

Training Overview

Your Instructors

This week your instructors could be any combination of the people listed below. If you have any comments about the course structure or content, feel free to message Jim Herz (e-mail addresses listed below.)

Jim Herz <Herz:OSBU North:Xerox>

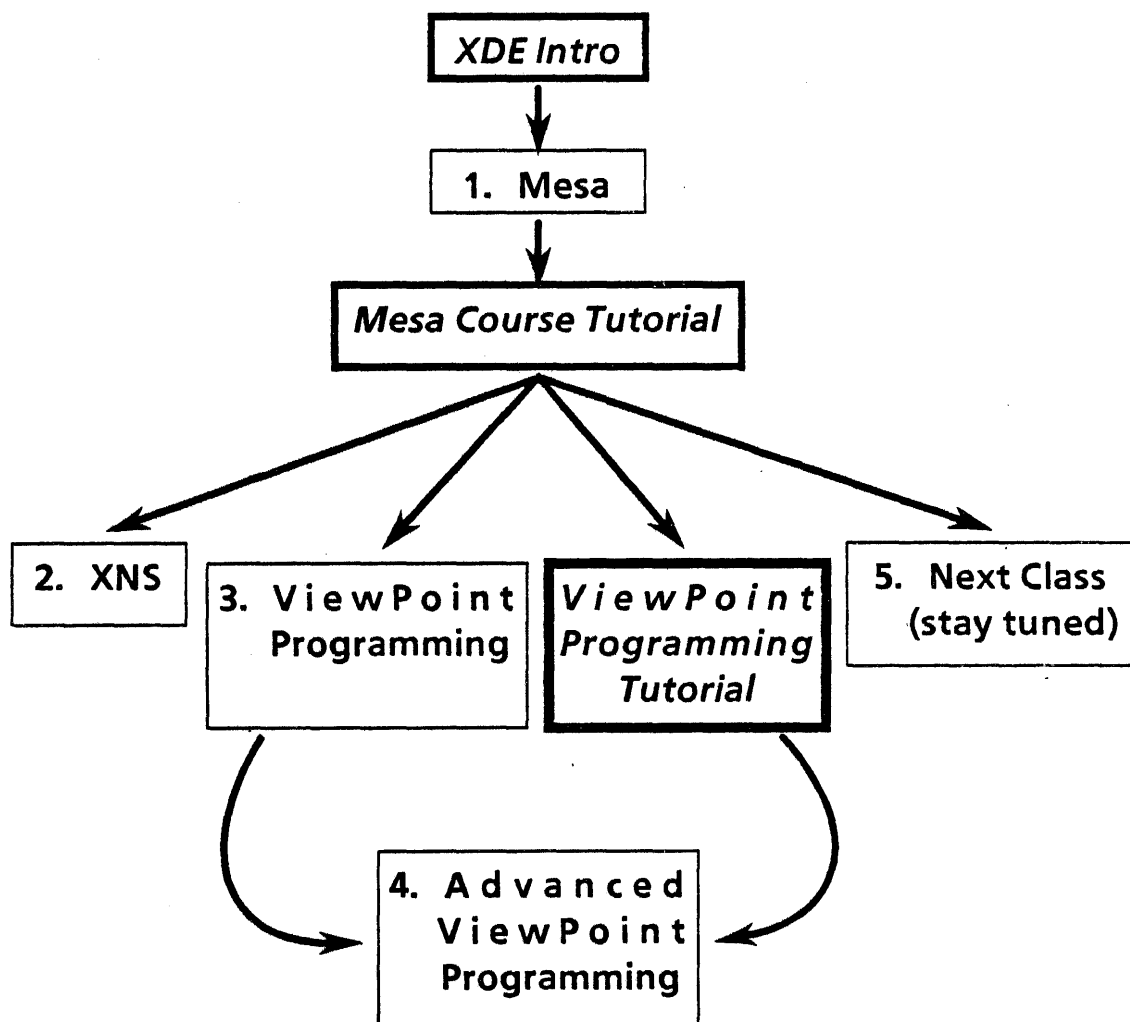
Gail Kubeczko

Holly Wanless

Grant Ruiz

Available XDE Training

Here is the ordering in which XDE training should be done. Bold boxes represent self-paced tutorials. The numbered boxes represent stand-up courses offered by our Training Group. Classes are usually held in Sunnyvale, California. Descriptions of each are on the following pages.



The Stand-Up Courses

UNIT 1. The Mesa Language

Introduces a new user to the Mesa programming language. Stress is placed on possibly unfamiliar features such as explicit storage management (allocation and deallocation), modules and configurations, monitors, processes, and signals. System issues are not covered in detail.

UNIT 2. Introduction to Xerox Network Systems Protocols and Agents

Defines the Xerox Internet Transport Protocols. Emphasis on definition and illustration of Network Systems remote procedure call protocol (Courier). Examination of service-like applications interacting with top level protocols.

UNIT 3. Xerox ViewPoint Programming

In this unit we introduce ViewPoint architecture, user interface, tool set, and design methodology for applications built on the ViewPoint base. Topics covered include Programmer/User interaction, running programs from the desktop, registering programs with the desktop, NS file manipulation, and application folders.

UNIT 4. Advanced ViewPoint Programming

This class will focus on writing applications that interact with documents and other applications in a sophisticated manner. Topics will include using advanced graphics, notifying procedures when user actions occur, writing and managing TIP (Terminal Interface Package) tables, and managing a selection.

The Self Study Materials

- **XDE Concepts and Principles**

A high level overview of XDE and Xerox Network Services architecture. Intended to provide a philosophical and conceptual framework on which to base an understanding of XDE.

- **Introduction to XDE**

A computer aided tutorial that will familiarize a new user with the XDE tool set and associated user interface, operation of essential tools, editing, and XDE documentation. This tutorial will take two to three days to complete.

- **The Mesa Course**

A self paced Mesa Language programming tutorial that includes representative instructional software and is intended for use at a customer site. Its purpose is to provide instruction in tool development skills and familiarize a new user with basic Mesa and Pilot interfaces. This course will take four to eight weeks to complete.

- **The ViewPoint Programming Course Tutorial**

A self paced programming tutorial completed at the customer site; it is designed to follow the one week Xerox ViewPoint Programming Course. The ViewPoint Programming Tutorial assumes that you are an experienced Mesa Programmer. The course covers the range of programming skills required for development of sophisticated ViewPoint applications. The course with all exercises will take three to eight weeks to complete.

Course Objective

Need: The Mesa language is one of the cornerstones of the Xerox Development Environment. Being a proficient Mesa programmer is an essential first step to effective use of the power of XDE.

Objective: Graduates will be able to prepare, compile, debug, and run Mesa programs. These programs will exercise features of Mesa that are new or unfamiliar to most students.

Please Note: This class is NOT geared to teach basic computer science topics. We assume that you are EXPERIENCED programmers. We will teach experienced programmers how to program in the Mesa Language.

Do: Ask questions at any time. We may, however, defer answering some questions or ask you to submit them through the mail system.

Submit your course evaluations daily using the mail system.

Have fun.

Don't: Instigate religious debates by touting your favorite (non-Mesa) language. We are here to teach you Mesa.

Expect to learn everything about Mesa in one week. We will get you started in the right direction.

We're Not Your Mother

The general format of the class is lecture in the morning, lab in the afternoon. The lab has no format: you are expected to complete your assignments (or at least try) but you can come and go as you wish.

Major No-No: Don't leave for three hours in the middle of the afternoon and then come back and expect us to stay here with you until 10 PM.

Daily Schedule

Day 1 - Monday

9:00-9:45	Introduction and Course Organization Training Overview Mesa Language Pilot Operating System Physical and Logical Volumes
9:45-10:15	Getting Started in Mesa The Fundamental Module Types What a PROGRAM module looks like Comments Basic Data Types and Expressions
10:15-10:30	Break
10:30-12:00	Getting Started in Mesa (Cont'd) Common constructed Data Types Statements Procedures Additional data types, Extensions The Debugger
12:00-1:00	Lunch
1:00-6:00	Debugging and Compiling Exercises Ask for Solution when finished

Daily Schedule (Cont'd)

Day 2 - Tuesday

- 9:00-10:15 Interfaces
Review Definitions
Opaque Types
Examples of Interface modules
Examples of Program modules
- 10:15-10:30 Break
- 10:30-11:15 Interfaces (Cont'd)
Examples of configurations
More Examples
- 11:15-12:00 Interfaces Exercise
- 12:00-1:00 Lunch
- 1:00-6:00 Interfaces Exercise (Cont'd)
Ask for Solution when finished

Day 3 - Wednesday

- 9:00-10:30 Dynamic Storage Allocation
Different types of heaps
Declaring and creating heaps
Allocating and deallocating from heaps
- 10:30-10:45 Break
- 10:45-11:30 Form Subwindow Layout Tool
- 11:30-12:00 Dynamic Storage Allocation Exercise
- 12:00- 1:00 Lunch
- 1:00- 6:00 Dynamic Storage Allocation Exercise (Cont'd)
Ask for Solution when finished

Daily Schedule (Cont'd)

Day 4 - Thursday

9:00-10:15	Signals Signal Examples Operations with signals Catch Phrases UNWIND signals
10:15-10:30	Break
10:30-11:30	Streams
11:30-12:00	Streams / Signals Exercise
12:00- 1:00	Lunch
1:00- 6:00	Streams / Signals Exercise (Cont'd) Ask for Solution when finished

Day 5 - Friday

9:00-10:30	Processes and Concurrency Concurrent execution New language features for processes Monitors
10:30-10:45	Break
10:45-12:15	Processes and Concurrency (Cont'd) Condition Variables More about monitors Signals Deadlocks Other operations on processes
12:15- 1:15	Lunch
1:15- 6:00	Monitors Exercise Ask for Solution when finished

What is Mesa?

- A programming language
- An operating system, Pilot
- A processor architecture

History of Mesa

- Research started at PARC in 1971
- Went into production use at Xerox (OSD, ISD) DSBU in 1976
- Various Mesa machines include the Alto, Dandelion, Dorado, Daybreak (6085).

Architecture Highlights

- High-level Language Oriented
- Stack Machine: 16 bits wide, 8 - 14 words deep
- Large Shared Virtual Memory
 - 32 bit address space, 256 word pages
 - Word addressed, 16 bit words
 - 64K word Main Data Space
- Shared code and Data
 - Read only code, shared across modules
 - Global (static) data shared across processes
- Procedure Oriented
 - Parameter passing on stack
 - Heap allocation of activation records
 - Single transfer primitive (XFER)
 - Local and Global calls, single return
- Process Mechanism
 - Preemptive event driven scheduling, 8 priority levels
 - Monitors and Conditions
 - Interrupts, Timeouts, and Aborts
 - Fork any procedure
- Dense Instruction
 - Average instruction is 1.4 - 1.5 bytes

Advantages of Mesa

- Support of large scale system application development through direct and efficient support of structured, modular programming.
- Reduced product costs through 2x programmer productivity gain *strongly typed, interfaces*
- Efficient machine implementations

Mesa Language

- High level systems programming language
- Strong type checking
- Modular programming
- Flexible control transfer mechanisms
- Concurrent processes with protected, shared data
- High density object code

The Mesa Language

Mesa is a language that is basically similar to Pascal but which extends Pascal in a number of directions intended to make it more effective for the development of large systems. Mesa features include (among others):

- **Rich Type System**

Basic: BOOLEANS, CHARACTERS, INTEGERS, CARDINALS, REALS

Constructed: Subrange, Enumerated, ARRAYS, RECORDS, POINTERS, STRINGS, Variant RECORDS, PROCEDURES, PROGRAMS, SIGNALS, ERRORS, PROCESSES

- **Simple Control Structures**

Statements: Assignment, IF, SELECT, FOR, WHILE, DO

Compounds: Blocks, PROCEDURES, PROGRAMS

- **Module Structure**

Interfaces: DEFINITIONS modules

Clients and Implementors: PROGRAM modules

Systems: CONFIGURATION modules

- **Explicit Storage Management**

Heaps and Zones

- **Exception Handling**

SIGNALS and ERRORS

- **Concurrent Processes**

Creation and Destruction: FORK, JOIN

Locking: MONITORS and ENTRY procedures

Synchronization: CONDITIONS, WAIT, NOTIFY

Mesa Features with no Pascal Counterparts

- Type checking across separately compiled modules
- Interface modules
- Concurrent process support
- Procedure variables - procedures are a full fledged type
- Explicit storage management - no garbage collection
- Default field values
- Base and relative pointers
- Pointer arithmetic
- Constructors and extractors
- Exception handling mechanisms

Pilot: The Mesa Operating System (Overview)

Pilot defines a "Basic Machine" that is an abstraction of the physical resources provided by the hardware. The purpose of the basic machine is to:

- Define a standard interface that is independent of the size, speed, model, and configuration upon which it is operating
- Provide a uniform environment for program design
- Insulate clients as much as possible from variation in hardware configuration

Facilities of Pilot

- Runtime support for the Mesa Programming Language including Mesa processes.
- Virtual Memory Management.
- Local Disk Management, for local file system setup and manipulation.
- Packages for management of processes, physical memory, bitmap display, and CPU.
- Device I/O Package, for local peripheral management.
- Communications package, for network streams, packet exchange, modem support, and other basic communication services.
- Streams for handling sequential I/O in a device independent way.
- Common Software, for handling device I/O, strings, and formatting.
- Diagnostics, for detecting and analyzing hardware problems.

Pilot is Not a General-Purpose Time Sharing System

- No master/slave mode; no protection against malicious programs
- No enforcement of resource allocation/billing/accounting
- Client assumed to be a collection of cooperating processes
- Client controls specialized devices directly
- No user interface

The Main Data Space

The Main Data Space (MDS) is a contiguous region of 64K words of virtual memory. Its purpose is to allow the most commonly used data structures to be referenced by single word pointers rather than double word pointers.

- It is used mostly by low level system clients. Generally, application programmers *should not* allocate storage for user data from a MDS.
- User storage allocated from the MDS is referenced indirectly using POINTERS, any other storage is referenced using LONG POINTERS.

Note: In pre Pilot 14.0 releases, both Local and Global Frames are allocated from the MDS. In post Pilot 14.0 releases, Local frames are allocated from the MDS but Global frames are allocated from an anonymous backing store.

The Xerox Development Environment

- XDE (Xerox Development Environment) is the programming environment for our products.
In XDE, one can write programs for XDE itself as well as for the Viewpoint environment. As an example of what can be done with XDE, all of XDE, Viewpoint and Network Services was developed using Mesa in the Xerox Development Environment.
- XDE includes everything the programmer uses:
 - all programming tools and applications
 - all programming interfaces
 - all support materials
 - product support

XDE Support

How you get initial support for XDE depends on who your company is.

If you are a:

Xerox internal programmer, contact: **XDEConsultants:All Areas:Xerox**

Rank Xerox employee, contact: **XDESupport:SBD-E:RX**

Commercial customer, USMG Analyst, USMG Sales Rep, contact:
XDESupport:OSBU North:Xerox

System Configuration

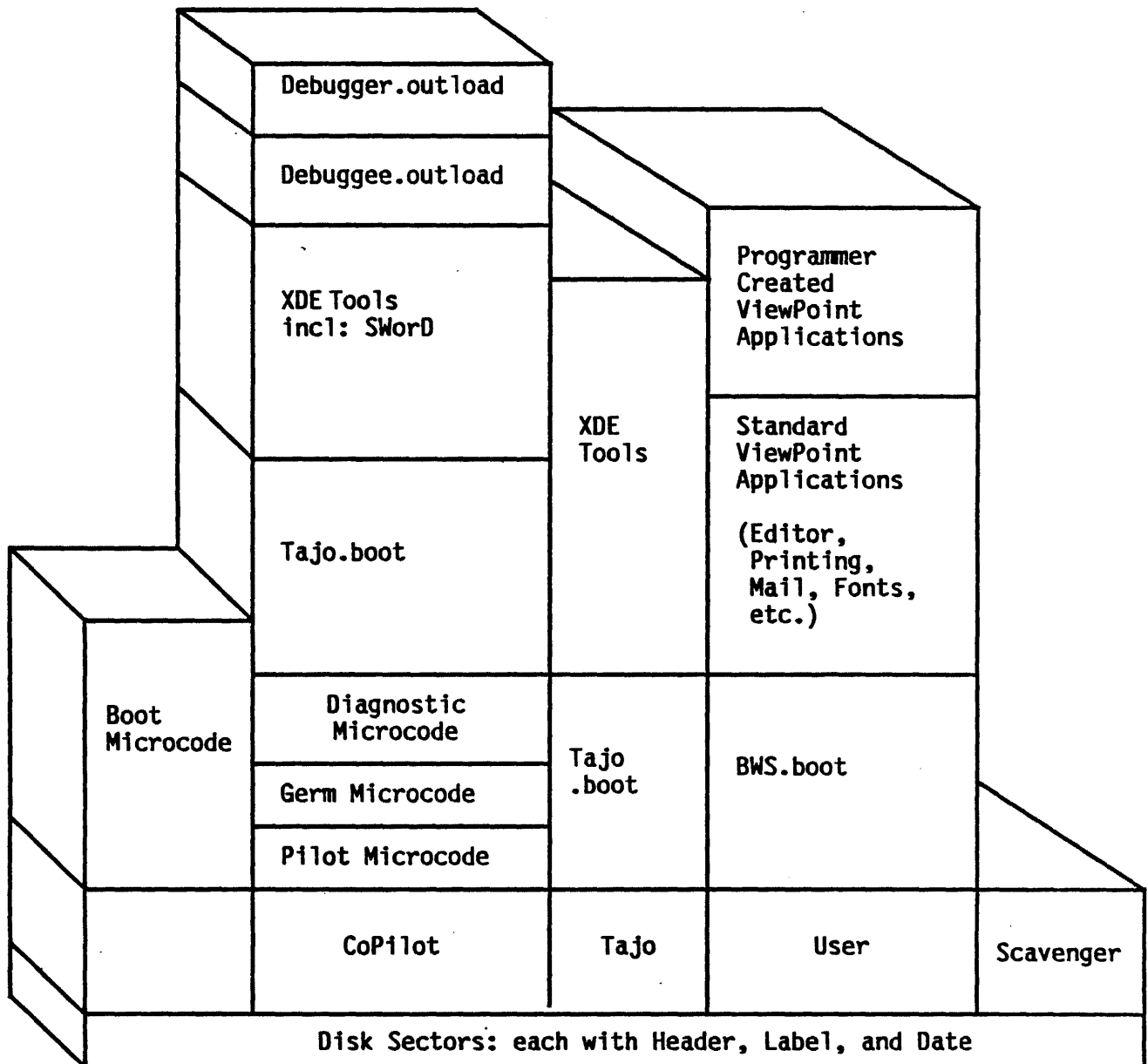
- A **physical volume** is the basic unit available for random access file page storage. A Physical volume corresponds to a storage device, typically a disk.
- A **logical volume** is a partition of storage for client files, including system data structures for manipulating those files.

Typically, a physical volume is divided into one or more logical volumes. Each logical volume is largely protected from actions in other logical volumes. Different logical volumes contain different systems.

<u>Volume Name</u>	<u>Contents</u>
User	BWS bootfile, ViewPoint data files and applications
Scavenger	Can be used as backing store for the User volume; File system recovery software for the User volume is located here, if needed
CoPilot	CoPilot bootfile [pre-14.0 releases only] (CoPilot bootfile = Tajo bootfile + built-in debugger), plus debugging files, XDE tools, XDE user files ** OR ** Tajo bootfile & Sword [12.3 releases or later only] plus debugging files, XDE tools, XDE user files
Tajo	(If present) Tajo bootfile, XDE tools

A standard configuration for Viewpoint developers might be a User and Scavenger volumes for Viewpoint and one XDE volume running Sword (Same WORld Debugger).

Graphically Speaking

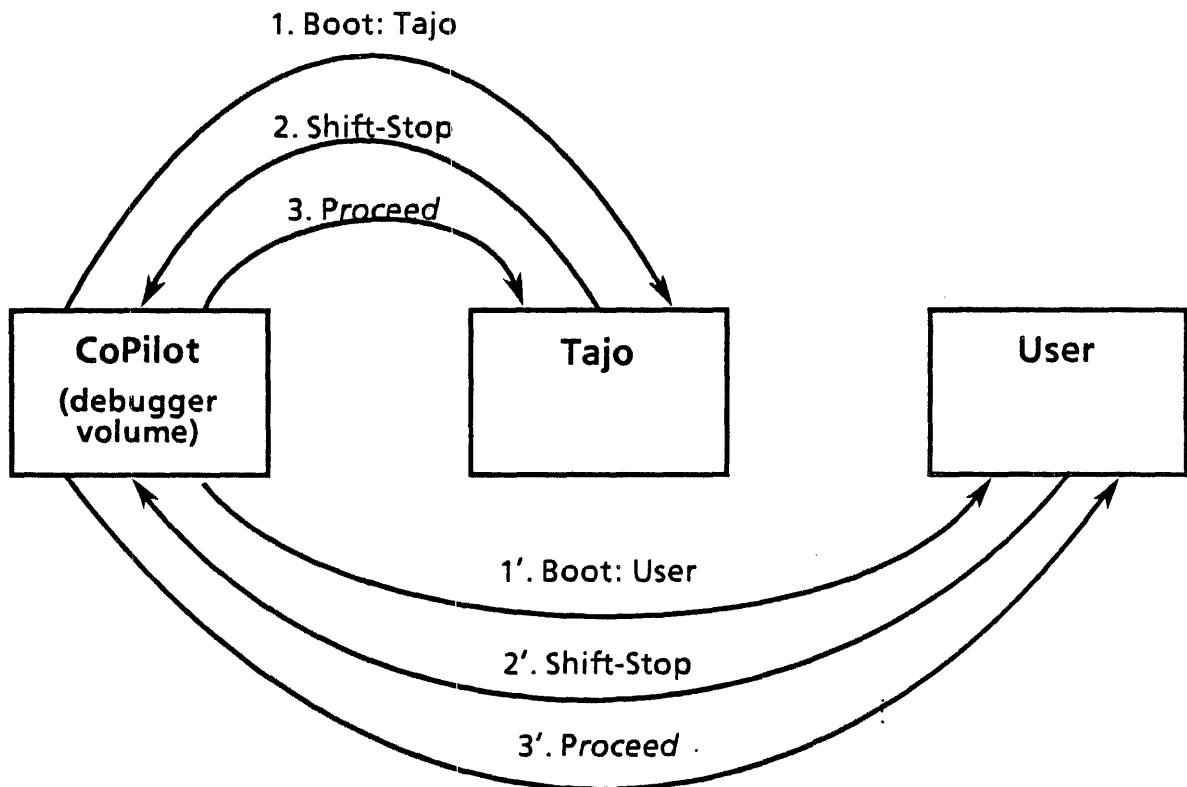


World Swapping

With the CoPilot debugger, you can only keep the state of *one* client around at any given time.

With Sword, you can debug *multiple* clients! In addition, Sword can debug in the same world, as well as be the resident debugger for client volumes.

For world-swap debugging, you must boot the client volume once in order to establish a debugger-client relationship. But once that relationship is established, it is easy to swap between the debugger and the client.

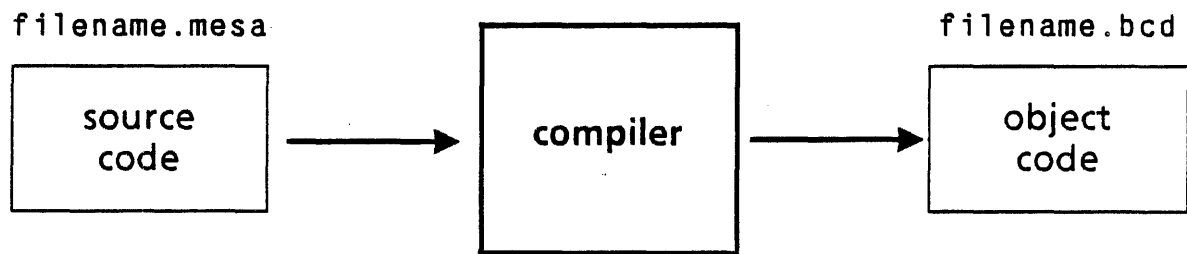


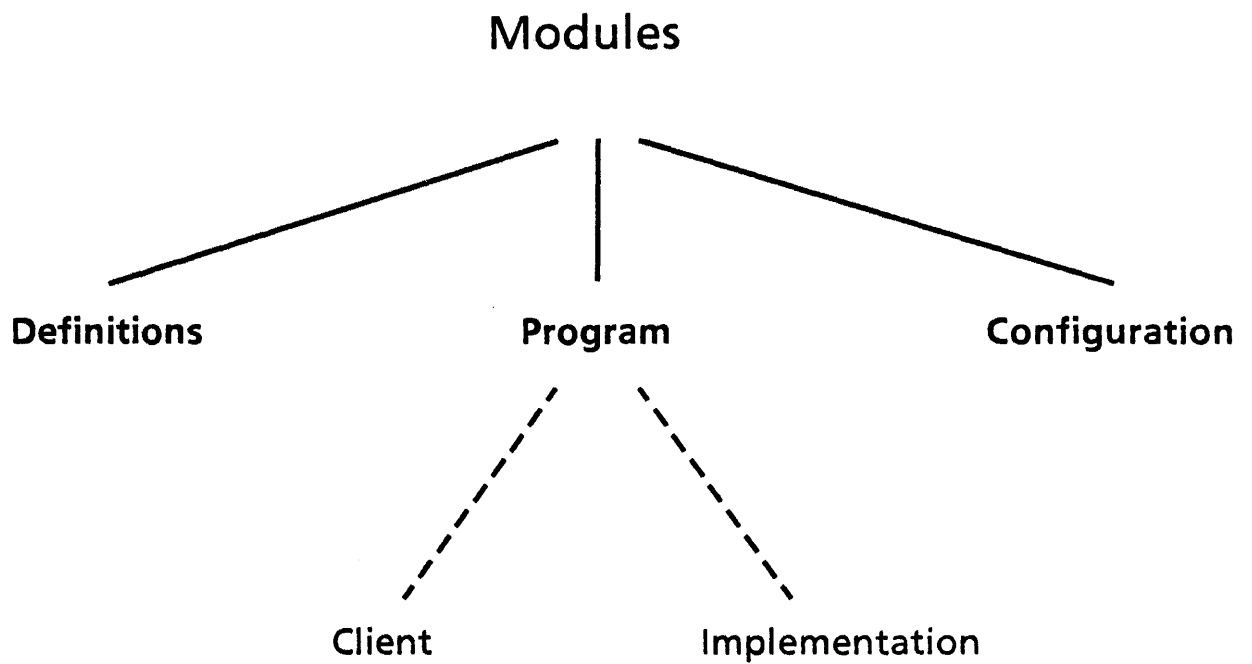
Getting Started in Mesa

Outline

1. Getting Started in Mesa
 - a. The Fundamental Module Types
 - b. What a PROGRAM Module looks like
 - c. Comments
 - d. Basic Data Types and Expressions

Compiling Modules



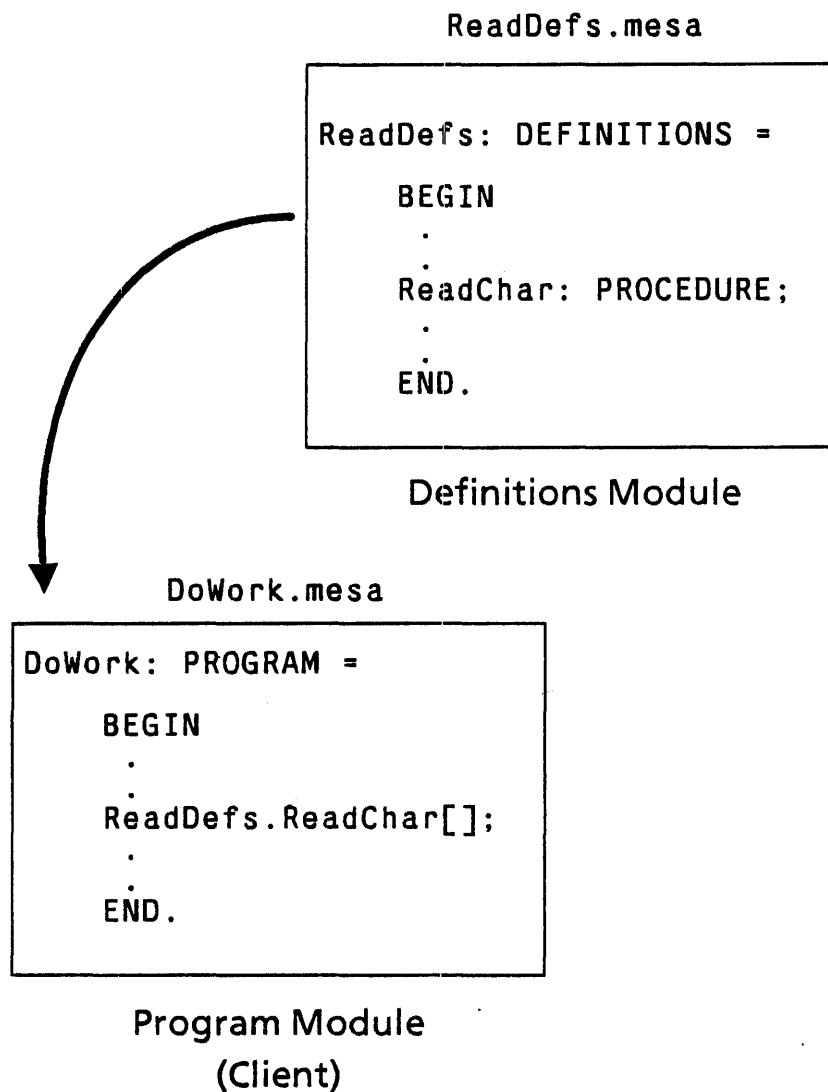


Definitions files (a.k.a Interfaces) define an abstraction.

