. The reason for this is that the built in check (and discard mechanism) is more efficient than any explicit test you could use to avoid the **NOTIFY**. Thus, for example, **ProduceBuffer** always notifies **notEmpty** even if no process is waiting.

This arrangement results in an extra evaluation of the condition after a wait. In return, however, it avoids extra process switches and puts no constraints on when the waiting process must run after a notify. This method is preferable and efficient in Mesa because in general few processes are waiting on the same condition variable at the same time (not many processes will be notified), and context switching is fast (it does not take long for all processes to recheck the state).

10.2.4.2 Single resource manager

Controlling access to a limited shared resource is another common problem that requires interprocess synchronization. The following code segment illustrates a simple storage allocator for objects of uniform size.

```
StorageAllocator: MONITOR =
  BEGIN
  storageAvailable: CONDITION;

  Block: TYPE = RECORD [...];              -- or some other data type
  ListPtr: TYPE = LONG POINTER TO ListElmt;
  ListElmt: TYPE = RECORD[block: Block, next: ListPtr];
  freeList: ListPtr ← NIL;

  Allocate: ENTRY PROC RETURNS [elmt:ListPtr] =
    BEGIN
    ENABLE UNWIND => NULL;
    WHILE freeList = NIL DO WAIT storageAvailable ENDLOOP;
    elmt ← freeList;
    freeList ←elmt.next;
    END;
```

```
Free: ENTRY PROC [elmt:ListPtr] =
  BEGIN
  ENABLE UNWIND => NULL;
  elmt.next ← freeList;
  freeList ← elmt;
  NOTIFY storageAvailable;
  END;
END...
```

freeList is the global linked list of available storage. **Allocate** waits until **freeList** is not empty to remove an element. **Free** puts an element back on the **freeList** and notifies any process waiting in **Allocate** that more storage is available.

10.2.4.3 Variable size, single resource manager

If a resource manager manipulates variable sized objects, notification will not work as well. The difficulty is that NOTIFY only wakes up one process when more storage is available. Since the size of storage requests vary, available storage may not be enough to meet the needs of the process that is awakened, but it may be enough to satisfy another waiting process.

In this case, you should use BROADCAST instead of NOTIFY. A BROADCAST wakes up all waiting processes. Since the WAIT condition statement occurs in a WHILE loop, each process will check state before continuing and put itself to sleep if there is not enough storage. Thus, processes that need a smaller amount of storage will be able to continue.

Here is an example of this sort of storage allocator:

```
StorageAllocator: MONITOR =
  BEGIN
  storageAvailable: CONDITION;

  Block: TYPE = RECORD [...];              -- or some other data type
  ListPtr: TYPE = LONG POINTER TO ListElmt;
  ListElmt: TYPE = RECORD[block: Block, next: ListPtr];
  freeList: ListPtr ← NIL;

  Allocate: ENTRY PROC[size: CARDINAL] RETURNS [elmt:ListPtr] =
    BEGIN
    ENABLE UNWIND => NULL;
    UNTIL <storage chunk of size words available> DO WAIT storageAvailable ENDLOOP;
    elmt ← <remove chunk of size words>;
    END;

  Free: ENTRY PROC [elmt:ListPtr, size: CARDINAL] =
    BEGIN
    ENABLE UNWIND => NULL;
    <put back storage of size words>
    ....
    BROADCAST storageAvailable;
    END;
END...
```

Again, the waiting processes treat notification only as a hint. A process that is awakened does not assume that the condition is true; rather, it assumes that state has changed, and that it should check to see if the condition is true.

10.3 Issues and concerns

This section discusses some issues associated with monitors and processes: how to abort a process, and the relationships between signals and processes, and signals and monitors.

10.3.1 Aborting a process

In addition to NOTIFY and BROADCAST, you can also resume a waiting process with a timeout or an abort. We discuss Abort in this section; for a discussion on using timeouts see section 9.3.2 of the MLM.

Abort does really not abort the process; it merely raises a signal that indicates to the process that it should clean itself up and return. (If the process is detached, Pilot will destroy it when it returns.) However, the aborted process is free to do arbitrary computations before returning, or indeed to ignore the abort entirely.