Program files contain executable code that implement the abstraction.

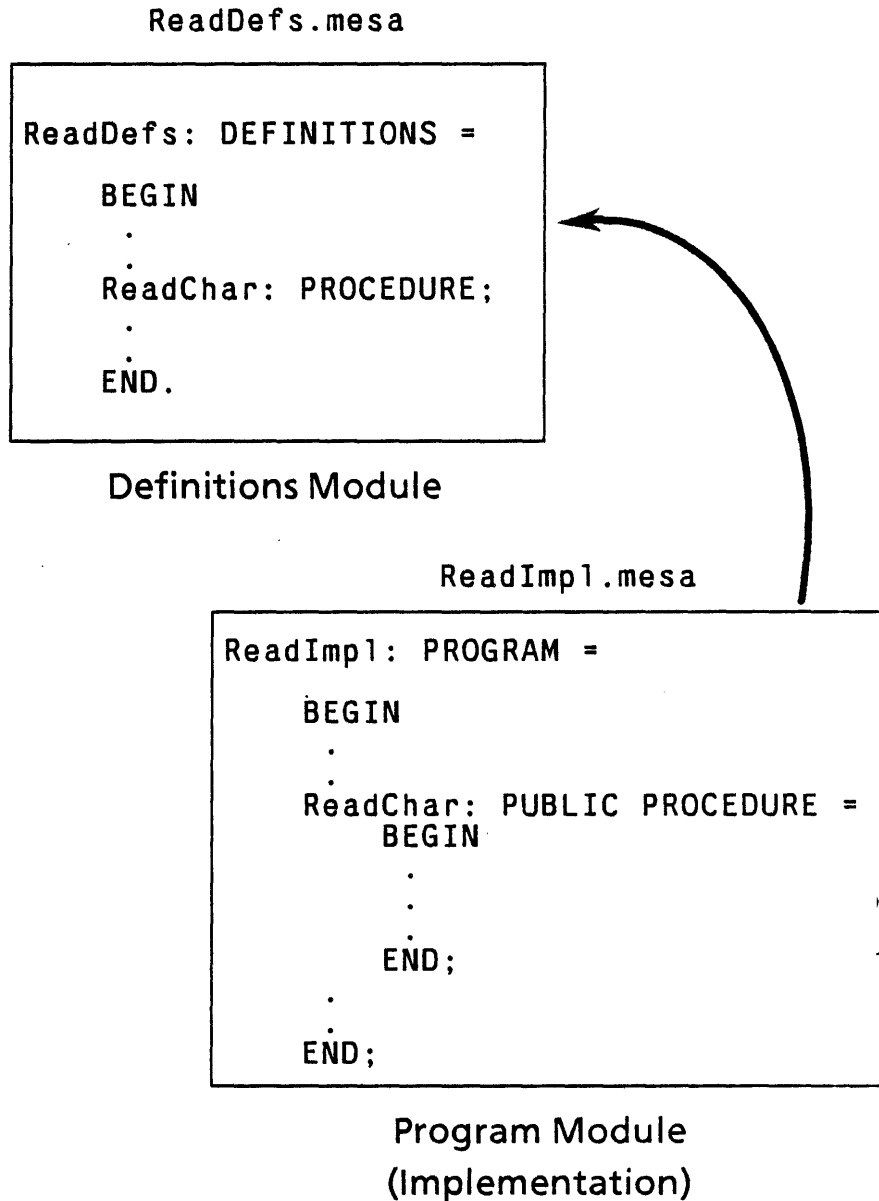
Configuration files specify how program modules are combined.

Program (Client) and Definition Modules



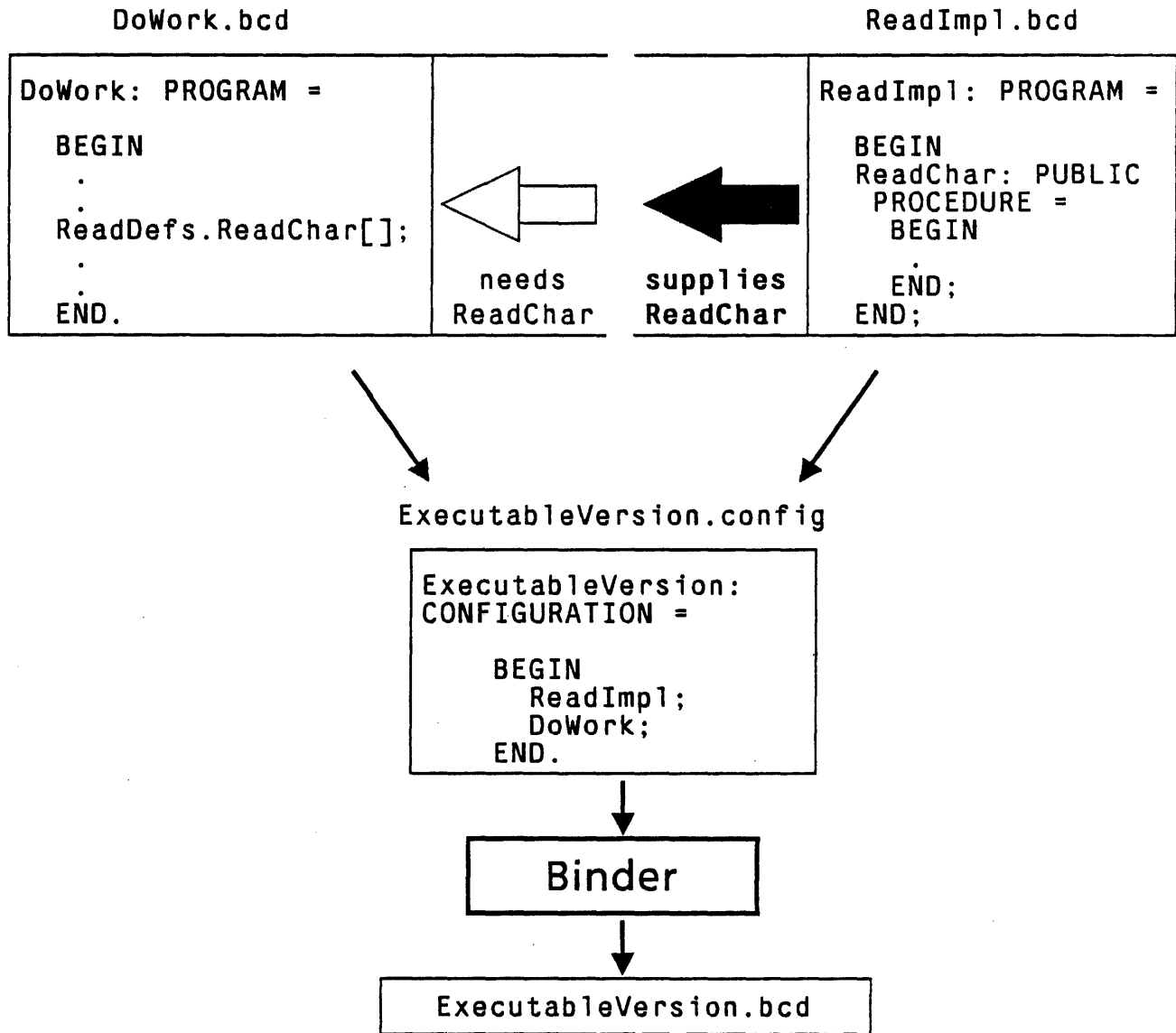
The PROGRAM module, DoWork, is using the procedure ReadChar that is defined in ReadDefs. The implementation of ReadChar is unknown to this module.

Program (Implementation) and Definition Modules



The PROGRAM module, *ReadImpl*, supplies the actual implementation for the procedure *ReadChar* that is defined in *ReadDefs*.

Fitting Client and Implementation Modules Together



A CONFIGURATION file brings together the PROGRAMS DoWork and ReadImpl so that the implementation for ReadChar is around when DoWork calls for it.

Program Module Structure & Syntax

DIRECTORY

```
<InterfaceName1> USING [<ProcName1>, ...],  
<InterfaceName2> USING [<ProcName1>, ...],  
.  
.  
<InterfaceNameN>;
```

<ModuleName>: PROGRAM

```
IMPORTS <InterfaceName1>, <InterfaceName2>  
EXPORTS <InterfaceNameN> =
```

BEGIN

```
<type, variable, procedure declarations>;
```

```
<main code, if any>
```

END.

Note: {...} is an alternative to any BEGIN...END.

Program Module Example

DIRECTORY

```
Stream USING [GetBlock, Delete],
Window USING [Create],
ReadDefs;
```

ReadImpl: PROGRAM

```
IMPORTS Stream, Window
EXPORTS ReadDefs =
```

BEGIN

```
...
n: CARDINAL ← 0;
...
DoSomethingInteresting: PUBLIC PROCEDURE = {
  h ← Window.Create[...];
  WHILE n < 100 DO
    ...
    Stream.GetBlock[...];
    ...
    Stream.Delete[...];
    ...
  ENDLOOP;
};
...
END.
```

Comments

--This comment is terminated at the end of the line.

IF i = 1 -- Middle of line comment -- THEN

<<This comment ignores all
carriage-returns. It ends here.>>

Identifiers

Identifiers can be any mixture of upper letters, lower case letters, and digits. The first character must be a letter. Upper and lower case letters *are different* and do distinguish identifiers. All characters are significant.

Examples:

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

Mesa Reserved words are always all upper case. For a complete listing see Appendix E of the Mesa Language Manual.

Examples:

```
BEGIN
END
PROGRAM
INTEGER
CARDINAL
```

Built-in Element Types

(INT)EGER -- $[-2^{N-1} .. 2^{N-1})$ -- $N = 16$
 (NAT)URAL -- $[0 .. 2^{N-1})$ -- $INTEGER \cap CARDINAL$
 CARDINAL -- $[0 .. 2^N)$
 (BOOL)EAN -- FALSE | TRUE
 (CHAR)ACTER -- all characters:

Examples:

```
lowerCaseA: CHAR ← 'a;  -- character literals are assigned with a single quote
mark: CHAR ← ' ;
endMarker: CHAR ← ';;
asciiCR: CHAR ← 15C;  -- octal
tab: CHAR ← \t;  -- escape convention
```

[] indicates inclusion

() indicates exclusion

LONG INTEGER and LONG CARDINAL

LONG INTEGER -- $[-2^{2N-1} .. 2^{2N-1})$ -- $N = 16$

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

REAL

Mesa has adopted the proposed IEEE standard for floating-point arithmetic.

Examples:

4.32

0.15

8.0E-19

The mantissa is therefore represented by 24 bits (including the "hidden bit") with the decimal point to the right of the first bit; the exponent is represented by 8 bits with a range of -126 to 127 (All 0's and all 1's are exceptional conditions).

Declarations

All identifiers must be declared in one of 3 ways:

Simple declaration:

```
<identifier>: <type>;
```

Initial Value declaration:

```
<identifier>: <type> ← <initial value>;
```

Constant declaration:

```
<identifier>: <type> = <constant value>;
```

Examples:

```
first: CARDINAL;  
condition: BOOLEAN;  
pageCount: CARDINAL ← 0;  
isOn: BOOLEAN ← TRUE;  
pi: REAL = 3.141562;  
indirectPi: REAL = pi;  
startingPageCount: CARDINAL = pageCount;
```

Numeric Operators

+ - / * MOD

/ truncates toward zero for integers

MOD yields the remainder of dividing 2 numbers

It does not apply to REAL operands

The sign of MOD is the sign of the dividend

Examples of various expressions:

n

15

(i + j + k)

-15

3.14

m*n

n MOD 8 -- the result has the sign of n

i+1

Exponents, powers, and roots are implemented in software.

Relational Operators

The operators below apply to all *ordered* types

< <= = # > >=

NOT ~

IN <interval>

Examples of various expressions:

n = 5

m # n

m ~= n

i <= j

(i < j) = (j < k)

n IN [1..5]

i NOT IN [-1..5]

Boolean Operators

NOT ~ AND OR

Evaluation is from left to right and stops when the value of the expression has been determined.

Examples of various expressions:

NOT i = 15

~q

~(p AND q)

i <= j AND j < k

p AND ~q

i=5 AND j NOT IN [-1..1]

m>n OR m= 5

~p OR ~q

Character Arithmetic

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

Subtracting 2 CHARACTER values yields an INTEGER value.

Examples:

```
c: CHARACTER ← 'c;  
digit: INTEGER;  
digit ← c - '0;  
c ← c + ('A - 'a);    -- converts lower case to upper case
```

Function-like Operators

- PRED SUCC -- used with all element types and LONG CARDINAL and LONG INTEGER. The values of PRED[x] and SUCC[x] are the predecessor and successor of x respectively.
- ORD -- converts a character or enumerated value into a numeric value.
- VAL -- is the inverse of ORD:
c:CHARACTER ← VAL[101B];
- FIRST LAST -- used with all element types and LONG CARDINAL and LONG INTEGER. These yield the least and greatest values respectively.

Precedence

Operators in order of decreasing precedence:

- + -- unary operators

* / MOD

+ -

= # < <= > >= IN

~ NOT

AND

OR

←

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

Static Type Determination

The inherent type of every expression and variable in Mesa can be determined by static analysis.

Type rules in Mesa take 2 forms:

Target Type Rule: The inherent type of a variable or expression must conform* to a target type.

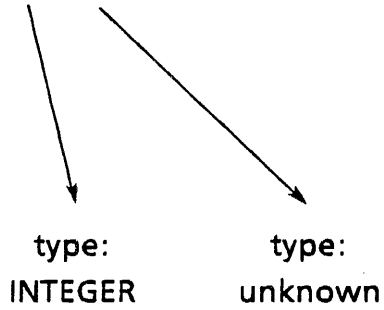
Balancing Rule: The inherent type of a variable or expression must satisfy a relation for a known set of types.

*Type A *conforms* to another type B if Mesa can convert a variable of type A to be of type B at runtime.

Target Type Example

x: INTEGER;

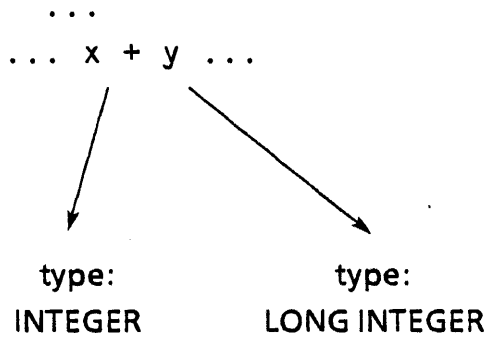
x ← y;



TYPE unknown must conform to TYPE INTEGER.

Balancing Example

```
x: INTEGER;  
y: LONG INTEGER;
```



An INTEGER cannot be directly added to a LONG INTEGER; balancing must take place.

Bounds Checking

When mixing numeric types:

Not all CARDINALS are valid INTEGERS

Not all INTEGERS are valid CARDINALS

Not all LONG INTEGERS are valid CARDINALS (using a range assertion).

.

.

If bounds checking is requested of the compiler (switch b), code will be inserted before each *cross assignment* to ensure that the value is within range.

Otherwise, it is the responsibility of the programmer to ensure that the conversion is valid.

Examples:

```
i: INTEGER; c: CARDINAL;
```

```
c ← 40000;
```

```
i ← c;    -- With the b switch, this generates a bounds fault  
          -- (because this is a cross assignment)
```

Domains of Numeric Operators

When the result of an operation falls outside the range of its assumed type, overflow or underflow occurs.

It is the programmer's responsibility to guard against overflow and underflow conditions.

Examples:

```
i, j, k: INTEGER;  
a, b, c: CARDINAL;
```

```
i ← 30000;  
j ← 30000;  
k ← i + j;           -- k has value -5536 (all variables are INTEGERS)
```

```
a ← 4;  
b ← 5;  
c ← a - b;          -- c has value 65535 (all variables are CARDINALs)
```

```
k ← a - b;          -- With the b switch, this generates a bounds fault  
-- (because this is a cross assignment)
```

Outline

1. Getting Started in Mesa (cont'd)
 - ...
 - e. Common Constructed Data Types
 - i. Type Conformance in general
 - ii. The element types
 - iii. Arrays
 - iv. Records
 - v. Pointers
 - f. Statements
 - i. Assignment statements and expressions
 - ii. IF statements and expressions
 - iii. SELECT statements and expressions
 - iv. Blocks
 - v. Loop statements and loop control

Conformance and Constructed Data Types

In an assignment statement, the right side must conform to the left side.

B ← A

There are 3 relationships a type A may have to type B:

A conforms to B.

A freely conforms to B.

A is equivalent to B.

is equivalent **→** freely conforms **→** conforms

Equivalence -- Mesa sees no difference between type A and type B if **A and B are equivalent.**

Free Conformity -- At runtime, Mesa can store any value of type A into a variable of type B without checking, change of representation, or other computation if **A freely conforms to B.**

Conformity -- Mesa can convert a variable of type A to type B at runtime if **A conforms to B.** (Runtime computation is required.)

Often whether or not A conforms to B depends on whether parts of A are equivalent or freely conform to parts of B.

Conformance issues related to specific Mesa TYPES will be covered when the TYPE is introduced.

Enumerated Types

Enumerated types may be declared as in Pascal:

```
Color:    TYPE = {red, orange, yellow, violet};
foreground: Color;
background: Color;
Fruit:    TYPE = {orange, lemon};
state:    {disconnected, busy, available};
```

```
i: CARDINAL;
```

```
foreground ← orange;
background ← VAL[2];    -- assigns yellow
i ← ORD[foreground];    -- assigns 1
IF Color[orange] > foreground THEN . . .
```

Note that `Color` and `Fruit` are 'named' enumerated types while `{disconnected, busy, available}` is an 'anonymous' enumerated type.

Conformance:

Every appearance of an enumerated type definition generates a new type that is not equivalent to, and does not freely conform or conform to, any other enumeration.

Therefore, 2 TYPE definitions with the same definitions, letter for letter, define different types.

Subrange Types

Subrange types may be declared for all *element* types:

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

UpperCase: TYPE = CHARACTER['A..'Z'];
AssertTrue: TYPE = BOOLEAN[TRUE..TRUE];
```

The *basetype* for a subrange is that type of which it is a subrange and which is not itself a subrange.

The assignment of a value to a subrange variable makes an associated assertion that the value is in the appropriate interval.

If bounds checking is requested of the compiler (switch *b*), code will be inserted before each assignment to a subrange variable to ensure that the value is within range.

Otherwise, it is the responsibility of the programmer to ensure that the value is valid.

Example:

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

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


Subrange Types

The syntax for subranges follows mathematical notation:

- `[]` indicate inclusion
- `()` indicate exclusion

The following intervals all designate the range from -1 to 5 inclusive. The endpoints of a subrange must be compile-time constants:

`[-1..5]` `[-1..6)` `(-2..6)` `(-2..5]`

The initial type identifier may be omitted if each bound in the interval specifies a short numeric value:

```
s1: [-10..10];      -- signed rep
s2: [100..33000];  -- unsigned rep
s3: [0..10);       -- both reps
```

Conformance:

A subrange type conforms to its base type and a base type conforms to any of its subrange types. Subrange type `T[i..j]` freely conforms to `T` if `i = FIRST[T]` and to `T[i..k]` if `j ≤ k`. Two subrange types are equivalent if their base types are equivalent and if the corresponding bounds are equal.

Arrays

An array variable can be declared like:

```
<varname>: <PACKED> ARRAY <indextype> OF <componenttype>;
```

Or you can declare an array TYPE like:

```
<name>: TYPE = <PACKED> ARRAY <indextype> OF <componenttype>;
```

And then array variables would be declared like:

```
<varname1>: <name>;
```

```
<varname2>: <name>;
```

Examples:

```
IndexTyp: TYPE = [0..10);
ArrayType1: TYPE = ARRAY IndexTyp OF INTEGER;
ArrayType2: TYPE = ARRAY [0..10) OF INTEGER;
nameArray: ARRAY [0..10) OF INTEGER;
anotherArray: ArrayType1;
```

The <indextype> can be any element type. The <componenttype> can be any type including another array type.

Arrays may be initialized when they are declared.

Examples:

```
octalChar: ARRAY[0..7] OF CHARACTER =
  ['0','1','2','3','4','5','6','7'];
dashes: ARRAY[0..7] OF CHARACTER ← ['-','-','-','-','-','-','-'];
dashes: ARRAY[0..7] OF CHARACTER ← ALL['-'];
freshVector: ARRAY[0..3) OF CARDINAL = ALL[0];
currentVector: ARRAY[0..3) OF CARDINAL ← freshVector;
```

Array Constructors

Array constructors may be used to assign all the components of an array in an assignment statement:

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

When the array type is implied by context, the type identifier may be omitted:

```
triplet ← [11, 12, 13];
```

The function ALL may also be used during initialization (only when the array type is implied by context):

```
Matrix3by4: TYPE = ARRAY[0..3) OF ARRAY [0..4) OF CARDINAL;  
allOnes: Matrix3by4 ← ALL[ALL[1]];
```

You can, of course, access individual components of an array:

```
triplet[1] ← 82;  
allOnes[2][3] ← 82; -- 2nd row, 3rd column
```

Array Equivalence and Conformance

Two array types are equivalent if both their index types and their component types are equivalent and if they are both packed or both unpacked.

Conformance:

An array type freely conforms to another if the component type of the first freely conforms to that of the second, the index types are equivalent, and they are both packed or both unpacked. Packed arrays with non-equivalent types do not freely conform.

Examples:

```
IndexTyp: TYPE = [0..10);  
ArrayType1: TYPE = ARRAY IndexTyp OF INTEGER;  
ArrayType2: TYPE = ARRAY [0..10) OF INTEGER;  
Numbers: TYPE = PACKED ARRAY [0..10) OF INTEGER;
```

ArrayType1 and ArrayType2 are equivalent.

Records

Most records are declared with named field-lists:

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

```
oldTime, newTime: MilitaryTime;
```

You can assign parts of the record field by field:

```
oldTime.hrs ← 8;  
oldTime.mins ← 0;
```

Or you can assign the entire record with either a keyword or positional constructor:

```
oldTime ← [mins:0, hrs:8];    -- Fields may be in any order  
oldTime ← [8,0];            -- Fields must be in order
```

Sometimes records are declared with unnamed field-lists:

```
RecType: TYPE = RECORD [CARDINAL, CARDINAL];
```

Only a positional constructor can be used with such records.

Default Field Values

When a record type is declared, default values may be specified for each field. Fields in a record constructor may be *voided*, *elided*, or *omitted*:

```
Rec:  TYPE = RECORD [  
    v1:  CARDINAL,  
    v2:  CARDINAL ← 3];
```

```
rec:Rec;
```

```
rec ← [v1:4, v2:5]; -- v2 gets 5 (overrides default)  
rec ← [v1:4];      -- v2 is omitted, so v2 gets 3  
rec ← [v1:4, v2: ]; -- v2 is elided, so v2 gets 3
```

Record Extractors

Extractors are used to "explode" record objects and assign their components to individual variables in a single statement:

```
MonthName: TYPE = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
  Oct, Nov, Dec};
```

```
Date: TYPE = RECORD[
  day: [1..31];
  month: MonthName,
  year: [1900..2000)];
```

```
birthDay: Date;
```

```
dd: [1..31];
mm: MonthName;
yy: [1900..2000);
```

```
birthDay ← [8, Aug, 1959];
[dd, mm, yy] ← birthDay;
[dd, , yy] ← birthDay;
[dd, mm, yy] ← Date[25, Apr, 1943];
[month:mm, day:dd, year:yy] ← birthDay;
```

The type of a constructor must be explicitly stated when an extractor is on the left side.

Record Extractors

A value with a single-component record type may be converted automatically to a value with the type of that component.

```
j: INTEGER;  
k: RECORD[i: INTEGER ← 5];  
...  
j ← k;
```


Record Conformance

Every appearance of a record constructor creates a new type that is not equivalent to, and does not conform to, any other record type.

```
RecType1: TYPE = RECORD[a,b: INTEGER];  
rec1: RecType1;
```

```
RecType2: TYPE = RECORD[a,b: INTEGER];  
rec2: RecType2;
```

```
rec3: RECORD[a,b: INTEGER];  
rec4: RECORD[a,b: INTEGER];
```

The record variables `rec1`, `rec2`, `rec3`, and `rec4` all have different, non-conforming types. None of these can be assigned to any of the others.

Pointers

POINTERS are one word objects that, therefore, reference objects in the MDS in virtual memory. The MDS is 64K words of virtual memory in which system data structures and all local frames of executing processes reside.

LONG POINTERS are two word objects that, therefore, reference objects outside the MDS. Most dynamically allocated objects are outside the MDS. For generality, LONG POINTERS are often used to reference objects in the MDS, as well.

In general, use LONG POINTERS.

```
intPtr: POINTER TO INTEGER;  
boolPtr: LONG POINTER TO BOOLEAN;  
intPtr ← NIL;
```

Note: NIL is a Mesa reserved word that conforms to any POINTER or LONG POINTER type. It denotes that the pointer value has no valid referent.

Also, In Post Pilot 14.0 releases, global frames do not reside in the MDS (as they did previously). Therefore (short) POINTERS can not be used to indirectly access global variables.

Pointer Dereferencing

Referents of pointers can be obtained with the operator \uparrow . Pointer values can be generated from objects with the operator $@$.

```
AgeRange: TYPE = [21..150);
SexValues: TYPE = {male, female};
PartyValues: TYPE = {democratic, republican};

Person: TYPE = RECORD[
  age: AgeRange,
  sex: SexValues,
  party: PartyValues];

candidate1, candidate2: Person;
winner, loser: LONG POINTER TO Person ← NIL;

loser ← @candidate1; -- loser gets the address of candidate1
winner ← @candidate2;
winner↑.age ← 45;
winner.age ← 45;      -- Mesa automatic dereferencing
```

Pointer Conformance and Equivalence

Two pointer types are equivalent if their reference types are equivalent and if they are both long pointers or both short pointers:

```
IntPtrType: TYPE = LONG POINTER TO INTEGER;  
a: IntPtrType;  
b: LONG POINTER TO INTEGER;
```

Here, a and b have the same type.

Dangling Pointers

Be careful to avoid dangling pointers to local storage:

```
pointer1, pointer2: LONG POINTER TO INTEGER;
...
RiskyProc: PROCEDURE[i: INTEGER] = {
  local: INTEGER;
  ...
  pointer1 ← @i;    -- the storage for i and local exists only for
  pointer2 ← @local; -- the life of this procedure
  ...
  RETURN;          -- all local storage is released
};
```

After the procedure, RiskyProc returns, pointer1 and pointer2 will point to variables that no longer exist.

Assignment Statements and Expressions

The type of the <rightside> must conform to the type of the <leftside>:

```
<leftside> ← <rightside>;
```

```
a ← b + c;
```

A variable may also be assigned using an assignment expression. The type and value of an assignment expression (multiple assignment statements) is the type and value of the <leftside>:

```
a ← b ← c ← 4;      -- a, b, and c all get 4
```

IF Statements

Standard If-Then statements:

```
IF <boolean expression> THEN
  <statement>;
```

Example:

```
IF x > 5 THEN
  x ← x * 2;
```

Standard If-Then-Else statements:

```
IF <boolean expression> THEN
  <statement1>
ELSE
  <statement2>;
```

Example:

```
IF x > 5 THEN
  x ← x * 2
ELSE
  y ← y + 200;
```

IF Expressions

The IF Expression syntax is similar to that of an IF Statement. There are two differences:

- 1) The clauses contain *expressions*, not statements, and
- 2) an IF Expression *must* have an ELSE clause.

```
IF <boolean expression> THEN
  <expression>
ELSE
  <expression>
```

Example:

```
x ← 

|               |
|---------------|
| IF x > 5 THEN |
| x*2           |
| ELSE          |
| x + 200       |

 ;
```


SELECT Statements

SELECT statements are similar to Pascal CASE statements.

```
SELECT <expression> FROM
  <expression1> => <statement1>;
  <expression2> => <statement2>;
  <expression3> => <statement3>;
  ...
  <expressionN> => <statementN>;
ENDCASE => <statement>;
```

Example:

```
i: CARDINAL;
```

```
SELECT i FROM
  0 => i ← i + 1;           --i = 0
  <3 => { j ← i; i ← i - 1 }; --i = 1 or i = 2
  =5 => i ← 0;             --i = 5
ENDCASE => i ← 2;         -- none of the above
```

Select Statements

A single SELECT arm may specify more than one test in one arm.

Example:

```
i,j,k: CARDINAL;
```

```
SELECT i*j+k FROM
```

```
1, IN[7..10] => <stmt1>; -- values: 1,7,8,9,10
```

```
2, 5, >10 => <stmt2>; -- values: 2,5,11,12, ...
```

```
ENDCASE; -- no arm for endcase
```

SELECT Expressions

SELECT expressions can be used in the same way as IF expressions (with the same restrictions):

```
SELECT <expression> FROM
  <expression1> => <result expression1>,
  <expression2> => <result expression2>,
  <expression3> => <result expression3>,
  ...
  <expressionN> => <result expressionN>,
  ENDCASE => <result expression>
```

Example:

```
pt: INTEGER;      -- Point on a line
lo, hi: INTEGER; -- Bounds for a line
...
PointPosition: TYPE = {leftMargin, rightMargin, inside,
  outside, degenerate};
position: PointPosition;
```

```
position ← 
SELECT pt FROM
  IN (lo..hi) => inside,
  NOT IN [lo..hi] => outside,
  < hi      => leftMargin, -- =lo but #hi
  > lo      => rightMargin, -- =hi but #lo
  ENDCASE  => degenerate  -- =lo and =hi--
;
```

Blocks

The general structure of a compound statement is:

```
BEGIN
  <OpenClause>
  <EnableClause>
  <DeclarationSeries>
  <StatementSeries>
  <ExitsClause>
END;
```

Everything but the <StatementSeries> is optional.

An <OpenClause> allows more convenient reference to the fields of a record and symbols in an interface.

An <EnableClause> is used with signals.

The <DeclarationSeries> allows you to declare variables at the beginning of any block:

```
IF lo > hi THEN
  BEGIN
    temp: INTEGER ← lo;
    lo ← hi;
    hi ← temp;
  END;
```

GOTO Statements

A series of labeled statements may be written immediately preceding the END in a block. One can jump to these statements from within the block using a GOTO statement:

```
IF ... THEN
  BEGIN
    IF ... THEN GOTO useDefault;
    IF ... THEN GOTO fileIsDefault;
    IF ... THEN GOTO newFile;
    ...
  EXITS
    useDefault, fileIsDefault => {...};
    newFile => pages ← 0;
  END;
```

Restrictions:

- 1) A GOTO may only jump forward in a program, never backward, and
- 2) A GOTO may only jump out of a block, never into one.

Loop Statements

The basic structure of a loop is shown below:

```
<LoopControl> DO
  <OpenClause>
    <EnableClause>
      <DeclarationSeries>
        <StatementSeries>
      <LoopExitsClause>
    ENDLLOOP;
```

The loop control can either be a form of conditional control or iterative control or both.

Conditional Control

Either a WHILE loop or UNTIL loop can be used in loop control:

```
WHILE <boolean expression> DO
  <statement>;
  <statement>;
  ...
ENDLOOP;
```

```
UNTIL <boolean expression> DO
  <statement>;
  <statement>;
  ...
ENDLOOP;
```

Examples:

```
i ← 1;
WHILE i < 10 DO
  i ← i + 1;
ENDLOOP;
```

```
i ← 1;
UNTIL i >= 10 DO
  i ← i + 1;
ENDLOOP;
```

Iterative Control -- Repetition

With repetition, a loop range specifies how many times the loop body is to be executed:

```
THROUGH <looprange> DO
  <statement>
ENDLOOP;
```

Example:

```
THROUGH [1..100] DO
  ...
ENDLOOP;
```

A loop range can have any element type *or* any subrange of LONG CARDINAL or LONG INTEGER. This is the one time that a subrange of a long numeric is allowed.

Repetition and a conditional test may be combined:

```
THROUGH [low..high] WHILE lineIsConnected DO
  ...
ENDLOOP;
```

Normal termination occurs after $(\text{high} - \text{low} + 1)$ iterations; conditional termination can occur sooner if `lineIsConnected` is FALSE prior to some iteration.

Iterative Control -- Iteration

Iteration is very similar to a FOR loop in Pascal:

```
FOR <preDeclaredControlVariable> IN <looprange> DO
  ...
ENDLOOP;
```

Example:

```
FOR i IN [1..length) DO
  ...
ENDLOOP;
```

The control variable can be declared at the point it's used:

```
FOR j: CARDINAL DECREASING IN [0..256) DO
  ...
ENDLOOP;
```

Iteration and a condition test may also be combined.

Iterative Control -- Assignment

With assignment, a control variable is given an initial expression and a next expression after each execution of the block. A condition test should be included to provide loop termination:

```
FOR <var> ← <initial expr>, <next expr> DO
  ...
ENDLOOP;
```

Example:

```
NodePtr: TYPE = LONG POINTER TO Node;
node: NodePtr;
head: NodePtr;
Node: TYPE = RECORD[
  listValue: CARDINAL,
  next: NodePtr];

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

GOTOs, LOOPS, and EXITS

A loop may be forcibly terminated by a GOTO or an EXIT. The LoopExitsClause serves the same purpose as the ExitsClause in a block. There are 4 differences.

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

GOTOs, LOOPS, and EXITs Examples

```
(1) & (2):  FOR i IN [0..nEntries) DO
             If a[i] = x THEN GOTO found;
             REPEAT                               -- REPEAT is not indicative
             found => old ← TRUE;
             FINISHED => {
                 a[i ← nEntries] ← x;
                 nEntries ← nEntries + 1;
                 old ← FALSE };
             ENDLLOOP;
```

```
(3):        DO
             ...
             IF ... THEN EXIT;
             ...
             ENDLLOOP;
```

```
(4):        stuff: ARRAY[0..100) OF PotentiallyInterestingData;

             FOR i: CARDINAL IN [0..100) DO
             ...
             IF NOT Interesting[stuff[i]] THEN LOOP;
             ...
             ENDLLOOP;
```

Outline

1. Getting Started in Mesa (cont'd)
 - ...
 - g. PROCEDURES
 - h. Additional Data Types, Extensions
 - i. Strings
 - ii. Variant Records
 - iii. Array Descriptors
 - iv. Base and Relative Pointers

Procedures

The definition of a procedure provides a name for a function or action:

```
<name>: PROCEDURE <argument record>
  RETURNS <result record> =
  BEGIN
    ...
  END;
```

Examples:

```
NewNumber: PROCEDURE RETURNS [x: CARDINAL] =
  BEGIN
    ...
  END;
```

```
Gcd: PROCEDURE[m, n: INTEGER] RETURNS [gcd: CARDINAL] =
  BEGIN
    r: INTEGER;
    UNTIL n = 0 DO
      r ← m MOD n;
      m ← n;
      n ← r;
    ENDLOOP;
    RETURN[ABS[m]];
  END;
```

Procedures -- Passing Arguments

In procedure call, the arguments are packaged into a record. Therefore, a procedure call may use all the syntax for record constructors in passing arguments. Arguments may be specified using either keyword or positional notation. Arguments not explicitly specified may be supplied by default. The following calls of `Gcd` are equivalent:

```
Gcd[x+1,y];      Gcd[m:x+1,n:y];      Gcd[n:y,m:x+1];
```

All parameters are passed by value.

If the procedure returns no results, the procedure is written as a normal statement:

```
...  -- code fragment
x ← x + 1;
Display[x];
...
```

Note: A procedure call that passes no arguments should include empty brackets to avoid confusion with procedure variables (explained later).

```
x ← SomeProc[];
```

Procedures -- Results

If the procedure returns results, the results are obtained with an extractor:

```
ExampleProc: PROCEDURE RETURNS [a,b:INTEGER] =  
  BEGIN  
    ...  
  END;
```

```
x,y: INTEGER;  
... -- code fragment  
[x,y] ← ExampleProc[];  
[b:y,a:x] ← ExampleProc[];
```

If the procedure returns only one result, the extractor is not necessary:

```
gcd: CARDINAL;  
gcd ← Gcd[m,n]; -- Calling the procedure declared earlier
```


Return Statements

A RETURN statement may be used anywhere in the procedure to terminate execution of the current procedure activation:

```
ReturnExample: PROC [option:[1..4]] RETURNS [a,b,c:INTEGER] = {
  a ← b ← c ← 0;
  SELECT option FROM
    1 => RETURN [a:1, b:2, c:3]; -- explicit RETURNS
    2 => RETURN [1,2,3];
    3 => RETURN;                -- returns the current values of a, b, c
  ENDCASE => b ← 4;
  c ← 9;
}; -- Implicit RETURN
```

Scoping Rules for Procedures

Mesa uses static scoping rules; each block defines a new scope:

```
SomeModule: PROGRAM =
```

```
BEGIN  
x,y: INTEGER;  
...  
OuterProc: PROCEDURE =
```

```
BEGIN  
x: BOOLEAN;  
...  
LocalProc: PROCEDURE
```

```
[parm1: CARDINAL] RETURNS[parm2: CARDINAL] =  
BEGIN  
x: CHARACTER;  
...  
END;
```

```
...  
END;
```

```
...  
END.
```

Procedure Types and Variables

Procedures are actually types that are similar to pointers.

```
TrigProc: TYPE = PROCEDURE[x: REAL] RETURNS[REAL]; -- Procedure TYPE
```

```
ArcSin: TrigProc = BEGIN ... END; -- Procedure Constant
```

```
ArcTan: TrigProc = BEGIN ... END; -- Procedure Constant
```

```
FooBar: TrigProc; -- Procedure Variable
```

```
...
```

```
FooBar ← ArcSin; -- this is not making a call to ArcSin, but rather assigning what  
-- the procedure ArcSin is equal to to the procedure Func
```

```
x ← FooBar[x]; -- since Func has meaning, it is valid to make a call to it
```

Procedure Equivalence and Conformance

Equivalence and conformance of procedure types are defined in terms of relations between fields of their parameter lists and result lists.

Two procedure types are equivalent if, for each pair of fields, the names are identical (or both are unnamed), the types are equivalent, and both default options are empty.

One field is compatible with another if the names are identical or if either is unnamed, and if the types are equivalent. A procedure type conforms freely to another if all corresponding fields are compatible.

Example:

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

```
SignedNumber: TYPE = INTEGER;
```

```
Int: TYPE = INTEGER;
```

```
ProcA: PROCEDURE[h: Handle, v: SignedNumber];
```

```
ProcB: PROCEDURE[h: Handle, v: Int];
```

```
ProcC: PROCEDURE[LONG POINTER TO Person, INTEGER];
```

ProcA, ProcB, and ProcC all conform to each other.

Inline Procedures

INLINE procedures can be used to eliminate the overhead of a procedure call and return usually at the cost of a longer object program.

If the attribute INLINE appears before the body in the declaration of a procedure, the call of that procedure is replaced by an inline expansion, a modified copy of the procedure's body:

```
Proc: PROCEDURE[v: INTEGER] RETURNS [INTEGER] = INLINE
  BEGIN
    RETURN[v*v + 3*v + 1];
  END;
```

See the rules in the MLM for applying the INLINE attribute.

Strings

In Mesa, a STRING represents a finite, possibly empty series of characters. Mesa contains the following predefined types:

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

Where:

text is a PACKED ARRAY of characters
 maxlength is the *maximum* numbers of characters that text can hold
 length is the number of characters *currently* in text

String variables can be declared and initialized two ways.

- 1) to specify the maxlength of the string, but not its contents
- 2) to assign a string literal to the variable

Examples:

```
LocalProc: PROC = {
  currentLine: STRING ← [256];    -- no value set here for the characters
  whatWasThat: STRING = "Eh?"L;  -- string literal
  ...                            -- currentLine gets a value somehow

  IF currentLine[2] = 'R THEN HelpaLot[];
    -- can reference a specific character in the string
  ...
};
```

Long Strings and String Operators

A LONG STRING is predefined as:

```
LONG STRING: TYPE = LONG POINTER TO StringBody;
```

Note: A LONG STRING does not mean that it can hold more characters than a STRING, it means that it is a LONG POINTER to a StringBody. The difference is that STRINGS are short pointers and therefore they reference into the MDS, whereas LONG STRINGS can access storage outside the MDS.

Also, in Post Pilot 14.0 releases, a string literal in the global frame can be assigned only to a LONG STRING, since the storage for global frames does not come from the MDS as in earlier releases.

In general, LONG STRINGS should be used instead of STRINGS.

Since STRINGS and LONG STRINGS are essentially pointers, the fundamental pointer operations \leftarrow , $=$, and $\#$ can be used with STRINGS and LONG STRINGS.

Variant Records

Variant records contain a set of common fields and a variant portion with a specified set of different possible interpretations. In Mesa, there are 4 different forms of the tag and they represent:

- an actual tag with an explicit enumerated type
- an actual tag implicitly defined
- a computed tag
- an OVERLAID tag

Variant Record Examples

```
Service: TYPE = {savings, checking};
Account: TYPE = RECORD[                -- actual tag with explicit type
  balance: REAL,
  specifics: SELECT variantType: Service FROM
    savings => [intRate: REAL, term: [30..365]],
    checking => [charges: REAL],
  ENDCASE];
```

```
Account: TYPE = RECORD[                -- actual tag with implicitly defined type
  balance: REAL,
  specifics: SELECT variantType:* FROM
    savings => [intRate: REAL, term: [30..365]],
    checking => [charges: REAL],
  ENDCASE];
```

```
Account: TYPE = RECORD[                -- computed tag
  balance: REAL,
  specifics: SELECT COMPUTED {savings,checking} FROM
    savings => [intRate: REAL, term: [30..365]],
    checking => [charges: REAL],
  ENDCASE];
```

```
Account: TYPE = RECORD[                -- overlaid tag
  balance: REAL,
  specifics: SELECT OVERLAID * FROM
    savings => [intRate: REAL, term: [30..365]],
    checking => [charges: REAL],
  ENDCASE];
```

* is used to indicate that the type of an actual tag is being defined implicitly by the set of adjectives naming the variants in the tag's variant part.

Variant Record Variables

Variables that are variant records can either have bound variant types or unbound variant types:

r: Account; -- *unbound variant type*

rSavings: savings Account; -- *bound variant type*

rChecking: checking Account; -- *bound variant type*

Variant Part, Tag Access

The variant part of a record and its tag can only be assigned with a constructor.

```
r ← [balance: 100.0, specifics: checking [charges: 0.5]];
```

```
r ← [100.0, savings[7.0,35]];
```

Assigning the variant part of a record has the side effect of assigning the tag.

Accessing Components of Variants

When a record is a bound variant, components of its variant part may be accessed as if they were common components:

```
rSavings.intRate ← 8.9;
```

```
rChecking.charges ← 1.0;
```

With unbound variants, a generalized SELECT statement must be used for actual tags:

```
WITH r SELECT variantType FROM  
  savings => {  
    intRate ← 8.8;  
    term ← 90;  
  };  
  checking => charges ← 1.0;  
ENDCASE;
```

For computed and overlaid tags, an expression must be supplied yielding a tag value:

```
WITH r SELECT (IF ... THEN savings ELSE checking) FROM  
  savings => ...  
  checking => ...  
ENDCASE;
```

Accessing Components of Variants

For overlaid tags, a field in a variant that appears in no other variant can be referenced directly:

```
r.intRate ← 8.8;  
...  
r.charges ← 1.0;
```

Array Descriptors

An array descriptor describes the location and length of an array. There are three operators that are relevant to array descriptors.

LENGTH[<an array>] -- yields the number of array elements
BASE[<an array>] -- yields a pointer value for locating the first element
DESCRIPTOR[<an array>] -- yields an array descriptor (record of base and length)

Example:

```
history: DESCRIPTOR FOR ARRAY OF CARDINAL;  
numbers: ARRAY [0..1000) OF CARDINAL;  
history ← DESCRIPTOR[numbers];
```

Descriptors historically were used for dynamic arrays and are present mainly for backward compatibility. They are used for parameter passing now. Sequences (described later) are now used for dynamic arrays instead and are the recommended method.

Base and Relative Pointers

Relative pointers require the addition of a base pointer to obtain an absolute pointer. This allows data structures with internal references that are independent of memory location.

They are used with movable storage areas called ZONEs. Consult the MLM and the PPM for details.

The Debugger

The Debugger

Some of the material in this section overlaps the material covered in the XDE tutorials that you should have already completed. This is desirable because:

- 1) some students will not have gone through the tutorials, thus making this lab their first exposure to the debugger.
- 2) the extra practice can't hurt.

CoPilot vs. Sword

There are two different debuggers depending on what version of software you are running. If you are running a pre 12.3 release, then you will be using the **CoPilot** debugger, which is built into the CoPilot bootfile.

If you are running at 12.3, then you have the option of using the built-in CoPilot debugger or using the new **SWorD** debugger, which is a separate application that must be run on a Tajo bootfile.

CoPilot will not be available for versions newer than 12.3.

The information in this lecture applies to both the CoPilot debugger and to SWorD.

The Debugger

- Interactive, sourceline-level
- Resides on an XDE volume.
The debugger need not reside on the same machine as the client (e.g. Remote debugging).
- Allows inspection of run-time stack, variables, processes, etc.; variables can be re-assigned, individual procedures can be called.
- What will be covered:
 - Common command usage
 - What the programmer sees
- What won't be covered:
 - Low-level and other operating system-related details

Getting the Debugger Window

If you are using the CoPilot debugger, then you do not run anything special. The Copilot debugger is built into the CoPilot bootfile. Look for a window labelled Debug.log, which when opened looks like this:

```

Debug.log
9-Nov-87 12:15
>

```

If you are using the Sword debugger, then you must run Sword.bcd. Sword requires a Tajo bootfile. Look for a window labelled Debug.log, which when opened looks like this:

```

LocalWorld (Debug.log2)
go: {proceed, abort, kill, screen, start}      client: {local}      destroy!
read: {}      write: {}      processes configs      attach: {source, symbols}
source!      findModule!      rep?!      showType!      type&bits!      options!
break: {set, clear, clearall, list, attachCond, attachKey}      watch: {off}
LocalWorld (Debug.log2)
Interpreter
>

```

Sword can have multiple windows. To get another window:

- 1) In 12.3, execute the "New Interpreter" command in the stack of root menus, or
- 2) In 14.0, execute the "Another!" command in the Sword form subwindow.

Reasons for Entering the Debugger

- Interrupt
- Breakpoint
- Map Log
- Error
 - Address Fault
 - Uncaught Signal

Commands for When in Doubt

- {zero or more characters} ?:
Displays all the commands that begin with the specified characters.
- the DELETE key:
Allows you to cancel any command before it's been invoked.
- the STOP key:
Allows you to cancel any command during its execution.

Commands to Inspect Modules

- **SEt Module Context:**
Sets the module context to the user-specified module. The variables and procedures in the current module are recognized.
- **Display Stack:**
Displays the top of the run-time stack, which is the procedure currently being executed. Also puts you in a subcommand mode to learn more about each element on the stack.

Subcommand Mode (after a Display Stack)

- source:
displays the source line where the debugger was invoked and loads the file in a window if it isn't loaded already. The source file is scrolled so that the line in question is in view.
- variables:
displays the local variables and the parameters for that procedure
- global variables
displays the global variables for that module
- parameters:
displays the parameters for that procedure
- next:
displays the next procedure call on the run-time stack
- back:
goes back up the stack (reverse of next)
- quit (or the DELETE key):
gets you out of the subcommand mode

Commands to Inspect Processes

- **List Processes** (also available in the Sword FormSW):
Lists the currently active processes in the client volume.
- **SEt Process Context:**
Sets the current process context to the user-specified process. The call stack for that process is recognized.

Commands Related to Swapping

- Proceed (also available in the Sword FormSW):
Continues execution of the program by going back to the client volume and executing from the point where the debugger was invoked.
- Quit (also available in the Sword FormSW as 'abort'):
Raises the signal ABORTED in the process that entered the debugger, which usually results in that process being deleted.
- Uscreen (also available in the Sword FormSW as 'screen'):
Applies only to world-swap debugging. Displays the screen bitmap of the client volume for 20 seconds, or less if the "STOP" key is toggled.
- ReDisplay swap reason :
Redisplays the reason why the debugger was called. Sometimes the user can give the debugger more information and then redisplay the swap reason to learn more about why the swap occurred.

Setting Breakpoints

- **B**reak **E**ntry procedure:
Sets a breakpoint at the point of entry to a procedure
- **B**reak **X**it procedure:
Sets a breakpoint at the point of exit of a procedure.
- To set a breakpoint in source other than at the beginning or end of a procedure, load the source file into a window, make a selection on the line where the breakpoint is to be set. Then,
 - 1) with Sword hit the "set" command of the "break:" enumerated within the FormSW.
 - 2) With CoPilot, click "Break" in the window menu.

A breakpoint will then be set at the beginning of the statement in which the selection was made.

Some Other Breakpoint Commands

- **List Breaks** (also available in the Sword FormSW):
Lists all the breakpoints that are currently set.
- **Clear All Breaks** (also available in the Sword FormSW):
Clears all currently set breakpoints.
- **Clear Break #** (also available in the Sword FormSW):
Clears the break that you specify by number.
- **Attach Condition** (also available in the Sword FormSW):
Attaches a boolean condition that must be TRUE in order for the break to occur.
- **Attach Keystrokes** (also available in the Sword FormSW):
Attaches a user-supplied string of characters to be entered into the debugger when the associated breakpoint is executed.

Using the Interpreter Mode

- Interpreter mode handles a subset of Mesa.
- In the interpreter mode, you can display and re-assign variables (simple or complex), dereference pointers, call procedures, and convert types.
- Interpreter mode is invoked by typing a space character at the beginning of a line. After typing the space, you can type a variable name to inspect that variable.
- To dereference a pointer variable, type the variable name followed by an up-arrow ("foo↑"). To dereference a pointer to a pointer, type the variable name followed by 2 up-arrows ("foo↑↑").

Setting the Options

The Debugger Options window allows you to change the default format that the debugger uses in displaying values of variables as well as specify (for Sword only) whether or not a particular kind of event is handled locally.

Invoke the **options!** command in the FormSW to bring up the Options window.