You can raise the signal **Abort** by calling Process.**Abort**, with the process to be removed as its argument. The signal is raised the next time the process WAITs on any condition variable that has aborts enabled (the default is to not have aborts enabled; you can call Process.**EnableAborts** to reverse this). If the process is currently waiting it is aborted immediately.

If you want to abort a process that never waits on a condition variable, you must periodically force the process to pause. Process.**Pause** causes a process to wait with aborts enabled for a specified length of time.

10.3.2 Signals and process

Though the creation of a new process via FORK is similar to a procedure call, the new process has a different call stack with the forked procedure as the root of the activation. The implication of this is that signals will not cross process activations. Any signal not caught by a new process will not continue to propagate to its parent; instead the debugger will be invoked with an uncaught signal.

10.3.3 Signals and monitors

Signals interact with monitors (entry procedures) in two special ways; in raising a signal and in handling UNWIND. Both cases are motivated by the need to release the monitor lock.

When you raise a signal from an entry procedure, the lock is not released. Thus, catch phrases, which can invoke arbitrary operations, may deadlock if they try to reenter the monitor. For errors, you can avoid this with the RETURN WITH ERROR construct.

```
RETURN WITH ERROR NoSuchObject;
```

This statement has the effect of removing the currently executing process from the call chain before issuing the ERROR. Thus, if you execute this statement within an entry procedure, the monitor lock is released before the error is started.

For example, consider the following code 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;
```

Executing 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. The lock is held until the unwind occurs.

Executing RETURN WITH ERROR Failure[2] releases the monitor lock and destroys the frame of Proc before propagation of the signal begins. 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.

Another important issue regarding signals is the handling of UNWIND. The monitor lock is released as part of the UNWIND, so any entry procedure that may experience an UNWIND must catch it and restore the monitor invariant:

```
Proc: ENTRY PROCEDURE[...] =
  BEGIN
  ENABLE UNWIND = > BEGIN <restore invariant> END;
  ...
  END;
```

At the end of the outermost UNWIND catch phrase, the compiler appends code to release the monitor lock before the frame is destroyed.

Even if you don't have to restore the monitor invariant, you should still catch UNWIND in every entry procedure in which it might propagate. The compiler will not generate the code to release the lock unless the UNWIND catch phrase is present. If the monitor is not released during an UNWIND, ensuing calls to the monitor will deadlock.

10.4 Summary

You can spawn new processes from existing ones via the FORK operation. FORK creates a new process, with the invoked procedure as the root of the activation, and returns a process id of type PROCESS to identify the object.

Once instantiated, a new process will either run forever, run for a finite time and return values to (or need to be synchronized with) another process, or run for a finite time without returning results to another process. In the first case, FORKing the new process is sufficient.

In the second case, when a process is expected to return results, you can synchronize its return with the JOIN construct. At this junction, the returning process is deleted and the joining process receives the results and continues its execution.

In the third case, when a process is not JOINed, you must ensure that the process activation is removed. If you use Process.Detach, Pilot will delete the process when it returns to its root procedure.

Concurrent processes create a need for cooperation and communication. Monitors and condition variables provide this cooperation by allowing controlled access and synchronization through shared variables and code.

Mesa monitors are module instances with an associated monitor lock. Mutual exclusion to shared variables (global variables in the monitor module) is ensured by allowing only one process to hold the lock at a time.

In addition to a collection of data and an associated lock, a monitor contains a set of procedures that perform operations on the data. There are three kinds of procedures: entry, internal, and external. External procedures are declared as normal procedures and logically live outside the monitor. Calls to these procedures do not acquire the monitor lock. Entry procedures provide controlled access into the monitor. Calls to an entry procedure either acquire the monitor lock or block until the lock can be acquired. Internal procedures contain the common routines shared among the several entry procedures. These procedures execute with the monitor lock held, and therefore may freely access the monitored data.

Synchronization is accomplished with condition variables and the operations WAIT, NOTIFY, and BROADCAST. A WAIT releases the monitor lock before it blocks. NOTIFY and BROADCAST do not release the lock. Therefore WAIT statements occur in loops, since the condition that was notified may no longer be true when the blocked processes wakes up.

This chapter discussed only the most common form of monitor lock, the global monitor lock. Mesa also supports more specialized forms of monitors, including monitored records and object monitors. Consult chapter 9 of the *Mesa Language Manual* for more details.

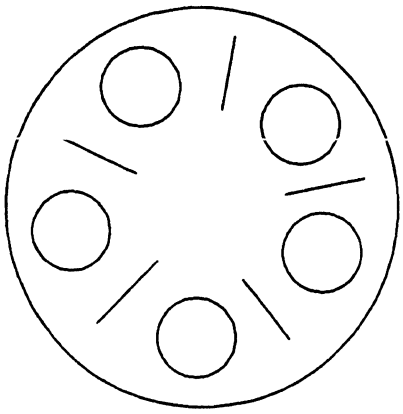
10.5 References

Read Chapter 9 of the *Mesa Language Manual* on Processes and Concurrency.

Read "Experience with Processes and Monitors in Mesa" by Lampson and Redell. (Page 191 of the *Office Systems Technology* book.)

10.6 Exercises

The basic assignment for this chapter is to implement the dining philosophers problem. In this problem, you have 5 philosophers at a dining table. However, there is only one chopstick between each plate, and a philosopher needs 2 chopsticks to eat. At any given time, a philosopher may be thinking, eating, or waiting for the philosopher next to him to put down a chopstick so he can use it.



You can tell a philosopher to try to start eating, or to stop eating and start thinking. When a philosopher is told to start eating, he will look around for some chopsticks and start eating if he can; otherwise he will wait. When a philosopher is told to start thinking, he stops eating (puts down his chopsticks); other waiting philosophers will then see if they can start eating.