There are four booleans at the bottom of the window. If a boolean is TRUE and the corresponding event occurs, it is handled locally. If a boolean is FALSE then the event causes a world-swap. The booleans map to:

fault = address fault, uncaught = uncaught signal,
 break = breakpoint, calldebug = SHIFT-STOP

```

LocalWorld (Debug.log2)
go: {proceed, abort, kill, screen, start}      client: {local}      destroy!
read: {}      write: {}      processes configs attach: {source, symbols}
source!      findModule!      rep?!      showType!      type&bits!      options!
break: {set, clear, clearall, list, attachCond, attachKey}      watch: {off}

```

```

LocalWorld (Debug.log2)
Interpreter
>

```

```

Interpreter Options
CARDINAL: {octal, decimal, hex} signed      INTEGER: {octal, decimal, hex} signed
Apply!  POINTER: {octal, decimal, hex}      PROCESS: {octal, decimal, hex}
Abort!  RELATIVE: {octal, decimal, hex}      UNSPECIFIED: {octal, decimal, hex}
Array elements = 10      String length = 150
filter:
fault  uncaught  break  calldebug  processes  configs

```

Be sure to invoke **Apply!** to have the changes take effect or **Abort!** to restore them to the previous options. (Do not just tiny or deactivate the options window.)

Show Type

Show Type allows you to find out the type of various procedures and variables. It operates on the current selection (anywhere on the screen). The format must be:

InterfaceName.symbolName

If only the Interface name is selected, then all of the types in that file are displayed.

LocalWorld (Debug.log2)

go: {proceed, abort, kill, screen, start}	client: {local}	destroy!
read: {}	write: {}	processes configs attach: {source, symbols}
source!	findModule!	rep?! showType! type&bits! options!
break: {set, clear, clearall, list, attachCond, attachKey}		watch: {off}

LocalWorld (Debug.log2)
 Interpreter
 >
 Exec.AddCommand: PROCEDURE [name: LONG STRING, proc: Exec.ExecProc, help:
 Exec.ExecProc ← NIL, unload: Exec.ExecProc ← Exec.DefaultUnloadProc, clientData:
 LONG POINTER ← NIL];

File: <CoPilot>WD>FactorialToolImpl.mesa

Create OPos Edit Load Empty Save Time Store Reset Split Match Destroy

A!!	S!	RS!	<:	SR!	R!	<:
-----	----	-----	----	-----	----	----

```

CreateFactorialTool: PROCEDURE = {
  Exec.AddCommand["FactorialTool.~"L, NoOp, NIL, Unload];
  wh ← Tool.Create[
    makeSWsProc: MakeSWs, initialState: default,
    clientTransition: ClientTransition, name: "Factorial Tool"L];
    
```

Processes Boolean (in Sword Only)

Turning on the **processes** boolean in the Sword FormSW creates a proceses subwindow. This subwindow contains processes, call stacks, and local variables. You can zoom or close a particular line by selecting the cross at the head of the line. Zooming displays more detail; for instance, zooming a stack frame line displays the local variables of the stack frame.

```

LocalWorld (Debug.log2)
go: {proceed, abort, kill, screen, start}      client: {local}      destroy!
read: {}      write: {}      processes configs attach: {source, symbols}
source!      findModule!      rep?!      showType!      type&bits!      options!
break: {set, clear, clearall, list, attachCond, attachKey}      watch: {off}
┌
x PSB: 53B, waiting CV, L: 457054B†, PC: 18936 (in TTYSWsA, G: 431674B†)
x PSB: 54B, waiting CV, L: 406010B†, PC: 18936 (in TTYSWsA, G: 431674B†)
x PSB: 55B*, waiting CV, L: 455304B†, PC: 2896 (in ITInstall, G: 517414B†)
x PSB: 56B, waiting CV, L: 405170B†, PC: 18936 (in TTYSWsA, G: 431674B†)
  x No symbols for L: 405170B†, PC: 18936 (in TTYSWsA, G: 431674B†)
    x No Variables!
  x No symbols for L: 406410B†, PC: 204 (in TTYImpl, G: 440244B†)
x PSB: 66B, waiting CV, L: 456114B†, PC: 89 (in TTYImpl, G: 440244B†)
x PSB: 67B, waiting CV, L: 414254B†, PC: 89 (in TTYImpl, G: 440244B†)
x PSB: 117B, waiting CV, L: 403040B†, PC: 389 (in Processes, G: 417674B†)
  x No symbols for L: 403040B†, PC: 389 (in Processes, G: 417674B†)
  x No symbols for L: 405664B†, PC: 268 (in ClockToolImpl, G: 523314B†)
x PSB: 124B, waiting CV, L: 455354B†, PC: 18936 (in TTYSWsA, G: 431674B†)
└
LocalWorld (Debug.log2)
Interpreter
>

```


Configs Boolean (in Sword Only)

Turning on the **configs** boolean creates a configs subwindow. This subwindow contains configurations, modules, and global variables. You can zoom or close a particular line by selecting the cross at the head of the line. Zooming a configuration line, for instance, displays the nested configurations and modules.

```

LocalWorld (Debug.log2)
go: {proceed, abort, kill, screen, start}      client: {local}      destroy!
read: {}      write: {}      processes configs      attach: {source, symbols}
source!      findModule!      rep?!      showType!      type&bits!      options!
break: {set, clear, clearall, list, attachCond, attachKey}      watch: {off}

x Activity
x MailTool
x RemoteExec
x AddHintMenus
x FileTool2
  x FileToolImplA, G:511270B No symbols.
    x No Variables!
  x FileToolImplB, G:511500B No symbols.
  x FileToolImplC, G:511564B No symbols.
x Sword
x ExpandType
x OriginalPosition
x SourceTime

LocalWorld (Debug.log2)
Interpreter
>

```

PROPS-STOP

Sometimes your machine will "hang", especially when you are locally debugging some program and it crashes. PROPS-STOP is a special command that is built into 12.3 (and later) Tajo bootfiles. This will create a new notifier to let you continue what you were doing.

PROPS-STOP will usually work when a particular operation is hung but not when the entire workstation is hung. (Look at the clock to see if everything is hung.) When PROPS-STOP does not work, you will have to re-boot.

CoPilot bootfiles do not have this feature.

Summary

This lecture was not complete coverage of all commands that are available in the debugger. It was meant to give you an introduction to using the debuggers. There are many more ways to use the debugger to help you in your development work.

For more information about the CoPilot debugger refer to the Debugger chapter in the XDE User's Guide.

For more information about using Sword, refer to Appendix E of the XDE User's Guide.

Appendix: Conformance and Type Determination

Conformance Among Numerics

INTEGER, NATURAL, CARDINAL	conform to INTEGER
INTEGER, NATURAL, CARDINAL	conform to CARDINAL
INTEGER, NATURAL, CARDINAL	conform to NATURAL
INTEGER, NATURAL, CARDINAL LONG INTEGER, LONG CARDINAL	conform to LONG INTEGER
INTEGER, NATURAL, CARDINAL LONG INTEGER, LONG CARDINAL	conform to LONG CARDINAL
INTEGER, NATURAL, CARDINAL LONG INTEGER, LONG CARDINAL, REAL	conform to REAL

Examples:

i: INTEGER; c: CARDINAL; n: NATURAL;
li: LONG INTEGER; lc: LONG CARDINAL; r: REAL;

--The right side conforms to the left side.

i ← c;
lc ← n;
c ← i;
n ← c;
r ← li;

Conformance Using Range Assertions

LONG INTEGER, LONG CARDINAL	conform to INTEGER
LONG INTEGER, LONG CARDINAL	conform to CARDINAL
LONG INTEGER, LONG CARDINAL	conform to NATURAL

Examples:

```
i: INTEGER;  c: CARDINAL
1i: LONG INTEGER;  1c: LONG CARDINAL;
```

-- *The right side conforms to the left side.*

```
i ← INTEGER[1i];
c ← CARDINAL[1i];
i ← INTEGER[1c];
```

Word Length Rules For Expressions (Balancing)

In determining what type of operation (INTEGER, CARDINAL, LONG INTEGER, LONG CARDINAL, REAL) should be performed in an expression, a common word length must first be found.

In general, the operation requiring the fewest automatic type conversions will be the one used. So for numerics:

1. If all (both) operands are short numerics, a short numeric operation will be used.
2. If all (both) operands are long numerics, a long numeric operation will be used.
3. If one operand is a long numeric, the other operand will be lengthened and a long operation will be used.
 - a. When an INTEGER is lengthened, its inherent type is LONG INTEGER.
 - b. When a CARDINAL or NATURAL is lengthened, its inherent type is LONG INTEGER *and* LONG CARDINAL.
4. If one operand is a REAL the other operand is converted and a REAL operation is used.

Determination of Representation (Balancing)

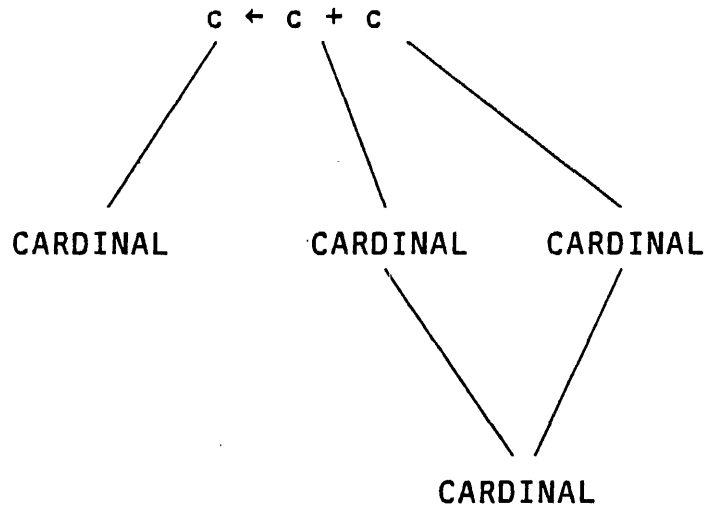
Once a common word length is found for an expression, the following rules will choose whether a signed (INTEGER, LONG INTEGER) or unsigned (CARDINAL, LONG CARDINAL) operation will take place.

1. If the operands have exactly one common inherent representation, the operation defined for that representation is selected (and the target representation is ignored).
2. If the operands have no common inherent representation but the target representation is well-defined, the operation yielding that representation is chosen.
3. If the operands have both inherent representations in common, and if the target representation is well-defined, The representation of the target type selects the operation.
4. If the operands have both inherent representations in common but the target representation is ill defined, the *signed* operation is chosen.
5. Unary minus converts its argument to a signed representation if necessary and produces a signed result.

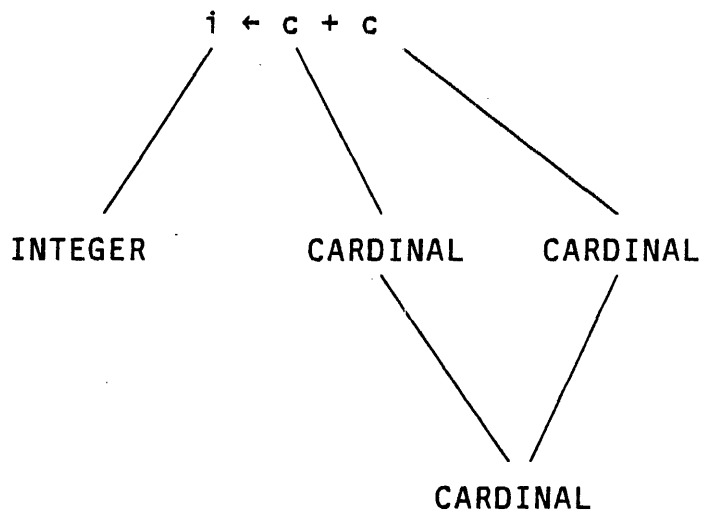
If the operands have no representation in comon and the target representation is ill-defined, the expression is in error.

Type Determination Examples

i: INTEGER; c: CARDINAL; n: NATURAL;
li: LONG INTEGER lc: LONG CARDINAL;

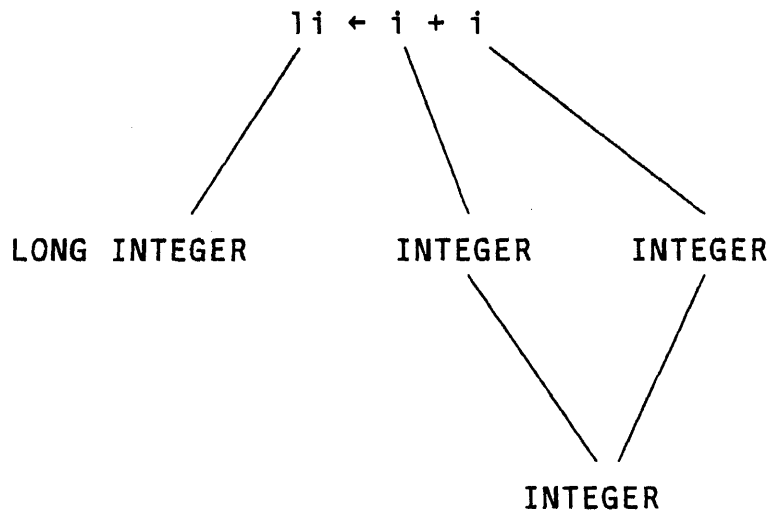


Length rule: 1
Sign rule: 1

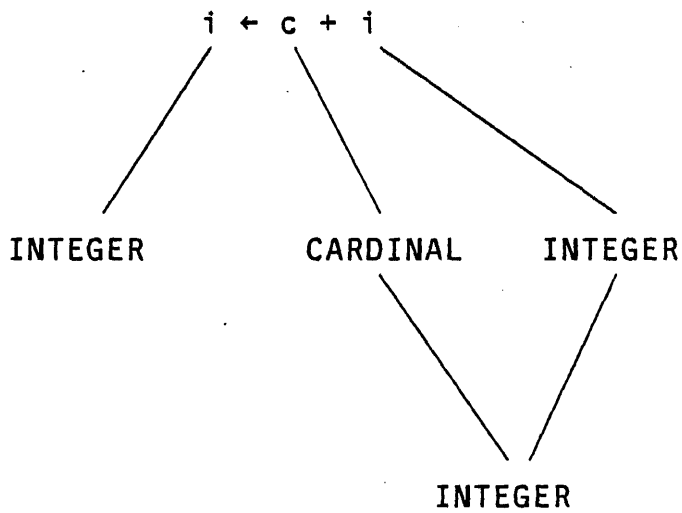


Length rule: 1
Sign rule: 1

Type Determination Examples

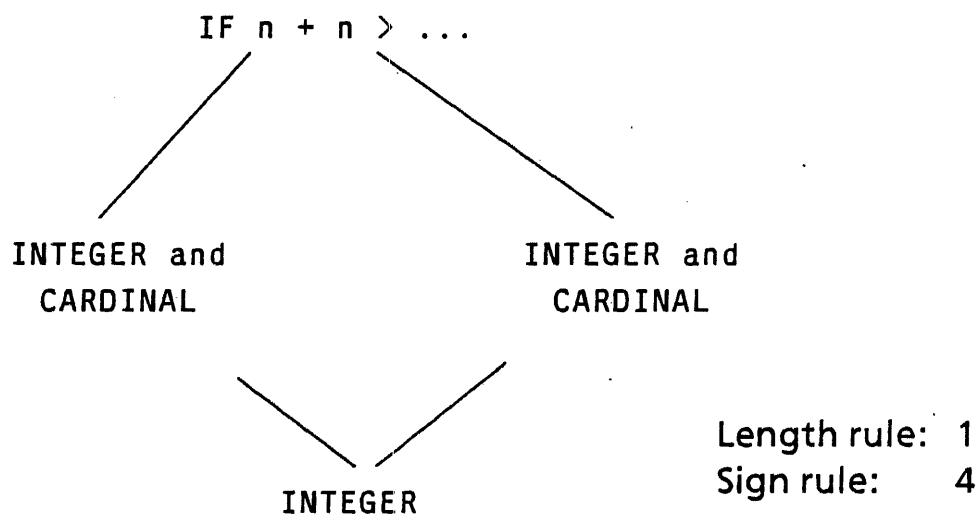
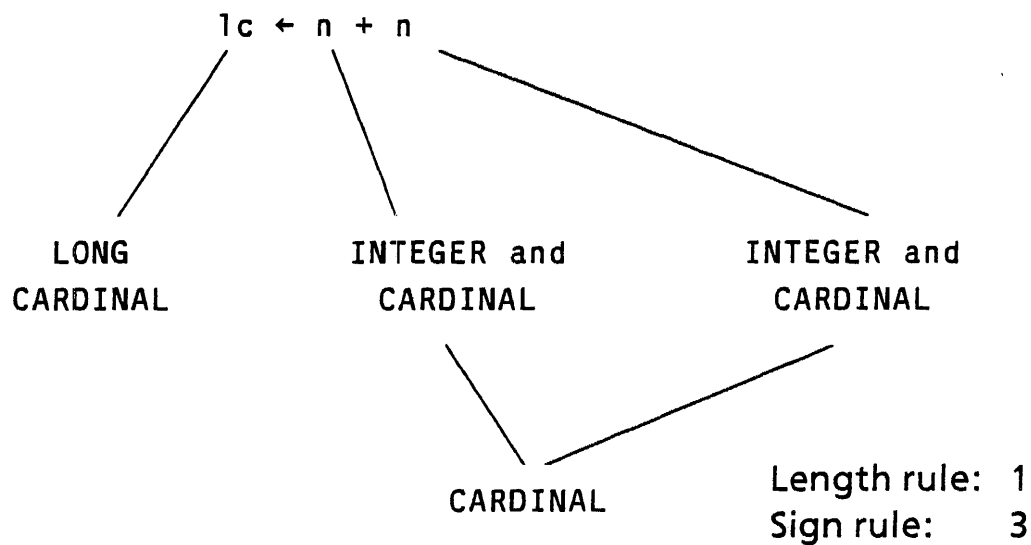


Length rule: 1
Sign rule: 1

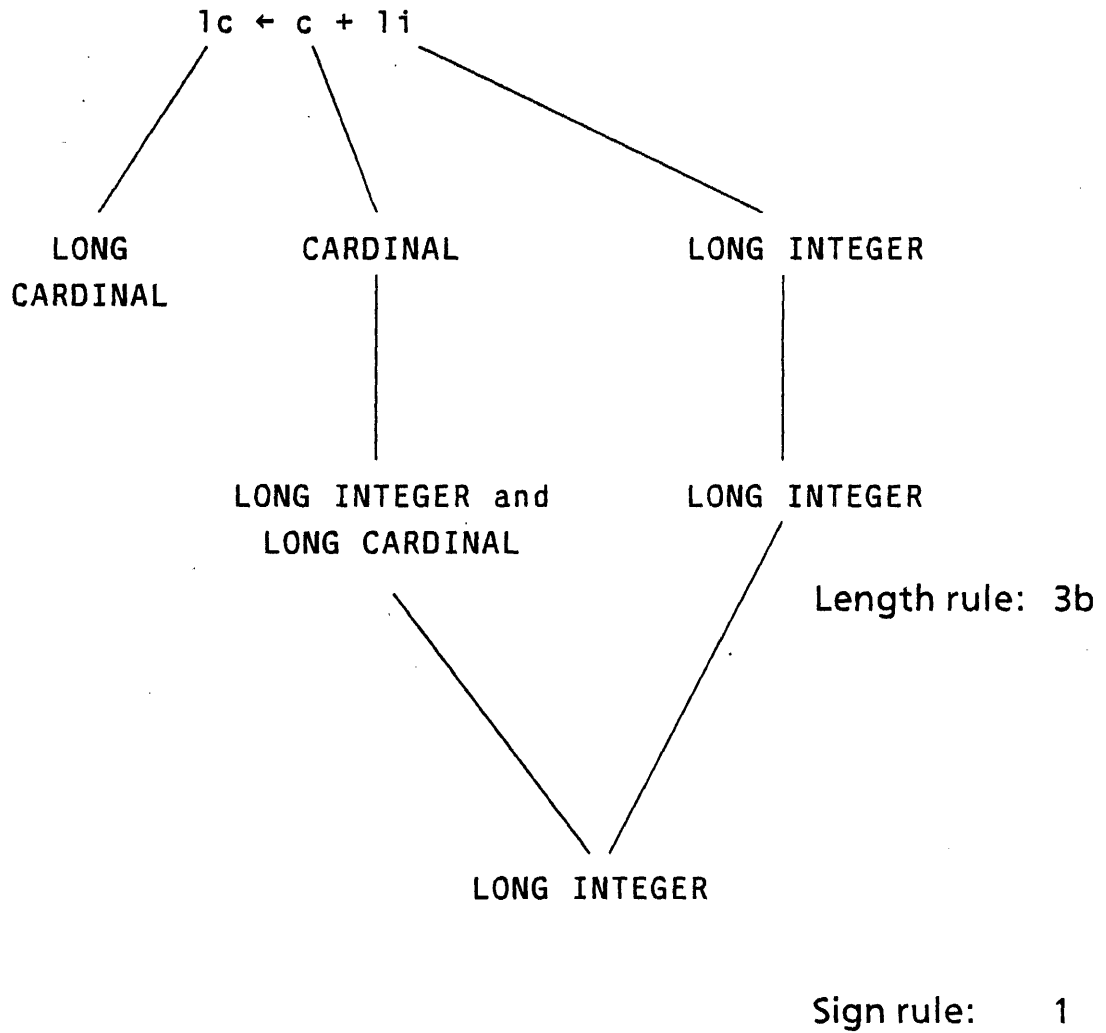


Length rule: 1
Sign rule: 2

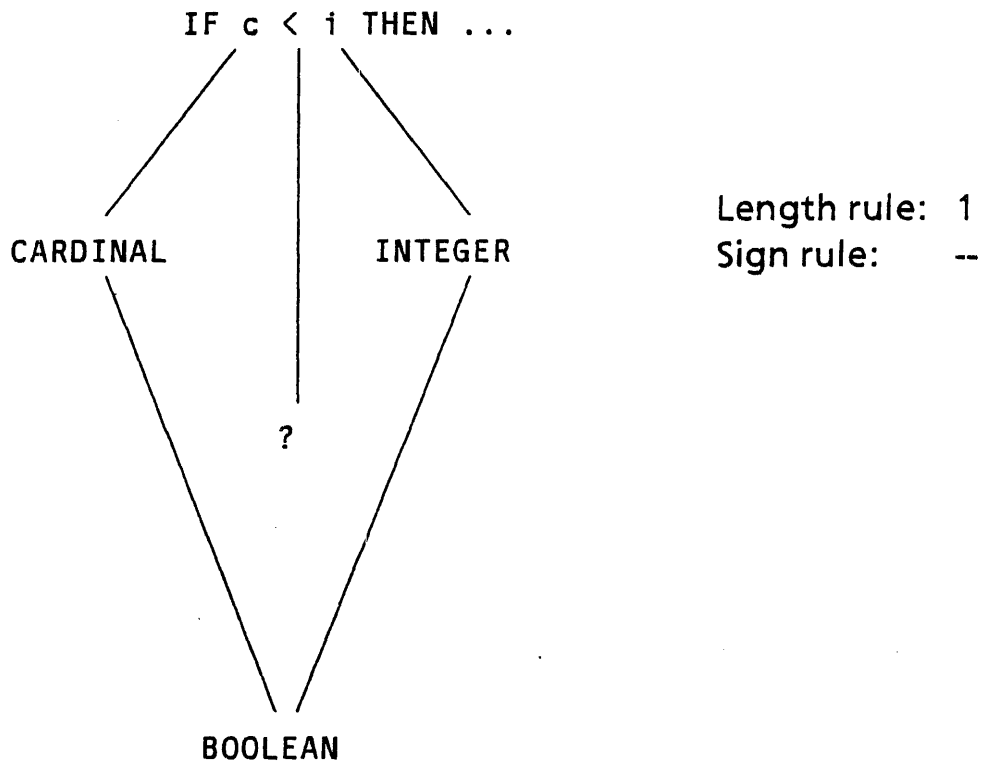
Type Determination Examples



Type Determination Examples



Example of Incorrect Statement

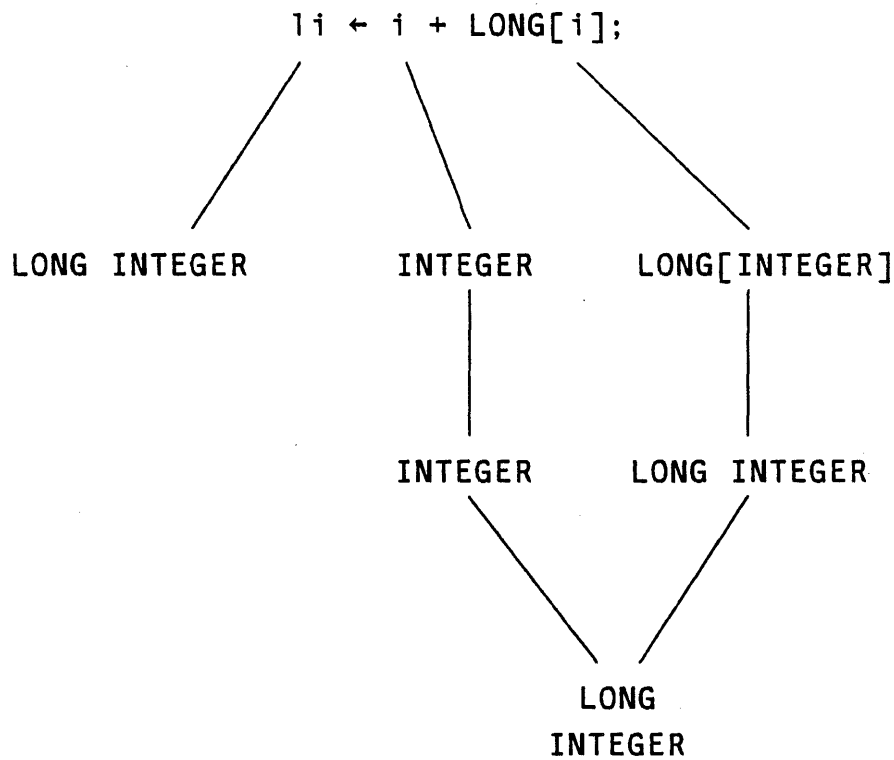


There is no target type; Mesa cannot determine if you want a signed < operation or an unsigned < operation.

The Long Operator

Use the LONG operator to force an operand to be lengthened according to the previous length rules.

Examples:



Length rule: 3a
Sign rule: 1

INDEX

A	
Array	1-33 - 1-34, 1-59, 1-76
Arrays	1-27, 1-32
Assignment	1-27, 1-44
B	
Balancing	1-22, 1-24
bcd	1-3, 1-7
Binder	1-7
Blocks	1-27, 1-50
BOOL	1-12
BOOLEAN	1-15, 1-30, 1-40, 1-64
Bounds	1-25, 1-49
C	
CARDINAL	1-12 - 1-13, 1-15, 1-20, 1-25 - 1-26, 1-29 - 1-30, 1-54
CHAR	1-12
CHARACTER	1-19 - 1-20, 1-30, 1-32, 1-64, 1-68
Client	1-4 - 1-5, 1-7
comment	1-10
Comments	1-2, 1-10
config	1-7
Configuration	1-4, 1-7
Conformance	1-27 - 1-29, 1-31, 1-34, 1-39, 1-42, 1-66
D	
Declarations	1-15
Definitions	1-4 - 1-6
Descriptors	1-59, 1-76
DIRECTORY	1-8 - 1-9
E	
Enumerated	1-29
EXIT	1-57 - 1-58
F	
Fields	1-35 - 1-36
FIRST	1-20, 1-31
floating-point	1-14

G

GOTO 1-51, 1-57 - 1-58

I

Identifiers 1-11

IEEE 1-14

IF 1-10, 1-27, 1-29, 1-45 - 1-46, 1-49 - 1-51, 1-58, 1-68

Implementation 1-4, 1-6 - 1-7

Inline 1-67

INT 1-12 - 1-13

INTEGER 1-12 - 1-13, 1-19 - 1-20, 1-25 - 1-26, 1-30, 1-54

Interfaces 1-4

L

LAST 1-20

Loop 1-27, 1-52, 1-57 - 1-58

M

MOD 1-16, 1-21, 1-60

Modules 1-3 - 1-7

N

NAT 1-12

O

Operators 1-16 - 1-18, 1-20 - 1-21, 1-26, 1-69

ORD 1-20, 1-29

P

Pointer 1-41 - 1-42

Pointers 1-27, 1-40, 1-43, 1-59, 1-77

Precedence 1-21

PRED 1-20

Procedures 1-60 - 1-62, 1-64 - 1-65, 1-67

Program 1-2, 1-4 - 1-9, 1-11, 1-64

R

REAL 1-14 - 1-16, 1-65, 1-71

Record 1-37 - 1-39, 1-71 - 1-72

Records 1-27, 1-35, 1-59, 1-70

Return 1-63

S

Scoping 1-64
SELECT 1-27, 1-47 - 1-49, 1-63, 1-71, 1-74
STRING 1-68 - 1-69
Strings 1-59, 1-68 - 1-69
Subrange 1-30 - 1-31
SUCC 1-20

U

underflow 1-26

V

VAL 1-20, 1-29
Variant 1-59, 1-70 - 1-73
Variants 1-74 - 1-75

**Modules:
Definitions, Programs, and Configurations**

Outline

1. Interfaces
 - a. What interfaces are all about
 - b. Definitions modules
 - c. Program modules
 - i. Clients
 - ii. Implementations
 - d. Configurations

Programming Without Interfaces

This code can not be shared with any other program.

```
--DoLittle.mesa
DoLittle: PROGRAM = {
  var: CARDINAL;

  var ← 0;
  THROUGH [1..100] DO
    var ← var + 5;
  ENDLLOOP;
}.
```

DoLittle.bcd

Execute: DoLittle.bcd

Types of Mesa Modules

There are two types of Mesa modules: DEFINITIONS and PROGRAM.

DEFINITIONS and PROGRAM modules are both written in Mesa. They are input to the compiler. The output of the compiler is a binary configuration description (bcd) file.

When a module is compiled a time stamp (right down to the second) is included in the bcd file. This time stamp is what differentiates multiple versions of the same file.

A DEFINITIONS module is commonly referred to as an **Interface**.

Overview

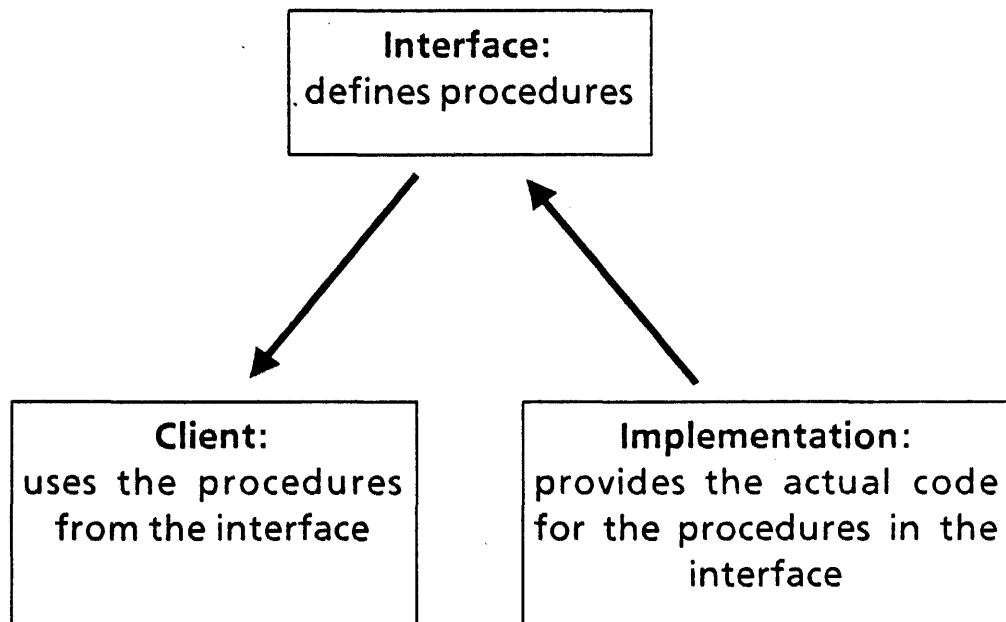
There are three basic pieces to the interface mechanism:

An *interface*, which defines an abstraction.
(An interface is a module of type DEFINITIONS.)

A *client*, which uses the facilities defined in the interface.
(A client is a module of type PROGRAM.)

An *implementation*, which provides the facilities defined in the interface.
(An implementation is a module of type PROGRAM.)

There can be more than one client of an interface, and the implementation is not necessarily a single module.



Definitions Module

Interfaces (DEFINITIONS modules) generally define the data structures and operations for an abstraction. Interfaces contain only declarations -- *no executable code*.

Here is a typical interface module:

```
--Example.mesa
Example: DEFINITIONS = {
  limit: INTEGER = 86;
  Range: TYPE = [-limit..limit];
  ...
  ReadChar: PROCEDURE RETURNS[input:CHARACTER];
  WriteChar: PROCEDURE[output: CHARACTER];
}.
```

Compile-Time Items and Run-Time Items

The items in a DEFINITIONS module fall into two classes:

compile-time items: definitions of TYPEs and constants

run-time items: definitions of procedures, signals, programs,
and other variables

In the example on the previous page, the first two items are compile-time items; the two procedures are run-time items.

Note: In the Mesa Language Manual, compile-time items are referred to as *non-interface elements*, and run-time items are referred to as *interface elements*.

Directory Statement

If any module uses information from an interface, that interface *must* be listed in the module's DIRECTORY clause. The Directory clause is the first thing in any module.

Note: ONLY DEFINITIONS modules are listed in a DIRECTORY clause.

Example:

```
--SomeModule.mesa
```

```
DIRECTORY
```

```
  Interface1,
```

```
  Interface2,
```

```
  Interface3;
```

```
SomeModule: DEFINITIONS = {
```

```
  ...
```

```
};
```

USING Clause

The USING clause is an optional part of the DIRECTORY statement. It is used to list the names of all of the items that will be used from the referenced interface. A USING clause is not required, though it is recommended. Although, if you do have a USING clause, then you must list each and every symbol that you want to use from that interface in the USING clause.

Examples:

```
--SomeModule.mesa
DIRECTORY
  Interface1 USING [item1, item2, item3];  -- recommended method
```

OR:

```
--Table.mesa
DIRECTORY
  Interface1;
```

Referencing Items from an Interface

To reference the item from the interface in the program body, you must put the Interface name first then the symbol name (separated by a period), i.e. **Interface1.item1**.

Example:

```
--Simple.mesa
Simple: DEFINITIONS = {
  limit: INTEGER = 86;
  Range: TYPE = [-limit..limit];
  Pair: TYPE = RECORD[first, second: Range]; -- want to use this item
  PairPtr: TYPE = LONG POINTER TO Pair;
}.
```

```
--Table.mesa
DIRECTORY
  Simple USING [Pair];

Table: DEFINITIONS = {
  limit: INTEGER = 256;
  Index: TYPE = [0..limit);
  PairTable: TYPE = ARRAY Index OF Simple.Pair;
}.
```

Program Modules

A program module can access the *compile-time items* from a definitions module as easily as a definitions module.

```
--TableUser.mesa
DIRECTORY
  Simple USING [Pair],
  Table USING [PairTable];

TableUser: PROGRAM = {
  pair: Simple.Pair;
  table: Table.PairTable;
  ...
}.
```

Note that you *must* qualify identifiers from definitions modules with the name of the DEFINITIONS module.

Summary: Using Compile-Time Items

To be a client of an interface, you need to know the name of the interface and the name of the symbols that you want to use from that interface.

To access *compile-time* items from an interface, a client module must do three things:

- Include the name of the *interface* in its **DIRECTORY** clause.
- Include the name of the *symbol* in the **USING** clause.
- Reference the symbol as *InterfaceName.Symbol*.

IMPORTS Clause

The Imports clause lists the *Interfaces* for which there needs to be an implementation supplied from somewhere. For example, procedures (and other run-time items) are defined in an interface but are implemented elsewhere. The system needs to know that it needs to match up the implementation of the procedure with the caller of that procedure.

The syntax of using *run-time items* is like that of using compile-time items except that you must also include the interface in the IMPORTS list.

Note: Only DEFINITIONS modules are listed in an IMPORTS clause.

```
--IO.mesa
IO: DEFINITIONS = {
  ReadChar: PROCEDURE RETURNS[input: CHARACTER];
  ReadLine: PROCEDURE RETURNS[input: LONG STRING];
  WriteChar: PROCEDURE[output: CHARACTER];
  WriteLine: PROCEDURE[output: LONG STRING];
}.
```

```
--CopyDriver.mesa
DIRECTORY
  IO USING [ReadLine, WriteLine];

CopyDriver: PROGRAM IMPORTS IO = {
  input: LONG STRING ← [256];
  DO
    input ← IO.ReadLine[];
    IF input[0] = '. THEN EXIT;
    IO.WriteLine[input];
  ENDLLOOP;
}.
```

Summary: Using Run-Time Items

To access *run-time items* from an interface, a client module must:

- Include the name of the *interface* in its **DIRECTORY** clause.
- Include the name of the *symbol* in the **USING** clause.
- Include the name of the *interface* in its **IMPORTS** clause.
- Reference the symbol as *InterfaceName.Symbol*.

Note: You do not need to know anything about the *implementation* of the procedures in the interface. You just take on faith that if something is advertised in an Interface, you can use it.

Compilation Order

The DIRECTORY clause lists all the interfaces from which you are using information. To type-check your program, the compiler must be able to read all of the interfaces listed in the DIRECTORY clause. This means that when you compile a program, the *compiled* version of all the interfaces in the DIRECTORY clause must be on your local disk.

```
--StringClient.mesa
DIRECTORY
  String USING [Equivalent];

StringClient: PROGRAM IMPORTS String = {
  ...
  sameString: BOOLEAN ← String.Equivalent[str1, str2];
  ...
}.
```

Thus, to compile StringClient.mesa, you must have the file **String.bcd** on your local disk. The compiler will include the time stamp of String.bcd in StringClient.bcd. If the referenced Interface is not present, the compiler will give the message "Can't Open String.bcd of ...".

What Exactly Do You Need?

What do you need at compile time and what do you need at run time? This topic usually confuses new Mesa programmers. Hopefully this table will help.

	<u>Examples of Interfaces</u>	<u>Examples of Implementations</u>
	1) System Interface 2) Own Interface	1) System Interface Impl 2) Own Interface Impl
Must it be present on the local disk when client program is <i>compiling</i> ?	1) Yes 2) Yes	1) No 2) No
Must it be present on the local disk when client program is <i>running</i> ?	1) No 2) No	1) Yes 2) Yes
Is it provided in the bootfile?	1) No 2) No	1) Yes 2) No

Example # 1

--Heap.mesa
DEFINITIONS

Heap.bcd
2/8/84

--String.mesa
DEFINITIONS

String.bcd
3/7/84

Note: Since these are system interfaces, you usually only have the object code (bcds), not the source code (mesa).

```

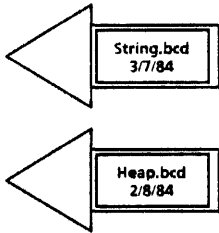
--DoSome.mesa

DIRECTORY
  String USING [...],
  Heap USING [...];

DoSome: PROGRAM
  IMPORTS String, Heap = {

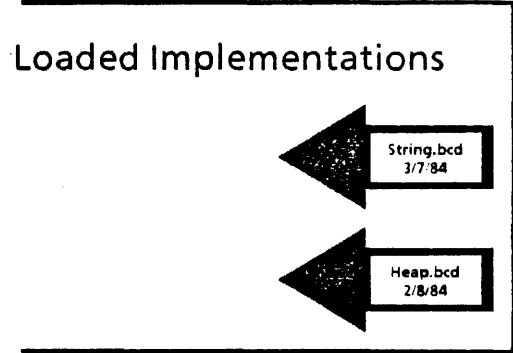
  s1: LONG STRING = "Hi!";
  c: CARDINAL;
  c ← String.Length[s1];
  Heap.Alloc[...];
  ...
  }.

```



DoSome.bcd

Execute: DoSome.bcd



The implementations for String.bcd and Heap.bcd are supplied at run time.
(Note: Since these are system interfaces, the implementation is supplied in the bootfile.)

EXPORTS Clause

The Exports clause lists the *Interfaces* for which this module supplies the implementation for some or all of the run-time items in that Interface.

To EXPORT a procedure (or any other run-time item) to an interface, you need to do three things.

- List the name of the *interface* in the DIRECTORY clause.
- List the name of the *interface* in the EXPORTS list.
- Include the word PUBLIC in the declaration of the procedure.

Example:

```
--IOImpl.mesa
DIRECTORY
  IO;

IOImpl: PROGRAM EXPORTS IO = {
  terminalState: {off, on, hung} ← off;

  ReadChar: PUBLIC PROCEDURE RETURNS[input: CHARACTER] =
    BEGIN
      ...
    END;

  ReadLine: PUBLIC PROCEDURE RETURNS[input: LONG STRING] =
    BEGIN
      ...
    END;
  ...
}...
```

Opaque Types

It is also possible to export the implementation of a type. Opaque types provide a way to hide the internal structure of a type from the client. The implementation can then be changed as needed without affecting the clients of the Interface.

In a *DEFINITIONS* module, you might have:

```
--AnotherDefs.mesa
AnotherDefs: DEFINITIONS = {
  SomeOpaqueType: TYPE;
  Ptr: TYPE = LONG POINTER TO SomeOpaqueType;

  -- Or:

  AnotherOpaqueType: TYPE[<expression>];
    -- where <expression> is a positive integer specifying the word length
}.

```

The *implementation* in the *PROGRAM* module might look like:

```
--AnotherDefsImpl.mesa
DIRECTORY
  AnotherDefs;

AnotherDefsImpl: PROGRAM EXPORTS AnotherDefs = {
  SomeOpaqueType: PUBLIC TYPE = RECORD[v: ...];
}.

```

When the word length is specified, a client using this type can declare variables of this type and perform the standard operations \leftarrow , $=$, and $\#$. Without a known length, no operations can be performed on the type so usually a LONG POINTER to the opaque type is included.

A Program Module Can Be Both Client and Implementor

In this example, TypicalProgram implements DumbDefs, and is a client of DumbDefs and String.

Example:

```
--DumbDefs.mesa
DumbDefs: DEFINITIONS = {
  Language: TYPE = {english, french, german}; -- will use this
  SomeProc: PROC RETURNS[isIt: BOOLEAN]; -- will implement this
}.

--TypicalProgram.mesa
DIRECTORY
DumbDefs USING [Language],
String USING [Equivalent];

TypicalProgram: PROGRAM
  IMPORTS String EXPORTS DumbDefs = {

  lang: DumbDefs.Language ← english;
  string1: LONG STRING ← "Charlie Brown";
  string2: LONG STRING ← "Snoopy";

  SomeProc: PUBLIC PROC RETURNS[isIt: BOOLEAN] = {
    IF (lang = english) AND
      (String.Equivalent[string1, string2]) THEN RETURN[TRUE]
    ELSE RETURN[FALSE];
  };
  ...
  }.
```

Multi-Module Implementations

The implementation of an interface is not necessarily a single module.

```
--ManyProcs.mesa
```

```
ManyProcs: DEFINITIONS = {
  Proc1: PROC [x,y: CARDINAL];
  Proc2: PROC RETURNS [true:BOOLEAN];
  Proc3: PROC[text: LONG STRING];
}.
```

ManyProcs.bcd

```
--ManyProcsAImpl.mesa
```

```
DIRECTORY
  ManyProcs;
```

```
ManyProcsAImpl: PROGRAM
  EXPORTS ManyProcs = {
```

```
  Proc1: PUBLIC PROC[x,y: CARDINAL] = {
    ...
  };
}.
```

ManyProcsAImpl.bcd

```
--ManyProcsBImpl.mesa
```

```
DIRECTORY
  ManyProcs;
```

```
ManyProcsBImpl: PROGRAM
  EXPORTS ManyProcs = {
```

```
  Proc2: PUBLIC PROCEDURE RETURNS[true:BOOLEAN] = {
    ...
  };
```

```
  Proc3: PUBLIC PROC[text: LONG STRING] = {
    ...
  };
```

```
}.
```

ManyProcsBImpl.bcd

Configuration Files

A configuration file groups together clients and implementations so that the implementation is loaded when the client calls it.

Remember:

If you import from System Interfaces -
the implementation is in the bootfile.

If you import from your own interface -
the implementation should be supplied by your program.

The easiest way to ensure that everything will be there when needed is with a *configuration file*.

A configuration file is input to the Binder. The output of the Binder is a bcd file (same as the output of the compiler).

Configuration File Syntax

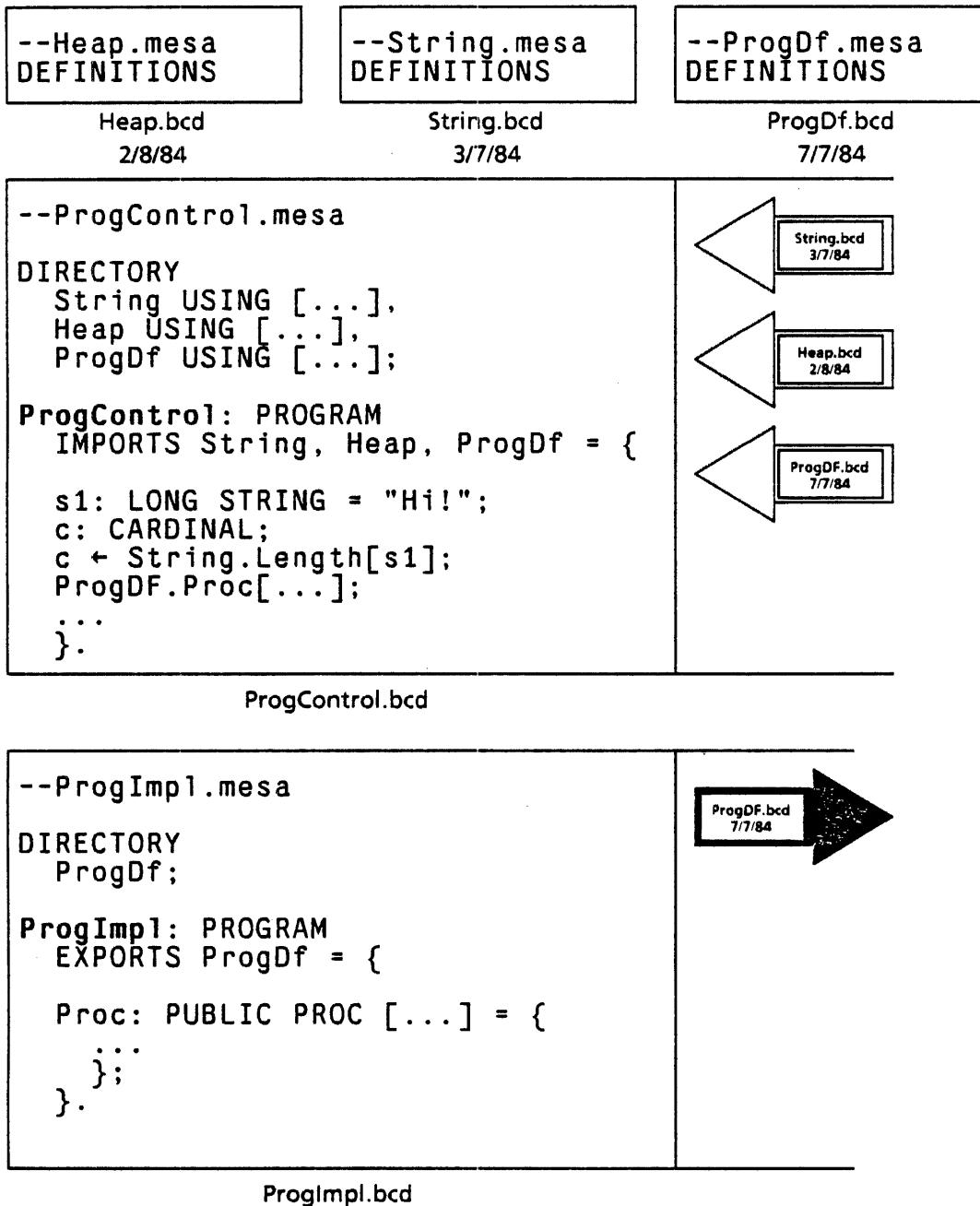
```
--Prog.config
Prog: CONFIGURATION
IMPORTS String, Heap
CONTROL ProgControl = {
    ProgControl;
    ProgImpl;
}.
```

The body (everything between BEGIN and END) should contain the names of all the **PROGRAM** modules that are part of your application.

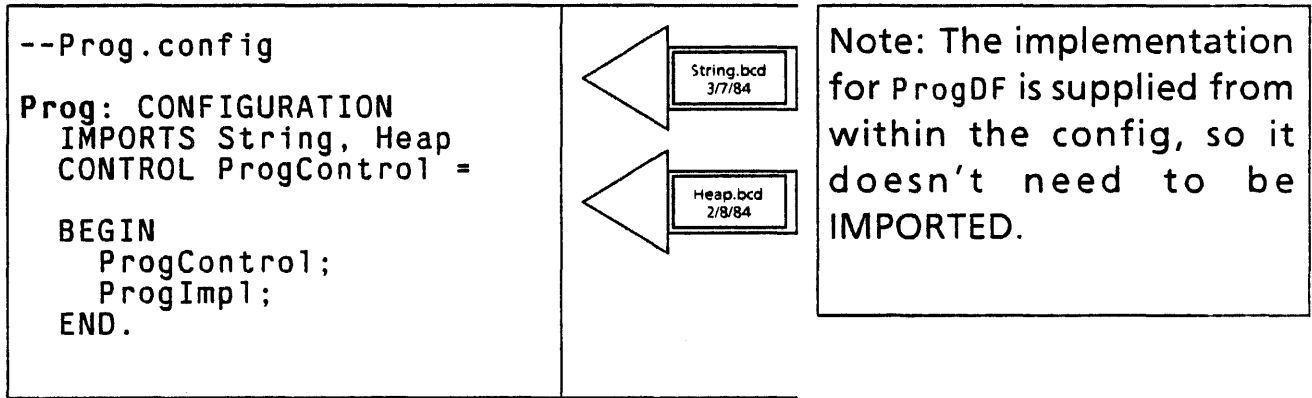
The CONTROL clause should list the PROGRAM module(s) that should be started when this configuration is started. Typically, only one program module will have mainline code; the other program module(s) will just contain procedures.

The IMPORTS list should contain every **DEFINITIONS** module that must be imported from outside the configuration. That is, if any of the included PROGRAM modules need an implementation that is *not* supplied by any of the other PROGRAM modules, then the implementation must be IMPORTED from outside the configuration. More about this on upcoming slides.

Example # 2



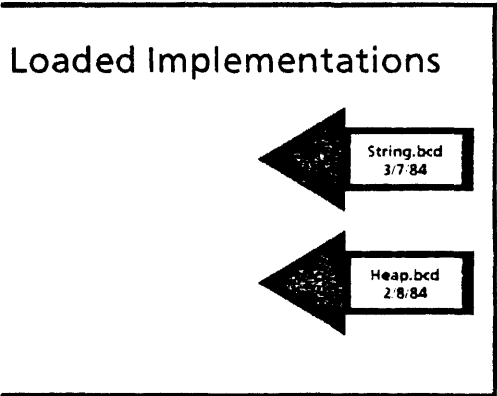
Example # 2 Continued



Prog.bcd

Bind: Prog.config

Execute: Prog.bcd



Implementations for String.bcd and Heap.bcd are bound at run time when Prog is run. (The implementations come from the bootfile.)

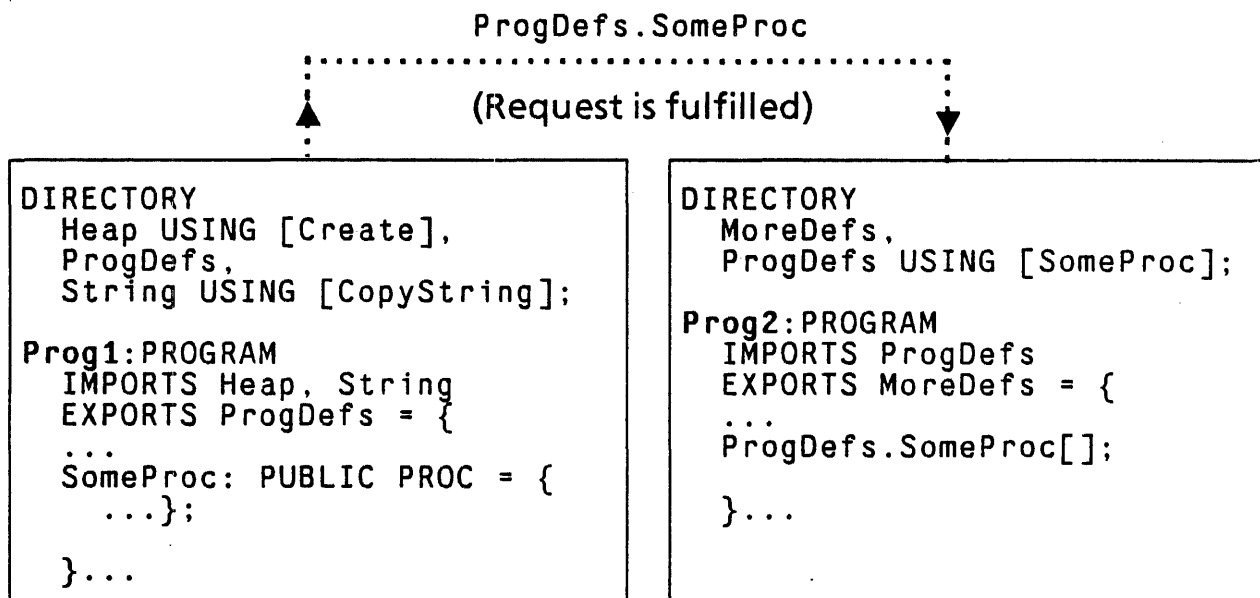
More on IMPORTING Into a Configuration

An imports clause lists interfaces for which their implementations need to be supplied from elsewhere (*outside* the configuration). If one module in a configuration imports certain procedures from an interface, and another module in that configuration exports those procedures, the interface does not have to be imported into the configuration.

```
TwoProgs: CONFIGURATION
  IMPORTS Heap, String

  CONTROL Prog2 = {
    Prog1;
    Prog2;
  }.
```

Note that ProgDefs does not need to be imported into the configuration, since Prog1 supplies the procedure that Prog2 requires.