```
Philosopher1: {thinking, waiting, eating}
Philosopher2: {thinking, waiting, eating}
Philosopher3: {thinking, waiting, eating}
Philosopher4: {thinking, waiting, eating}
Philosopher5: {thinking, waiting, eating}

Philosopher # 1 is eating.
Philosopher # 2 must wait to eat.
Philosopher # 1 has finished eating.
Philosopher # 2 is eating.
```

There are two levels to this problem, easy and hard. The hard assignment is to solve the dining philosophers problem by yourself. For the easy assignment, we have provided two interfaces and part of the implementation; you only need to write two procedures. If you are adventurous, go start solving the problem now. If you are less adventurous, read the next page to get some help in solving this problem.

For the easier version of this problem, you need to implement the procedures **BeginEating** and **EndEating** from the DP interface:

```
-- DP.mesa

DP: DEFINITIONS =
  BEGIN
  numOfPhils: CARDINAL = 5;

  BeginEating:  PROCEDURE[philosopher: CARDINAL];
  EndEating:    PROCEDURE[philosopher: CARDINAL];
  IsWaiting:    PROCEDURE[philosopher: CARDINAL];
  IsEating:     PROCEDURE[philosopher: CARDINAL];

  END..
```

BeginEating will be called every time a philosopher (a process) thinks it might be able to eat. The philosopher will look around him (look at an array) and see if he can start eating. If he can't, he informs the world that he must wait to eat, calls the procedure **DP.IsWaiting**, and then waits. If he can eat, he informs the world that he is eating, uses his chopsticks (sets some variables in an array) and calls the procedure **DP.IsEating**.

EndEating will be called every time a philosopher has been told to stop eating and start thinking. He should inform the world that he is no longer eating, set down his chopsticks, and tell all waiting philosophers (if any) that they might want to try to start eating. Note that although the tool refers to philosophers 1 through 5, **philosopher** in the above procedures will range from 0 through 4.

To communicate with the world, use the procedures provided in the **ToolDefs** interface:

```
-- ToolDefs.mesa

ToolDefs: DEFINITIONS =
  BEGIN

  PostText: PROCEDURE[string: LONG STRING];     --writes a string of text
  PostLine: PROCEDURE[string: LONG STRING];     --writes a string of text with CR
  PostNumber: PROCEDURE[num: CARDINAL];         --writes a number

  END..
```

You need to write the implementation module DPImpl.mesa, which implements the procedures **BeginEating**, and **EndEating** in the **DP** interface. Use a monitor and a condition variable to synchronize access to the chopsticks by the 5 philosophers (processes). You will need the files DP.mesa, ToolDefs.mesa, DPTool.mesa, and DiningPhilosophers.config, which are on the course directory for this chapter.