General Rule in Writing a Configuration

Here are a few simple steps in writing a configuration file. Note that this is to be used a *general rule of thumb*. There are exceptions to this rule. Can anyone think of one?

- 1) List in the body (between the BEGIN - END) all *PROGRAM* modules that make up the configuration.
- 2) Decide which *PROGRAM* modules have mainline code that needs to be started when the configuration is loaded. List these *PROGRAM* modules in start order on the CONTROL line.
- 3) Make a list of all of the IMPORTS of all of the *PROGRAM* modules listed in the body. Make another list of all of the EXPORTS of the *PROGRAM* modules listed in the body. Subtract the list of EXPORTS from the list of IMPORTS. Put the interfaces that remain in the IMPORTS clause of the configuration.

Inter-Modular Type Checking

In a nutshell, the job of the binder is to match up export requests to import requests, and to ensure that type safety is maintained across module boundaries. Recall that when a module references interfaces, the compiler includes the time stamp of the interface as part of the .bcd file. When the binder processes a config file, it reads the compiled versions of the constituent modules to verify that all modules reference the same versions of interfaces.

Exporting From a Configuration File

An implementation for an interface may be given the same status as already loaded implementations (e.g. implementations in the bootfile) by exporting the implementation from the configuration file. This is done by including the *Interface* in an EXPORTS clause in the configuration file:

Example:

```
--Exporter.config

Exporter: CONFIGURATION
  IMPORTS String, Heap
  EXPORTS ExportedInterface
  CONTROL ExportControl = {
    ExportControl;
    ExportImpl;
  }.
```

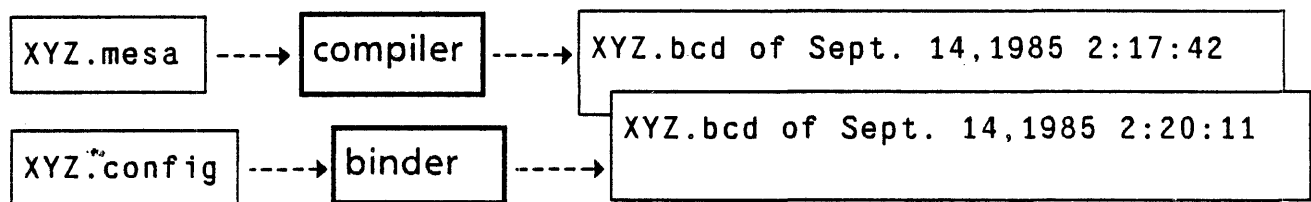
Once `Exporter` is run, the implementation of `ExportedInterface` remains loaded for use by later clients. The loaded state is preserved until the next system boot.

Naming Configuration Files

The compiler and the binder both produce files with the .bcd extension. Thus, the root name of your configuration file must be different from its component modules.

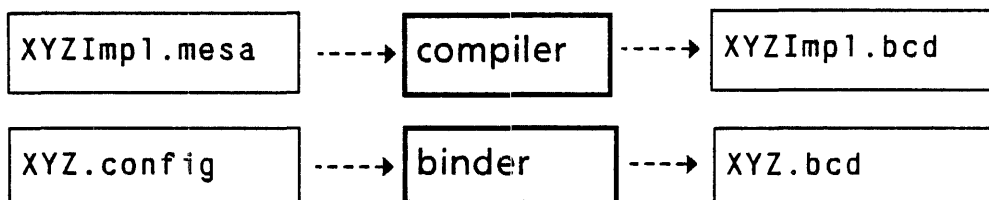
WRONG WAY:

The bound configuration *overwrites* the compiled source code.



RIGHT WAY:

The configuration file and its components have different names, so nothing is overwritten.



Other Naming Conventions

Configuration Modules:

<Bar>.config or <Foo>Tool.config
i.e. Editor.config, MathTool.config

Interfaces (Definitions Modules):

Public Interfaces:

<Foo>.mesa
i.e. Edit.mesa, MT.mesa

Private Interfaces:

<Foo>Ops.mesa, <Foo>Defs.mesa
i.e. EditOps.mesa, MTOps.mesa, MTDefs.mesa

Implementation Modules (most frequently used):

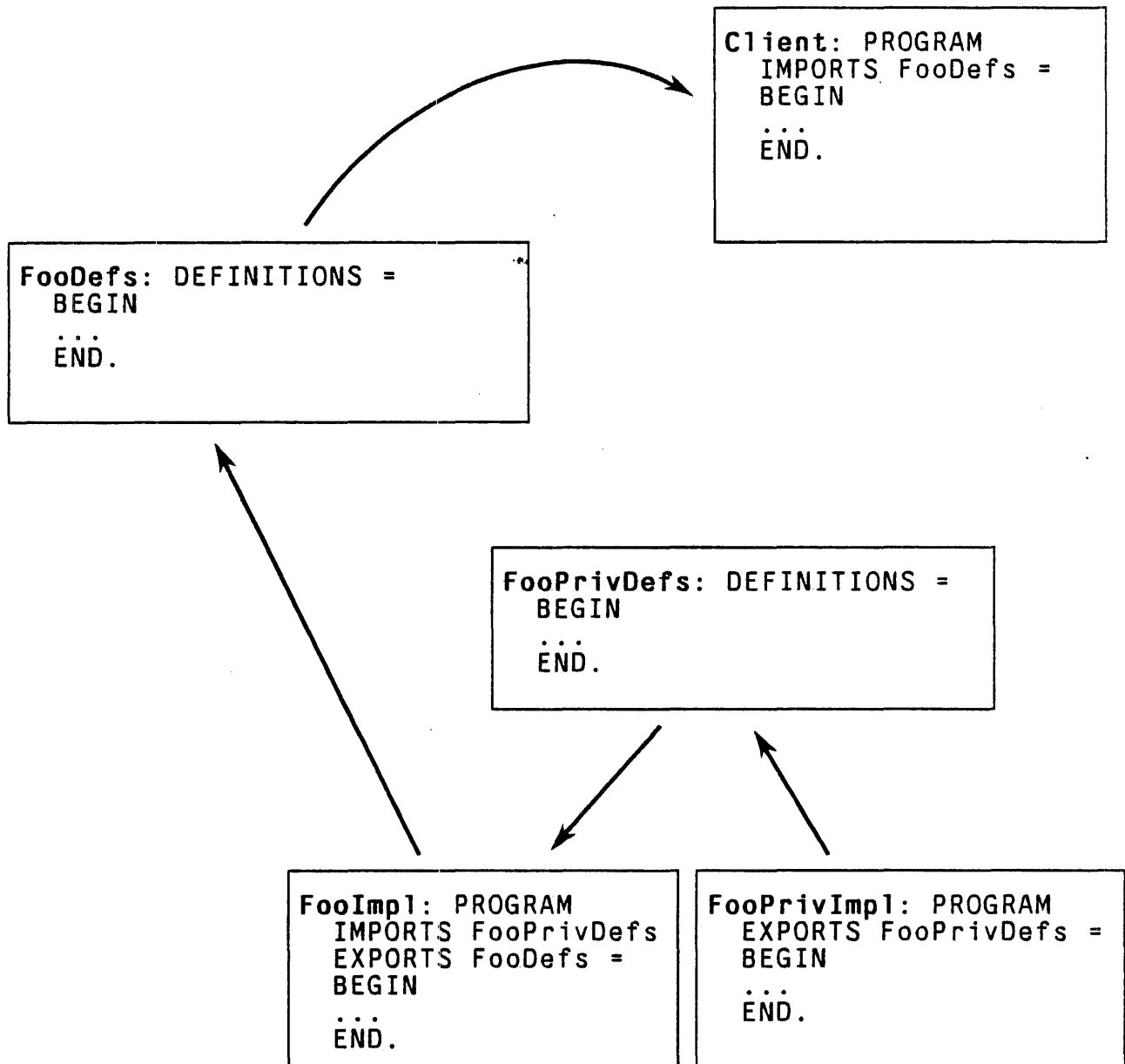
<Foo>Impl.mesa
-- or <Foo><function>Impl.mesa -- recommended method
-- or <Foo>AImpl.mesa, <Foo>BImpl.mesa -- old method
i.e. EditImpl.mesa,
MTAImpl.mesa, MTBImpl.mesa
MTFormImpl.mesa, MTCalcImpl.mesa

Client Modules (rarely used):

<Foo>Control.mesa or <Foo>Client.mesa
i.e. EditControl.mesa or MTClient.mesa

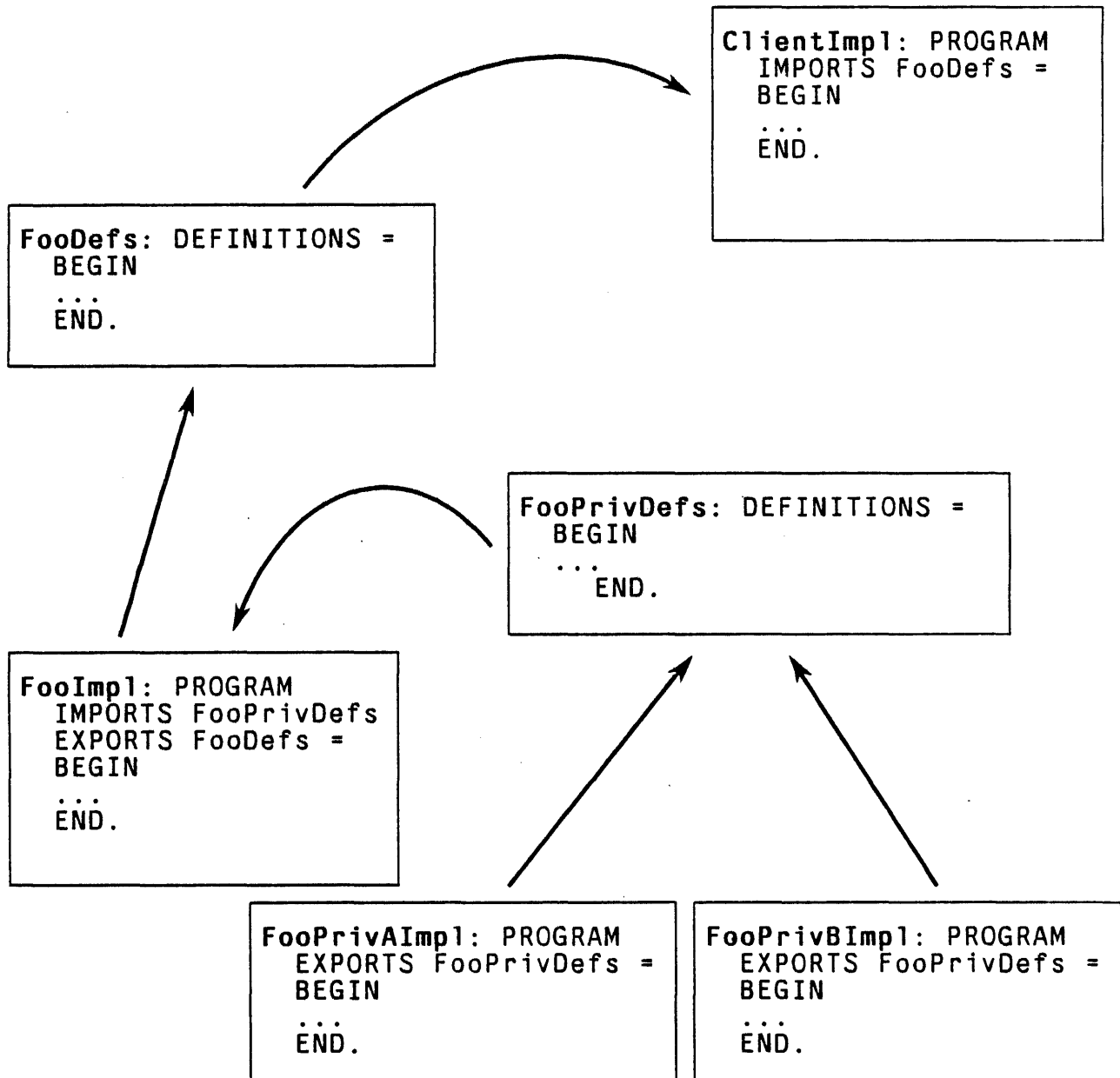
More Examples

This example shows how you might restrict a client to only see what is necessary in one interface. A separate interface that the client never sees is used by the implementation module(s).



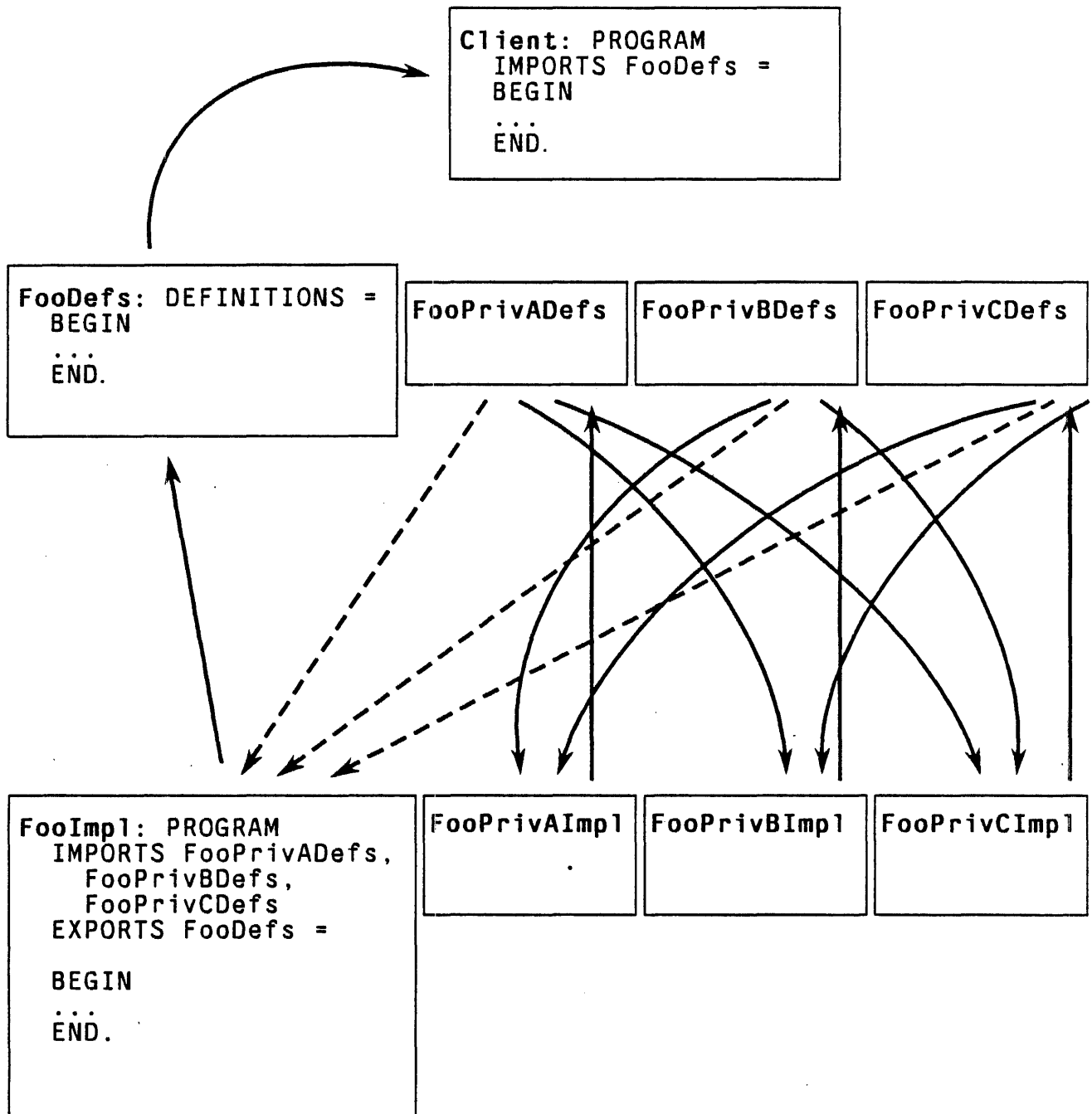
More Examples

In this example, several program modules implement the private interface:



More Examples

In this example, several program modules implement several private interfaces:



Configurations of Configurations

In the earlier example, all of the implementation files can be combined into one configuration file:

```
--FooDefsImpl.config
FooDefsImpl: CONFIGURATION
  IMPORTS Heap, String -- or whatever --
  EXPORTS FooDefs = {
    FooImpl;
    FooPrivAImpl;
    FooPrivBImpl;
    ...
  }.
```

Notice there is no CONTROL clause. The resulting file *FooDefsImpl.bcd* can then be included in the client's own configuration:

```
--Client.config
Client: CONFIGURATION
  IMPORTS Heap, String -- must include ClientImpl & FooDefsImpl imports --
  CONTROL ClientImpl = {
    ClientImpl;
    FooDefsImpl;
    ...
  }.
```

INDEX

C

Client 2-5, 2-20, 2-31 - 2-32, 2-34 - 2-35
 Compile-Time 2-7, 2-12
 Configuration 2-22 - 2-23, 2-25 - 2-27, 2-29 - 2-31, 2-35

D

DEFINITIONS 2-4 - 2-8, 2-10 - 2-11, 2-13, 2-17, 2-19 - 2-21, 2-23 - 2-24, 2-32 - 2-34
 DIRECTORY 2-8 - 2-15, 2-17 - 2-18, 2-20 - 2-21, 2-24, 2-26

E

EXPORTS 2-18, 2-20 - 2-21, 2-24, 2-26 - 2-27, 2-29, 2-32 - 2-35

I

IMPORTS 2-13 - 2-15, 2-17, 2-20, 2-23 - 2-27, 2-29, 2-32 - 2-35
 Interface 2-5, 2-10, 2-14 - 2-16, 2-18 - 2-19, 2-29

N

Naming 2-30 - 2-31

O

Opaque 2-19
 Order 2-15

P

PROGRAM 2-3 - 2-5, 2-11, 2-13, 2-17 - 2-21

R

Run-Time 2-7, 2-14

S

Summary 2-12, 2-14

U

USING 2-9 - 2-15, 2-17, 2-20, 2-24, 2-26

Dynamic Storage Allocation

Outline

1. How is DSA in Mesa different?
2. Heaps
 - a. What are they? What kinds are there?
 - b. How do you declare / get rid of them?
 - c. How do you allocate / deallocate from them?
 - d. What does a typical example of heap usage look like?
 - e. How do you know which heap to use?
3. Special data structures that use the Heap facility
 - a. Strings
 - b. Sequences

Overview of Dynamic Storage Allocation (DSA)

When do you need DSA? When the amount of storage needed is not known until run-time.

This lecture is intended to help you understand the different issues involved in dynamically allocating storage as it relates to the Mesa Language and to learn how to allocate storage *efficiently*.

We will not be talking about low level issues such as operating system issues.

Why Dynamically Allocate?

- Space, logical considerations

Dynamic allocation is needed for objects which can not or should not be allocated in your program's global and local frames.

Remember, storage for local frames comes from the Main Data Space.

How is DSA in Mesa Different?

What's the big deal about DSA as it pertains to Mesa? How is it different from forms of DSA you've seen in other systems?

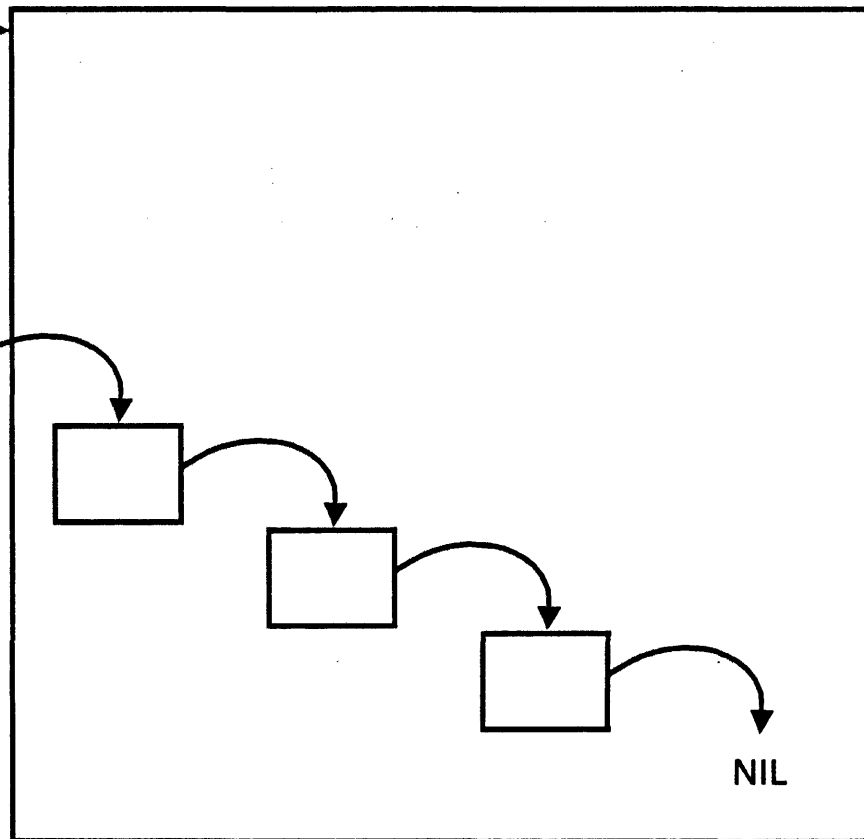
- Choice of storage type
- Type checking
- No automatic garbage collection at either the micro or macro level

Macro Level:

myHeap

Micro Level:

linkedListPtr



Heaps

- Heap facility = **Heap Interface + NEW and FREE**
- Allocate fixed or arbitrary sized blocks of storage (nodes)
- Three types of heaps:

Normal: allows allocation of arbitrary-sized nodes

Uniform: allows allocation of nodes less than or equal to a specified size

MDS: allows allocation of arbitrary-sized nodes from the Main Data Space (MDS)

- Public and Private Heaps

Public: The environment provides a normal heap (the *systemZone*) and an MDS heap (the *systemMDSZone*). Information in these heaps can be shared among subsystems. There is no public uniform heap.

Private: User can create private heaps (normal, uniform, or MDS) for specific purposes.

Choosing the Proper Heap Type for a Specific Application

- If a set of programs requires a lot of private storage, it is often more efficient to create a *private* heap than to use the *system-provided (public)* heaps.
- If objects being allocated are all the same size, *uniform* heaps are more efficient since less overhead is required for each node.
- A process that uses a *private* heap instead of a *system (public)* heap avoids direct competition with other processes for resources.
- *System* heaps can be used with low overhead for transient storage.
- *MDS* heaps would generally be used by low-level system clients.

Using the Public Heaps

In the Heap Interface, there are two constants that are available for use in your program.

```
Heap.systemZone: READONLY UNCOUNTED_ZONE;  -- the Normal system heap
Heap.systemMDSZone: READONLY MDSZone;      -- the MDS system heap
```

You can either reference these constants directly throughout your program, or create a local variable and assign the value of the system heap to it.

Example:

```
z: UNCOUNTED_ZONE = Heap.systemZone;
mz: MDSZone = Heap.systemMDSZone;
```

Using Private Heaps

To create a private heap, you must call one of the Create procedures provided in the Heap interface. Abbreviated declarations are shown below with only the required parameters (others are defaulted). For a complete list of parameters, consult the Pilot Programmer's Manual.

```
Heap.Create: PROCEDURE [initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4, ... ]
  RETURNS [UNCOUNTED ZONE];
```

```
Heap.CreateUniform: PROCEDURE [initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4,
  objectSize: Heap.NWords, ... ]
  RETURNS [UNCOUNTED ZONE];
```

```
Heap.CreateMDS: PROCEDURE [initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4, ... ]
  RETURNS [MDSZone];
```

Examples:

```
z: UNCOUNTED ZONE ← Heap.Create[initial: 4];
uz: UNCOUNTED ZONE ← Heap.CreateUniform[
  initial: 5, objectSize: 10];
mz: MDSZone ← Heap.CreateMDS[initial: 2];
```

Hint: Since there are so many parameters to these procedures, it is recommended that you name all of the parameters that you are passing.

How to Get Rid of Private Heaps

When you no longer need a private heap, you should delete it so that the space can be used by other processes. To delete private heaps, use one of the delete procedures provided in the Heap interface. Deleting a heap will automatically deallocate any currently allocated storage (which is what you usually want), unless the `checkEmpty` parameter is set to `TRUE`.

```
Heap.Delete: PROC[      -- for Normal or Uniform Heaps
  z: UNCOUNTED_ZONE, checkEmpty: BOOLEAN ← FALSE];
```

```
Heap.DeleteMDS: PROC[   -- for MDS heaps only
  z: MDSZone, checkEmpty: BOOLEAN ← FALSE];
```

Note: When you have deleted a heap, you should set your zone pointer to `NIL` to avoid accessing a previously deleted heap.

Examples:

```
z: UNCOUNTED_ZONE ← Heap.Create[4];
mz: MDSZone ← Heap.CreateMDS[2];
...
Heap.Delete[z];
z ← NIL;

Heap.DeleteMDS[mz];
mz ← NIL;
```

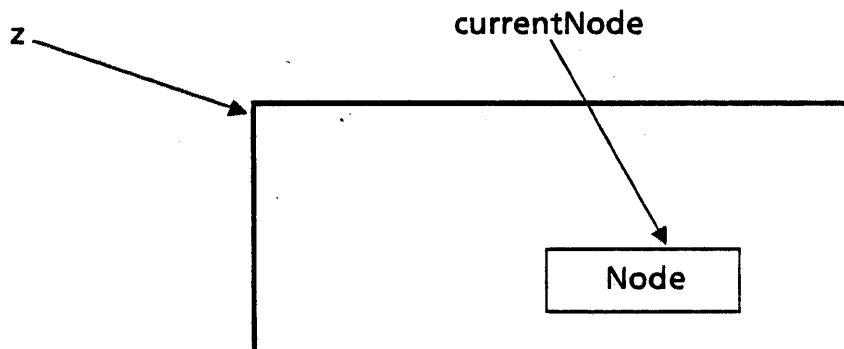
Allocating Storage From a Heap

Nodes are allocated from a heap using the NEW operator. NEW returns a LONG POINTER to the object in the heap.

Example:

```
z: UNCOUNTED_ZONE ← Heap.Create[initial: 5];  
Node: TYPE = RECORD[...];  
NodePtr: TYPE = LONG POINTER TO Node;  
  
currentNode: NodePtr ← z.NEW[Node];
```

Graphically speaking:



Other Examples:

```
sz: UNCOUNTED_ZONE ← Heap.systemZone;  
anotherNode: NodePtr ← sz.NEW[Node];  
yetAnotherNode: NodePtr ← Heap.systemZone.NEW[Node];
```


Getting Initial Values

There are several ways for a new node to pick up its initial values. They are called *type* initialization, *field* initialization, and *explicit* initialization. An example is the best way to understand the differences.

Example:

```

z: UNCOUNTED_ZONE ← Heap.Create[initial: 4];
NodePtr: TYPE = LONG_POINTER_TO Node;
Node: TYPE = RECORD [
  value: LONG_CARDINAL ← 0,  -- this is a default for the field
  next: NodePtr ← nilInit] ← [1, NIL];  -- this is a default for the type

currentNode, otherNode: NodePtr;
nilInit: NodePtr ← NIL;

-- to explicitly set the values, use a record constructor
currentNode ← z.NEW[Node ← [value: 4, next: otherNode]];

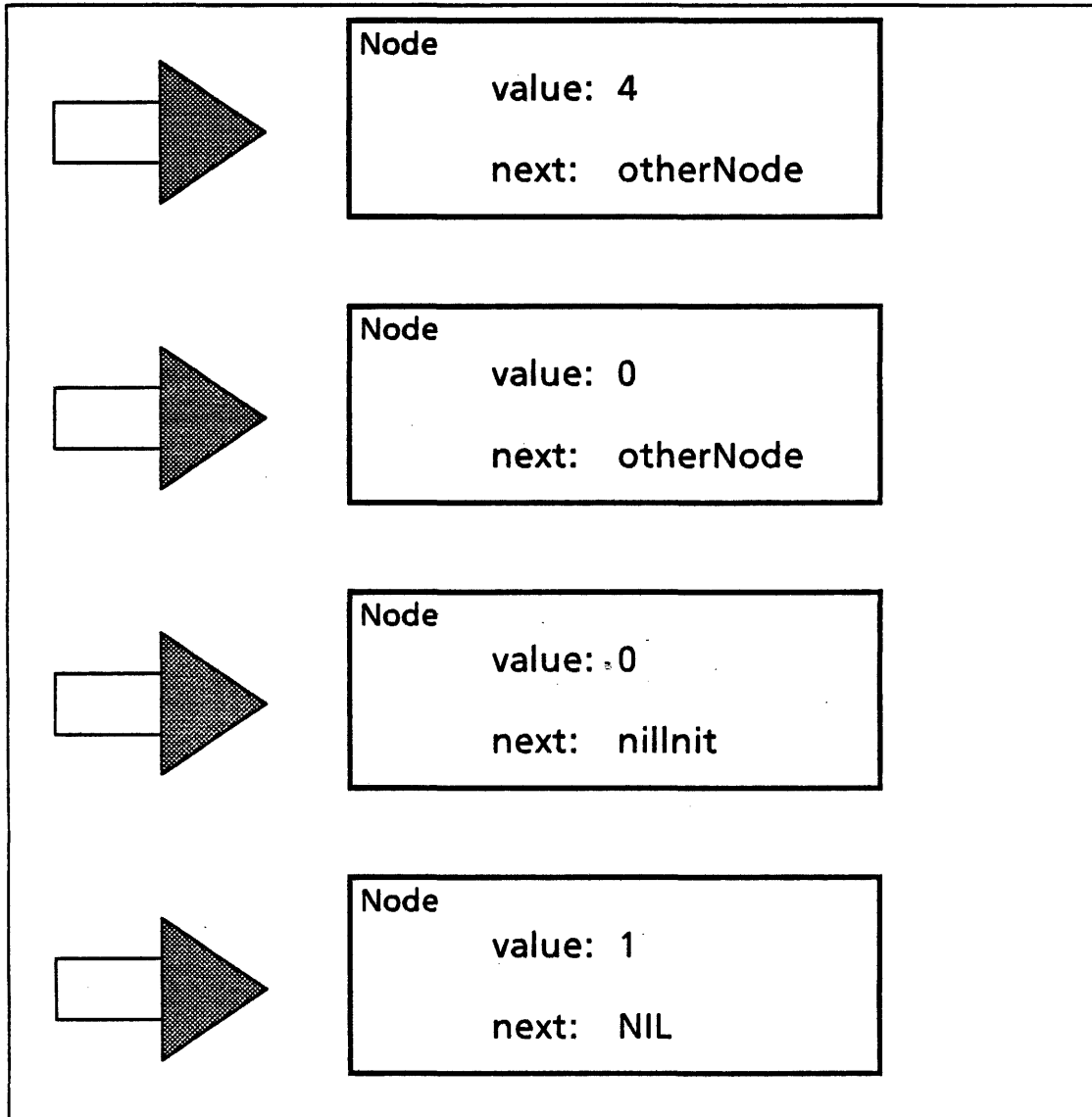
-- if you omit a field in the record constructor, the field will pick up its
-- the default for the field.
currentNode ← z.NEW[Node ← [ , next: otherNode]];

-- to pick up all of the field defaults, elide all fields
currentNode ← z.NEW[Node ← [ . ]];

-- to pick up the default for the type, do not specify a record constructor
-- if no type default exists, it will pick up the field defaults
currentNode ← z.NEW[Node];

```

Getting Initial Values (Cont'd)



Deallocating Storage From a Heap

Nodes are deallocated from a Heap using the Mesa FREE operator. FREE sets the node pointer to NIL and then frees the storage used for the node. For FREE to make changes to the pointer, it must have a *reference* to the node pointer (since all parameter passing is call-by-value).

Example:

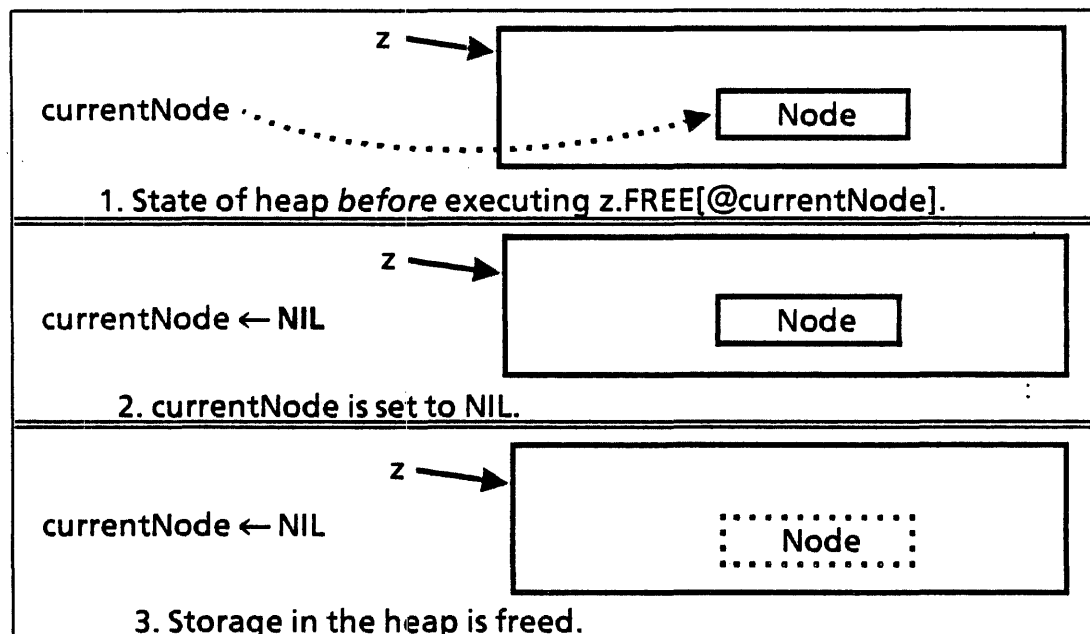
```

z: UNCOUNTED_ZONE ← Heap.Create[initial: 10];
NodePtr: TYPE = LONG_POINTER_TO Node;
Node: TYPE = RECORD [
  value: LONG_CARDINAL ← 0,
  next: NodePtr ← nilInit] ← [1, NIL];

currentNode: NodePtr ← z.NEW[Node ← [value: 4, next: NIL];
...
z.FREE[@currentNode];

```

Graphically speaking:



Typical Use of a Heap in a Program

DIRECTORY

```
Heap USING [Create, Delete],
...;
```

```
SampleProgram: PROGRAM IMPORTS Heap, ... = {
```

```
z: UNCOUNTED_ZONE ← NIL;
```

```
RecStructure: TYPE = RECORD[...];
```

```
PtrType: TYPE = LONG POINTER TO RecStructure;
```

```
ptr: PtrType ← NIL;
```

```
Allocate: PROCEDURE[...] = {
  ptr ← z.NEW[RecStructure ← []];
  ...};
```

```
Deallocate: PROCEDURE[...] = {
  z.FREE[@ptr];
  ...};
```

```
CleanUp: PROCEDURE = {
  If z # NIL THEN {
    Heap.Delete[z];
    z ← NIL };
  ...};
```

```
Init: PROCEDURE = {
  IF z = NIL THEN z ← Heap.Create[initial: 5]
  ...};
```

```
};
```

Expanding a Heap

When there is not enough contiguous space in the heap to allocate a node, the system *automatically* expands the heap by a specified number of pages. This number is specified by a parameter to the Create procedure, called *increment*. (It is defaulted to 4.) Expansions are not guaranteed to be contiguous in VM.

Let's trace a representation of an automatic expansion by the system when more space was needed in the heap. In this case: `initial = 3`, `increment = 2`



1. Heap is created in VM;



2. Two different 1-page objects are allocated;



3. First object is freed; program needs 2 pages for next object;



4. Heap is expanded;



5. New object is allocated from the extension;



6. A new 1-page object is allocated from previously used page that was freed.

What would happen if `objectSize > increment` ?

Expanding a Heap (Cont'd)

Fragmentation and other problems may be avoided by doing a manual expansion, if the heap needs to be expanded by more than *increment* number of pages. This expansion can be done by calling the Expand procedure in the Heap interface.

```
Heap.Expand: PROC[  
  z: UNCOUNTED_ZONE, pages: Environment.PageCount];
```

Other Operations on Heaps

- To determine initial parameters and current attributes of a heap:
Heap.GetAttributes: PROC[z: UNCOUNTED ZONE]
RETURNS[heapPages, maxSize,
increment: Environment.PageCount, ...];
- To return heap to its virgin state:
Heap.Flush: PROC[z: UNCOUNTED ZONE];
- To release unused extensions of a heap:
Heap.Prune: PROC[z: UNCOUNTED ZONE];
- To allocate an untyped node:
Heap.MakeNode: PROC[
z: UNCOUNTED ZONE ← Heap.systemZone, n: Heap.NWords]
RETURNS[p: LONG POINTER];
- To deallocate an untyped node:
Heap.FreeNode: PROC[
z: UNCOUNTED ZONE ← Heap.systemZone, p: LONG POINTER];

Note: There are similar procedures for MDS heaps.

Strings

Definition from Day 1:

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

Where:

text is a PACKED ARRAY of characters
maxLength is the *maximum* numbers of characters that text can hold
length is the number of characters *currently* in text

Allocating and Deallocating Strings

- 1) Allocate fixed-sized storage from a local or global frame of the program:

```
string: LONG STRING ← [256];
-- sets maxlength to 256, length to zero & text is uninitialized
```

- 2) Assign a string literal to a string variable:

```
gString: LONG STRING ← "Hello";
lString: LONG STRING ← "Hello"L;
-- sets maxlength to 5, length to 5 & text to the characters 'H, 'e, 'l, 'l, 'o
```

- 3) Use the NEW operator to allocate storage from a heap:

```
str: LONG STRING;
str ← someZone.NEW[StringBody[8]]; -- MUST specify size of StringBody
-- sets maxlength to 8, length to zero & text is uninitialized
```

Deallocate using Mesa construct FREE:

```
someZone.FREE[@str];
```

- 4) Use procedures provided by the String interface to allocate storage from a heap:

```
str: LONG STRING;
str ← String.MakeString[z: someZone, maxlength: 8];
-- sets maxlength to 8, length to zero & text is uninitialized
```

Deallocate using String interface procedures:

```
String.FreeString[z: someZone, s: str];
```

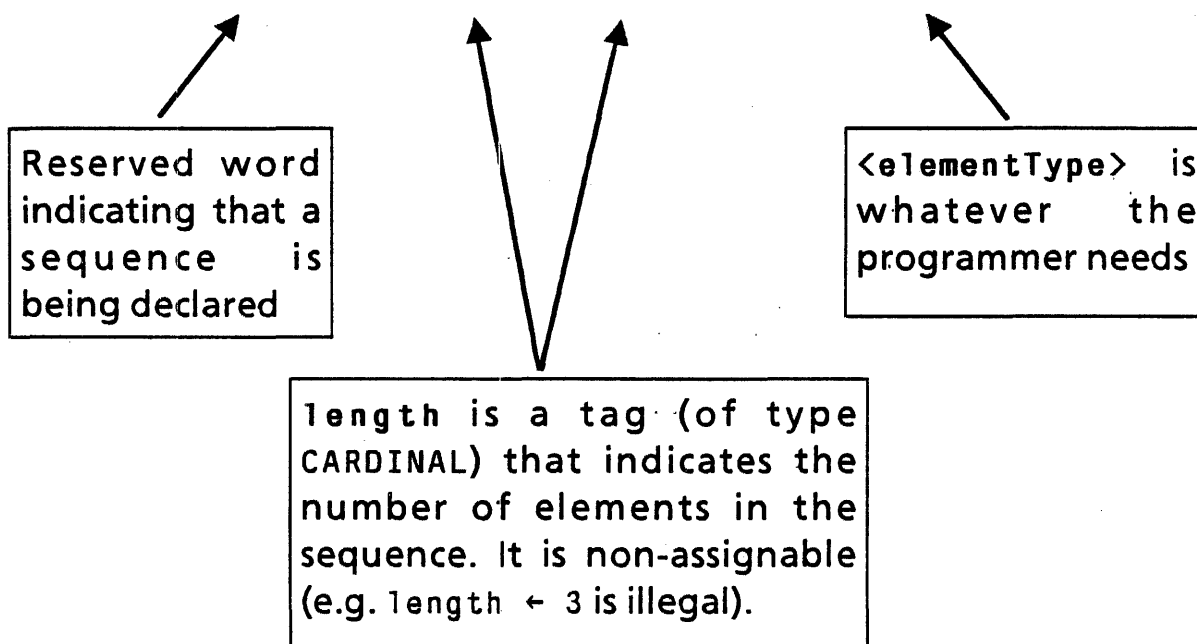
Sequences

Sequences are dynamic arrays

- They must be declared within a record
- There cannot be more than one sequence per record
- They must be the last field in a record

Syntax:

```
ptr: LONG POINTER TO Rec;  
Rec: TYPE = RECORD [  
  <zero or more fields>,  
  seq: SEQUENCE length: CARDINAL OF <elementType>];
```



Sequence Example

```

myRecordPtr: LONG POINTER TO MyRecord;
MyRecord: TYPE = RECORD [
  bool1: BOOL,  -- These booleans are just for this example application
  bool2: BOOL,  -- and are not specific to Sequences in general
  seq: SEQUENCE length: CARDINAL OF INTEGER];

```

To allocate this record to contain a sequence of ten elements and set the boolean variables:

```

myRecordPtr ← someZone.NEW[
  MyRec[10] ← [TRUE, FALSE, ]];  -- MUST specify size of sequence

```

Although other fields can be assigned during allocation, the sequence must be assigned in a separate statement, usually a loop.

```

FOR i: CARDINAL in [0..myRecordPtr.length) DO
  myRecordPtr.seq[i] ← 0;
ENDLOOP;

```

To deallocate this record:

```

someZone.FREE[@myRecordPtr];

```

To use the sequence:

```

num: INTEGER;
num ← myRecordPtr.seq[3];
num ← myRecordPtr[3];  -- equivalent to above statement, and
                       -- num ← myRecPtr ↑ .seq[3]; , and
                       -- num ← myRecPtr ↑ [3];
ok: BOOLEAN ← myRecordPtr.bool1;
IF myRecordPtr.length > 5 THEN RETURN[TRUE];

```

Example Using Sequences, Strings and Heaps

DIRECTORY

Heap USING [Create, Delete];

GetInfo: PROGRAM IMPORTS Heap = {

Dossier: TYPE = RECORD[

 name: LONG STRING, ← NIL,

 age: CARDINAL,

 ssn: LONG STRING ← NIL];

DossierPtr: TYPE = LONG POINTER TO Dossier;

Personnel: TYPE = RECORD[

 currentEntry: CARDINAL ← 0,

 stuff: SEQUENCE maxEntries: CARDINAL OF DossierPtr];

PersonnelPtr: TYPE = LONG POINTER TO Personnel;

people: PersonnelPtr ← NIL;

z: UNCOUNTED ZONE ← Heap.Create[initial: 5];

--Procedures

Init: PROC = {

 people ← z.NEW[Personnel[6]]; -- allocate the sequence of pointers

 FOR i: CARDINAL IN [0..people.maxEntries) DO

 people[i] ← z.NEW[Dossier]; -- allocate each of the records in the sequence

 people[i].age ← 0; -- initialize any fields that don't have defaults

 ENDLOOP;

}; -- of Init

Example Using Sequences, Strings and Heaps (cont'd)

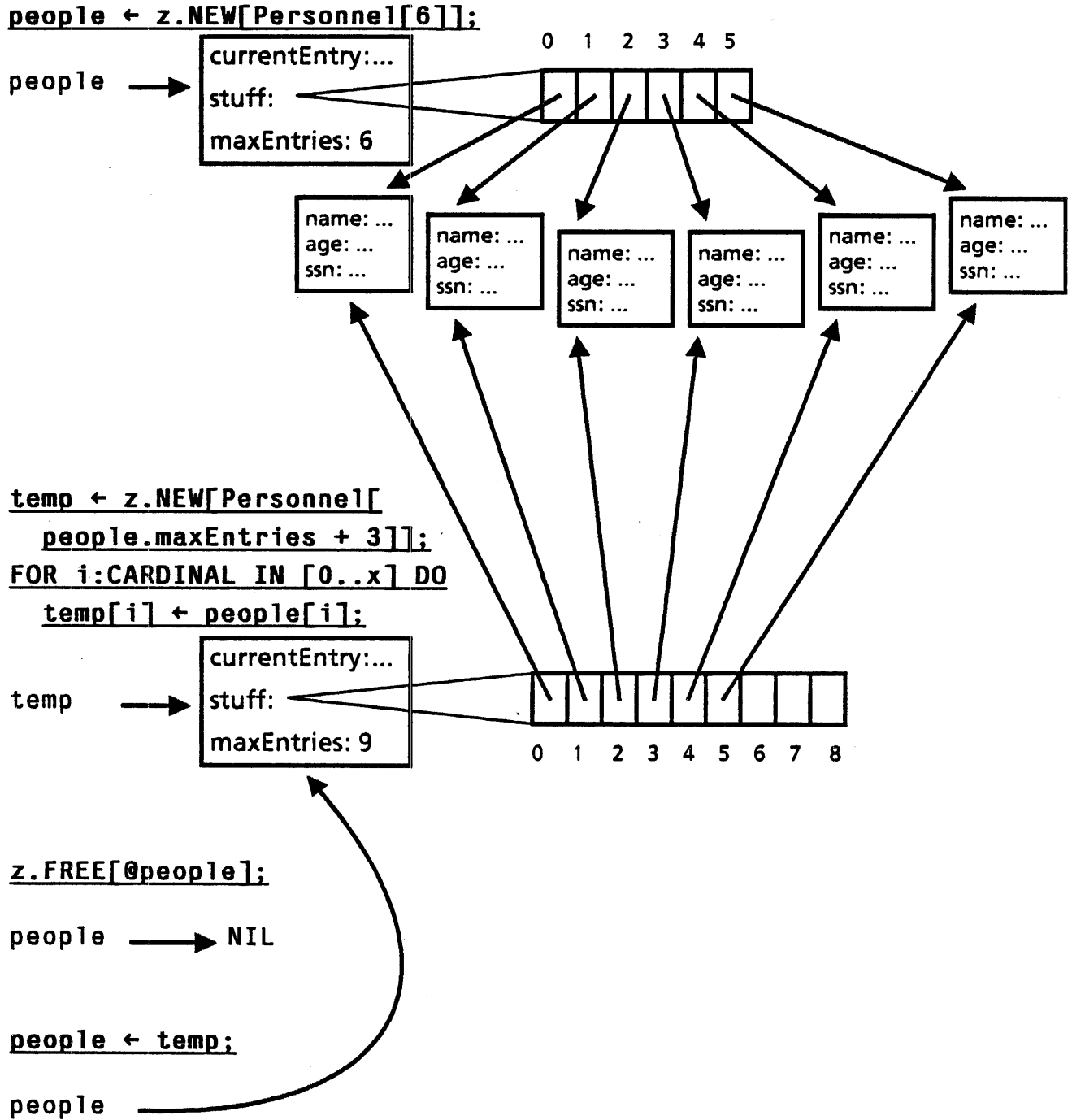
```

ExpandList: PROC = { -- expands the sequence of pointers to include 3 new entries
  -- create a new, larger (temporary) sequence
  temp: PersonnelPtr ← z.NEW[Personnel[people.maxEntries + 3]];
  FOR i:CARDINAL IN [0..people.maxEntries) DO
    -- copy each of the pointers to the Dossier records into the new sequence
    temp[i] ← people[i];
  ENDLOOP;
  temp.currentEntry ← people.currentEntry; -- copy over any other fields
  FOR j:CARDINAL IN [people.maxEntries..temp.maxEntries) DO
    temp[j] ← z.NEW[Dossier]; -- allocate the three new records in the sequence
    temp[j].age ← 0;
  ENDLOOP;
  z.FREE[@people]; -- deallocate the old sequence of pointers
  people ← temp; -- make the global variable point to the new sequence; Why??
}; -- of ExpandList

ShrinkList: PROC [toBeDeleted: CARDINAL] = {
  -- ShrinkList removes the record indexed by 'toBeDeleted' from the sequence of pointers
  IF toBeDeleted >= people.maxEntries THEN RETURN -- or handle elegantly
  ELSE { -- move each of the pointers of the sequence over one position
    z.FREE[@people[toBeDeleted]]; -- deallocate the record indexed by 'toBeDeleted'
    FOR i:CARDINAL IN [toBeDeleted..people.maxEntries - 1) DO
      people[i] ← people[i + 1]; -- copy each pointer over ignoring 'toBeDeleted'
    ENDLOOP;
    people[people.maxEntries] ← NIL; -- reinitialize the last record in the sequence
  }; -- End If
}; -- of ShrinkList

```

Example Using Sequences, Strings and Heaps (cont'd)



Sequences vs. Array Descriptors

- Array descriptors are useful for parameter passing.
- Sequences are type-safe.
- Sequences can be more readily treated as regular arrays.
- Sequences must be allocated from a heap; arrays can be allocated from a local or global frame.
- Sequences are the replacement for array descriptors. Newer code uses mostly sequences; older code uses array descriptors.

Review of DSA

- Heaps are of the type you choose: normal, uniform, MDS, public, private.
- To use a public heap, use a variable from the Heap interface (`Heap.systemZone`.) To use a private heap, call `Heap.Create` (and later `Heap.Delete`.)
- To allocate and deallocate from heaps, use the language operators `NEW` and `FREE`.
- Allocation is strongly typed; you almost never allocate untyped storage.
- Allocate storage from a heap when it's not logical to allocate storage from a local or global frame.
- Among the things allocated from a heap are strings and sequences, which require special attention.
- Deallocation is your responsibility; it's not performed automatically. When you create a heap, you must delete it. When you allocate storage for a data structure, you must deallocate that storage.

Selected Bibliography

Information on Heaps:

Pilot Programmers Manual (version 12.0). Ch. 4, pp. 49 - 54.

Information on Strings:

Mesa Language Manual (version 12.0). Ch. 6, pp. 2 - 5

Pilot Programmers Manual (version 12.0). Ch. 7, pp. 5 - 10

Information on Sequences:

Mesa Language Manual (version 12.0). Ch. 6, pp. 25 - 30

Information on Array Descriptors:

Mesa Language Manual (version 12.0). Ch. 6, pp. 5 - 9

Information on Virtual Memory:

Pilot Programmers Manual (version 12.0). Ch. 4, pp. 29 - 43

FormSWLayoutTool Code

Outline

1. Tool Window code generated by FormSWLayoutTool (Handout)
 - a. Global variables
 - b. Main Program
 - c. Call-back Procedures
 - d. Command Procedures
2. Adding code to your tool

FormSWLayoutTool Code

The tool FormSWLayoutTool generates most of the tool code for you. You only have to supply the command procedures that are specific to your tool. See the Tool-Written Factorial Tool handout.

Global Variables

--Types

```
DataHandle: TYPE = LONG POINTER TO Data;
Data: TYPE = MACHINE DEPENDENT RECORD [
  msgSW(0): Window.Handle ← NIL,
  formSW(2): Window.Handle ← NIL,
  fileSW(4): Window.Handle ← NIL,
  number(6): CARDINAL ← 0,
  format(7): FormatType ← decimal];
```

```
FormatType: TYPE = {hex, octal, decimal};
```

--Variables

```
data: DataHandle ← NIL;
wh: Window.Handle ← NIL; -- the window handle for the entire window
zone: UNCOUNTED_ZONE ← Heap.Create[initial: 4];
busyBit: BOOLEAN ← FALSE; -- is TRUE when tool is busy
```

Where:

`data↑.msgSW`, `data↑.formSW`,
`data↑.fileSW` are window handles for the three subwindows

`data↑.number` holds the current value of **Number** in the tool window

`data↑.format` holds the current value of **Format** in the tool window

Main Program

```
BEGIN
```

```
  -- Mainline code
```

```
  Init[];
```

```
END...
```

When the program is started, one call is made to the procedure `Init`.

Call-Back Procedures

```

Init: PROCEDURE = {
  --This procedure is called once from the main program.
  --It creates the tool and registers a command with the Executive.
  Exec.AddCommand[name: "FactorialTool."~"L, proc: NoOp,
    help: NIL, unload: Unload];
  wh ← Tool.Create[makeSWsProc: MakeSWs, initialState: default,
    clientTransition: ClientTransition, name: "FactorialTool"L,
    cmSection: "FactorialTool"L];
};

NoOp: Exec.ExecProc = { };

Unload: Exec.ExecProc = {
  IF Busy[] THEN {
    Exec.OutputProc[h]["Tool is busy. Sorry.\n"L];
    RETURN[error] };
  Tool.Destroy[wh];
  Exec.RemoveCommand[h, "FactorialTool."~"L];
  Done[] };

ClientTransition: ToolWindow.TransitionProcType = {
  --This procedure is called whenever the tool undergoes a user-invoked
  --transition between the 3 states: active, tiny, and inactive.
  --input Parameters: window, old, new
  SELECT TRUE FROM
    old = inactive =>
      IF data = NIL THEN data ← zone.NEW[Data ← []];
    new = inactive =>
      IF data # NIL THEN {
        zone.FREE[@data]};
  ENDCASE;
};

```

Call-Back Procedures (cont'd)

MakeSWs: Tool.MakeSWsProc = {

*--This procedure is called when the tool is created and whenever the tool's state
--changes from inactive to active, or tiny. This procedure creates three subwindows.*

--Input Parameters: window

```
logName: LONG STRING ← [20];
Tool.UnusedLogName[unused: logName, root: "FactorialTool.log"L];
data.msgSW ← Tool.MakeMsgSW[window: window];
data.formSW ← Tool.MakeFormSW[
    window: window, formProc: MakeForm];
data.fileSW ← Tool.MakeFileSW[window: window, name: logName];
};
```

MakeForm: FormSW.ClientItemsProcType = {

--This procedure is called when the form subwindow is created.

--Input Parameters: sw

--Output Results: items, freeDesc

```
format: ARRAY [0..3) OF FormSW.Enumerated ← [
    ["hex"L, 0], ["octal"L, 1], ["decimal"L, 2]];
items ← FormSW.AllocateItemDescriptor[3];
items[0] ← FormSW.NumberItem[
    tag: "Number"L, place: [6, FormSW.line0], signed: FALSE,
    notNegative: TRUE, value: @data.number];
items[1] ← FormSW.EnumeratedItem[
    tag: "Format"L, place: [168, FormSW.line0], feedback: all,
    choices: DESCRIPTOR[format], value: @data.format];
items[2] ← FormSW.CommandItem[
    tag: "Factorial"L, place: [6, FormSW.line1],
    proc: Factorial];
RETURN[items: items, freeDesc: TRUE];
};
```


Call-Back Procedures (cont'd)

The `Init` procedure calls `Exec.AddCommand` passing in 2 procedures: `NoOp` and `UnLoad`. These procedures are then called as needed by XDE. `Tool.Create` is also passed 2 procedures: `MakeSWs` and `ClientTransition`.

The procedure `MakeSWs` calls procedure `Tool.MakeFormSW` passing in the procedure `MakeForm` to make the form subwindow. `MakeForm` is then called as needed by XDE.

These procedures that are passed to System procedures, and then called as needed by XDE are "Call-back procedures".

Scheduling Procedures

```
Busy: ENTRY PROCEDURE RETURNS [isBusy: BOOLEAN] = {  
  ENABLE UNWIND => NULL;  
  isBusy ← busyBit;  
  busyBit ← TRUE;  
};
```

```
Done: ENTRY PROCEDURE = {  
  ENABLE UNWIND => NULL;  
  busyBit ← FALSE;  
};
```

Printing Procedures

```
Write: Format.StringProc = {Put.Text[data.fileSW, s]};
```

```
Msg: Format.StringProc = {Put.Text[data.msgSW, s]};
```

Where:

```
Format.StringProc = PROCEDURE[  
  s: LONG STRING, clientData: LONG POINTER ← NIL];
```

The Put Interface

While we're on the topic of displaying information to the screen, here is how to do it from any program. The Put Interface, defined in the Mesa Programmer's Manual, has many procedures to aid you in displaying data to windows. Some of the most commonly used procedures are shown here, but there are many others. Refer to the MPM.

All of these procedures take a `Window.Handle` and (usually) a piece of data to be formatted. If the `Window.Handle` is `NIL` then the output is directed to the Herald Window.

```
Put.Char: PROCEDURE[h: Window.Handle ← NIL, char: CHARACTER];
```

```
Put.Text: PROCEDURE[h: Window.Handle ← NIL, s: LONG STRING];
```

```
Put.Line: PROCEDURE[h: Window.Handle ← NIL, s: LONG STRING];  
-- puts a LONG STRING and then a CR
```

```
Put.CR: PROCEDURE[h: Window.Handle ← NIL];
```

Dummy Command Procedures

```
Factorial: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L]; RETURN};
  Process.Detach[FORK FactorialInternal[]]};
```

```
FactorialInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  Write["Factorial called\n"L];
  Done[];
};
```

The resulting file from the FormSWLayoutTool can be compiled and executed. Command procedures like the above are used since the tool cannot know what the command is supposed to do.

Whenever **Factorial!** (in the tool window) is invoked, the procedure **Factorial** is called. **Factorial** checks to see if the tool is busy.

FactorialInternal is where the real work to calculate the factorial would be done.

Adding Code

The dummy command procedures can be modified to contain either the actual implementation (shown below), or, more commonly, a call to an interface that implements the command (shown on next page).

```

FactorialInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF data.number > 12 THEN {      -- Out of range
    Put.CR[data.msgSW];
    Put.Text[data.msgSW, "Too high, try again."L]
  }
  ELSE {
    result: LONG CARDINAL ← 1;
    temp: CARDINAL ← data.number;
    WHILE temp > 0 DO
      result ← result * temp;
      temp ← temp - 1;
    ENDLOOP;
    Put.CR[data.fileSW];
    Put.Text[data.fileSW, "The factorial of "];
    Put.Decimal[data.fileSW, data.number];
    Put.Text[data.fileSW, " is "];
    SELECT data.format FROM
      hex      => Put.LongNumber[data.fileSW, result, [16]];
      octal    => Put.LongNumber[data.fileSW, result, [8]];
      decimal => Put.LongNumber[data.fileSW, result, [10]];
    ENDCASE;
    Put.CR[data.fileSW];
  };
  Done[];
};

```

Adding Code (con't)

If the implementation of the command is something more complex than a simple procedure, it is wise to put the implementation in another module and simply make a procedure call to it.

```
FactorialInternal: PROCEDURE = {  
  ENABLE ABORTED => {Done[]; CONTINUE};  
  MyDefs.Factorial[data.msgSW, data.fileSW, data.number];  
  Done[];  
};
```

Using the FormSWLayoutTool

Layout Mode / Edit Mode

FormSWLayoutTool

FormType: {**bool**, command, enum, longNum, numb, source, string, tag}
Tag: Verbose **Zone:**
AlignX Usebox Anyfont Root: SimpleTool
DoIt! Clear! SetDefaults! Load! Save! Plagiarize!

Verbose **This is a Phone Book Tool**
Find! Insert! Delete!
Building: {fix the enums}
Name: **Phone=**
Address:

AlignX

FormSWLayoutTool	
<p>FormType: [000], command, enum, longNum, numb, source, string, tag}</p> <p>Tag: Verbose Zone:</p> <p>AlignX Usebox Anyfont Root: SimpleTool</p> <p>Do! Clear! SetDefaults! Load! Save! Plagiarize!</p>	
<p>Verbose This is a Phone Book Tool</p> <p>Find! Insert! Delete!</p> <p>Building: {fix the enums}</p> <p>Name: Phone=</p> <p>Address:</p>	

FormSWLayoutTool	
<p>FormType: [000], command, enum, longNum, numb, source, string, tag}</p> <p>Tag: Zone:</p> <p>AlignX Usebox Anyfont Root: SimpleTool</p> <p>Do! Clear! SetDefaults! Load! Save! Plagiarize!</p>	
<p>Verbose This is a Phone Book Tool</p> <p>Find! Insert! Delete!</p> <p>Building: {fix the enums}</p> <p>Name: Phone=</p> <p>Address:</p>	

UseBOX

FormSWLayoutTool

FormType: {bool}, command, enum, longNum, numb, source, string, tag)
 Tag: Zone:
 AlignX **Usebox** Anyfont Root: UseBoxTool
 Dolt! Clear! SetDefaults! Load! Save! Plagiarize!

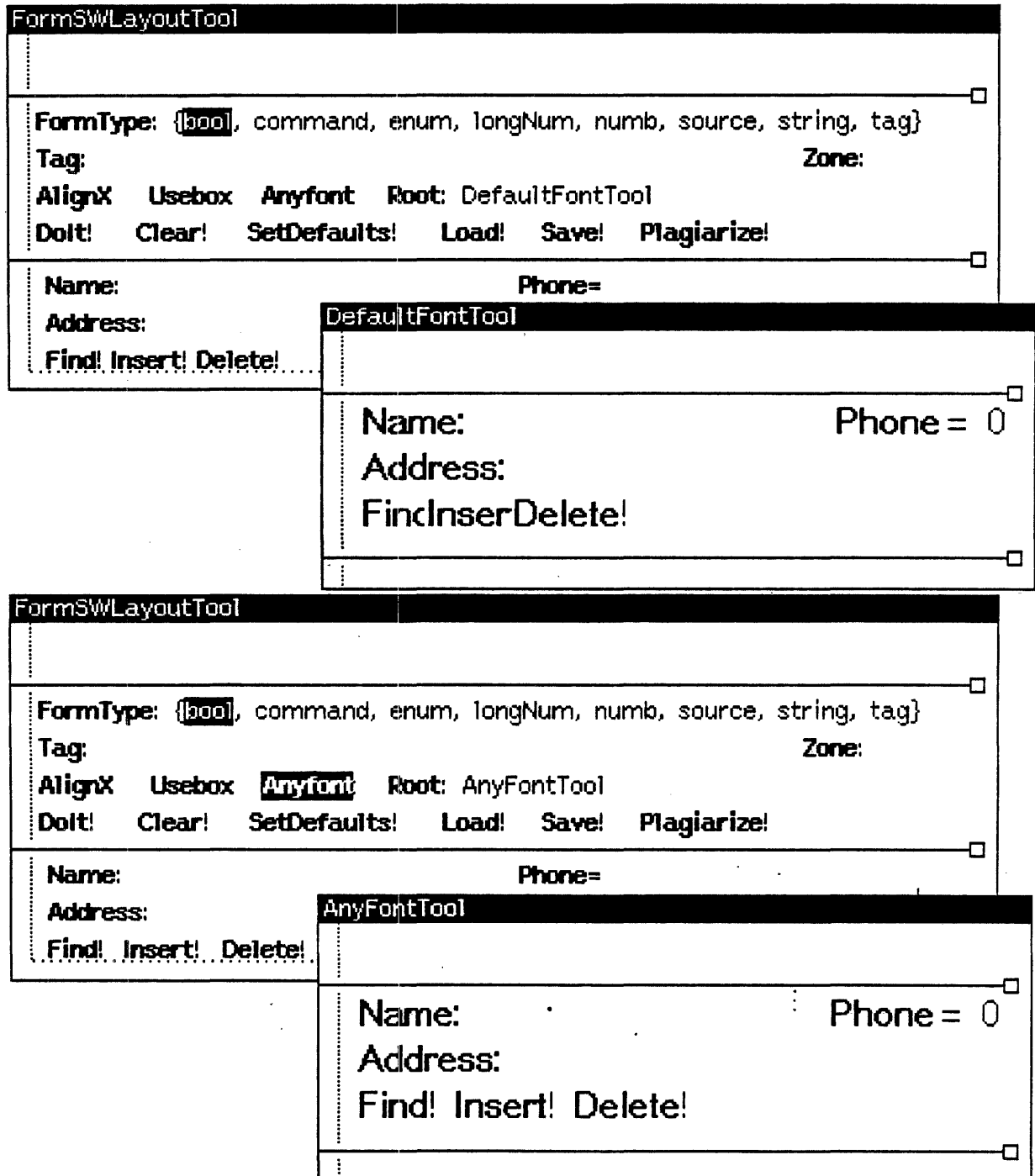
Name: Phone=
 Address:
 Find! Insert! Delete!

UseBoxTool

Name: Phone= 0
 Address:
 Find! Insert! Delete!

Using the FormSWLayoutTool

AnyFont



Dolt! / Root:

FormSWLayoutTool	
FormType: {bool}, command, enum, longNum, numb, source, string, tag}	
Tag:	Zone:
AlignX Usebox Anyfont Root: SimpleTool	
Dolt! Clear! SetDefaults! Load! Save! Plagiarize!	
Verbose	This is a Phone Book tool
Find! Insert! Delete!	
Name:	Phone=
Address:	Building: {fix the enums}
File: <CoPilot>WD>SimpleTool.mesa	
Create OPos Edit Load Empty Save Time Store Reset Split Match	
All! S! RS! <:	SR! R! <:
<pre>-- SimpleTool.mesa -- Create by FormSWLayoutTool on 31-Jul-87 16:02 DIRECTORY Exec, Format, FormSW, Heap, Process, Put, TextSource, Tool, ToolWindow, Window; SimpleTool: MONITOR IMPORTS Exec, FormSW, Heap, Process, Put, Tool = { DataHandle: TYPE = LONG POINTER TO Data; Data: TYPE = MACHINE DEPENDENT RECORD [msgSW(0): Window.Handle < NIL, formSW(2): Window.Handle < NIL, fileSW(4): Window.Handle < NIL, verbose(6): BOOLEAN < FALSE,</pre>	

Clear!

FormSWLayoutTool	
FormType: {0000}, command, enum, longNum, numb, source, string, tag} <input type="checkbox"/>	
Tag:	Zone:
AlignX Usebox Anyfont Root: SimpleTool	
DoIt! Clear! SetDefaults! Load! Save! Plagiarize! <input type="checkbox"/>	

SetDefaults!

FormSWLayoutTool

FormType: {0000}, command, enum, longNum, numb, source, string, tag}
 □

Tag: **Zone:**

AlignX **Usebox** **Anyfont** **Root:** SimpleTool

Dolt! **Clear!** **SetDefaults!** **Load!** **Save!** **Plagiarize!**
□

Verbose **This is a Phone Book tool**

Find! **Insert!** **Delete!**

Name: **Phone=**

Address: **Building:** {fix the enums}

Defaults

Close!

[Defaults for new items]

ReadOnly **Invisible** **DrawBox** **HasContext**

[Enumerated defaults]

CopyChoices: **Feedback:** {one}

[(Long) Number defaults]

Radix: {decimal} **NotNegative** **Signed** **BoxWidth= 64**

(Long)Default= 2147483647 **(Short)Default= 32767**

[String defaults]

InTea: **Feedback:** {normal} **BoxWidth= 0**

[MACHINE DEPENDENT RECORD]

Starting word= 0

[Global Things]

EnumType: FormItems **HandleName:** data

ProcName: MakeForm

StuffString:

Load! / Save!

FormSWLayoutTool	
FormType: {bool}, command, enum, longNum, numb, source, string, tag}	
Tag:	Zone:
AlignX Usebox Anyfont Root: SimpleTool	
Dolt! Clear! SetDefaults! Load! Save! Plagiarize!	
Verbose	Phone Book Tool
Find! Insert! Delete!	
Name:	Phone=
Address:	Building: {fix the enums}
File: <CoPilot>WD>SimpleTool.by	
Create OPos Edit Load Empty Save Time Store Reset Split Match Destroy	
All! S! RS! <<	SR! R! <<
<pre> Global[enumType: FormItems, handleName: data, procName: MakeForm]; BooleanItem[tag: "Verbose", enumName: verbose, place: [6, 2], switch: verbose]; TagOnlyItem[tag: "Phone Book Tool", enumName: phoneBookTool, place: [204, 2]]; CommandItem[tag: "Find", enumName: find, place: [6, 19], proc: Find]; CommandItem[tag: "Insert", enumName: insert, place: [54, 19], proc: Insert]; CommandItem[tag: "Delete", enumName: delete, place: [114, 19], proc: Delete]; StringItem[tag: "Name", enumName: name, place: [6, 36], inHeap: TRUE, string: name]; NumberItem[tag: "Phone", enumName: phone, place: [204, 36], signed: FALSE, value: phone]; StringItem[tag: "Address", enumName: address, place: [6, 53], inHeap: TRUE, string: address]; EnumeratedItem[tag: "Building", enumName: building, place: [204, 53], choices: building, value: building]; -- Waitaminnit [buddy, 43] "fix the enums" </pre>	

Plagiarize!

```

FormSWLayoutTool
-----
FormType: {bool, command, enum, longNum, numb, source, string, tag}
Tag:
AlignX Usebox Anyfont Root: SimpleTool
Dolt! Clear! SetDefaults! Load! Save! Plagiarize!
-----
Expand! Compile! Bind! Run! Go! Options!
Compile:
Bind:
Run:
Log: {Compiler}
-----
Command Central 12.3 of 17-May-85 12:40:38
SimpleTool/be~ju.....lines: 154, code: 1406, links: 25, frame: 8, time: 56
Elapsed time: 1:00
-----
Expand! Compile! Bind! Run! Go! Options!
Compile: SimpleTool
Bind:
Run:
Log: {Compiler}
-----
Mesa Compiler 12.3 of 24-Sep-84 11:45:20
31-Jul-87 16:22:59

Command: SimpleTool
SimpleTool.mesa
lines: 154, code: 1406, links: 25, frame: 8, time: 56

```

Zone:

FormSWLayoutTool

FormType: {bool}, command, enum, longNum, numb, source, string, tag} □
Tag: Zone:
AlignX **Usebox** **Anyfont** **Root:** SimpleTool
DoIt! **Clear!** **SetDefaults!** **Load!** **Save!** **Plagiarize!** □

Verbose **Phone Book Tool**
Find! **Insert!** **Delete!**
Name: **Phone=**
Address: File: <CoPilot>WD>SimpleTool.mesa

Create OPos Edit Load Empty Save Time Store Reset □

All! **S!** **RS!** **<:** **SR!** **R!** **<:** □

data: DataHandle ← NIL;
 wh: Window.Handle ← NIL;
 zone: UNCOUNTED_ZONE ← Heap.systemZone;

FormSWLayoutTool

FormType: {bool}, command, enum, longNum, numb, source, string, tag} □
Tag: Zone: myHeap
AlignX **Usebox** **Anyfont** **Root:** SimpleTool2
DoIt! **Clear!** **SetDefaults!** **Load!** **Save!** **Plagiarize!** □

Verbose **Phone Book Tool**
Find! **Insert!** **Delete!**
Name: **Phone=**
Address: File: <CoPilot>WD>SimpleTool2.mesa

Create OPos Edit Load Empty Save Time Store Reset □

All! **S!** **RS!** **<:** **SR!** **R!** **<:** □

data: DataHandle ← NIL;
 wh: Window.Handle ← NIL;
 myHeap: UNCOUNTED_ZONE ← Heap.Create[initial: 4];

Enum Props

```

FormsWLayoutTool
-----
FormType: {bool}, command, enum, longNum, numb, source, string, tag}
Tag:
AlignX Usebox Anyfont Root: SimpleTool
Dolt! Clear! SetDefaults! Load! Save! Plagiarize!
-----
Verbose Phone Book Tool
Find! Insert! Delete!
Name: Phone=
Address: Building: {fix the enums}
-----
    
```

```

Enumerated Props >> Tag: Building <<
Close! Enum Name: building Tag: Building
readOnly invisible drawBox hasContext
CopyChoices Feedback: {one}
Value: building Proc: ChoiceName: building
Choices: Waitaminnit [buddy, 43] "fix the enums"
    
```

Change Choices field:

```

Enumerated Props >> Tag: Building <<
Close! Enum Name: building Tag: Building
readOnly invisible drawBox hasContext
CopyChoices Feedback: {one}
Value: building Proc: ChoiceName: building
Choices: one two three "more than one word"
    
```

Other Props

FormSWLayoutTool	
FormType: {bool}, command, enum, longNum, numb, source, string, tag} Tag: AlignX Usebox Anyfont Root: SimpleTool Dolt! Clear! SetDefaults! Load! Save! Plagiarize!	Zone: Phone Book Tool
Verbose Find! Insert! Delete! Name: Phone= Address:	Building: {more than one word}
Number Props >> Tag: Phone <<	
Close! Enum Name: phone readOnly invisible drawBox hasContext Signed NotNegative BoxWidth= 64 Default= 32767 Radix: {decimal} Proc: Value: phone	Tag: Phone
String Props >> Tag: Name <<	
Close! Enum Name: name readOnly invisible drawBox hasContext InTag: Feedback: {normal} BoxWidth= 0 FilterProc:	Tag: Name String: name MenuProc:
Command Props >> Tag: Find <<	
Close! Enum Name: find readOnly invisible drawBox hasContext Proc: Find	Tag: Find
Boolean Props >> Tag: Verbose <<	
Close! Enum Name: verbose readOnly invisible drawBox hasContext Proc:	Tag: Verbose Switch: verbose

INDEX

A

Allocating 3-11, 3-20

Allocation 3-1, 3-3, 3-27

C

Call-Back 3-34-3-36

Char 3-39

CR 3-39, 3-41

Create 3-9-3-12, 3-14-3-16, 3-23, 3-27, 3-34, 3-36

CreateMDS 3-9-3-10

CreateUniform 3-9

D

Deallocating 3-14, 3-20

Deallocation 3-27

Delete 3-10, 3-15, 3-23, 3-27

DeleteMDS 3-10

Dummy 3-40

E

Expand 3-16, 3-18

Expanding 3-16-3-17

F

Flush 3-18

FormSWLayoutTool 3-29-3-31, 3-40, 3-43

FREE 3-6, 3-14-3-15, 3-20, 3-22, 3-24-3-25, 3-27, 3-34

FreeNode 3-18

FreeString 3-20

G

GetAttributes 3-18

H

Heap 3-2, 3-6-3-12, 3-14-3-18, 3-23, 3-27, 3-32

L

Line 3-39

M

MakeNode 3-18
 MakeString 3-20
 MDS 3-6-3-8, 3-10, 3-18, 3-27

N

NEW 3-6, 3-11-3-12, 3-14-3-15, 3-20, 3-22-3-25, 3-27, 3-34
 Normal 3-6, 3-8, 3-10

P

Printing 3-38
 Private 3-6, 3-9-3-10
 Prune 3-18
 Public 3-6, 3-8
 Put 3-38-3-39, 3-41

S

Scheduling 3-37
 SEQUENCE 3-21-3-23
 Sequences 3-2, 3-21-3-26, 3-28
 String 3-19-3-20, 3-23, 3-35, 3-39
 Strings 3-2, 3-19-3-20, 3-23-3-25, 3-28
 systemMDSZone 3-6, 3-8
 systemZone 3-6, 3-8, 3-11, 3-18, 3-27, 3-32

T

Text 3-38-3-39, 3-41

U

Uniform 3-6, 3-10

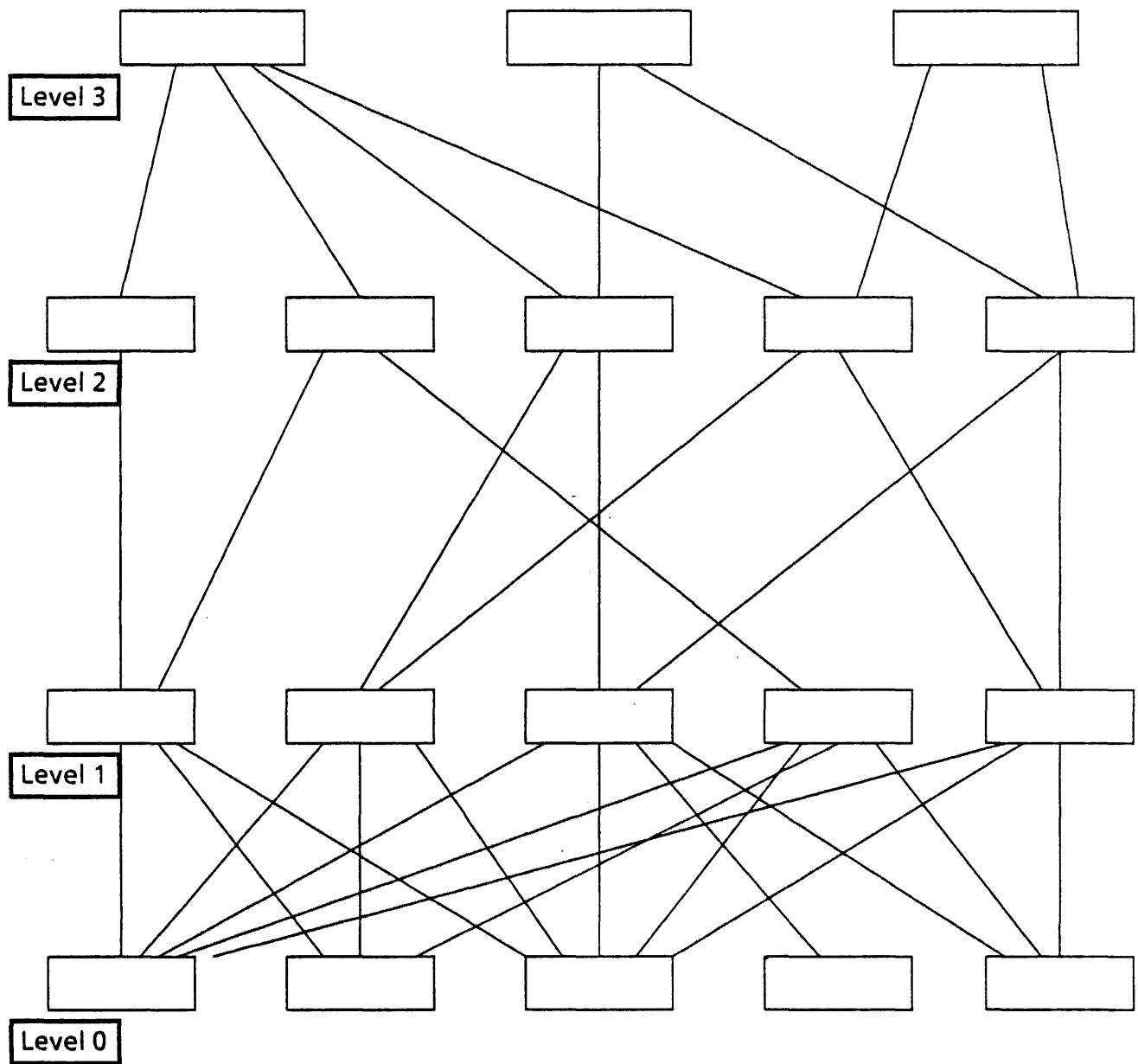
Signals

⋮

Outline

1. Signals
 - a. Signals overview
 - b. Declaring signals
 - c. Generating signals
 - d. Catching signals
 - e. The signal UNWIND

Layered Software



Signals

Types of Signals Crossing Abstraction Boundaries

- Informational signal (e.g. bounds fault)
- Caller error (e.g. bad data, wrong order)
- Abstraction failure (e.g. buffer overflow, no channel)
- Internal Error

Signals vs. Polling

An example of polling:

```
.  
.   
-- -1 is a return code for invalid data.  
n ← GetInput[];  
WHILE n = -1 DO  
  n ← GetInput[]; -- if input too big; try again  
ENDLOOP;
```

An example of signals:

```
.  
.   
n ← GetInput[!InputTooBig => RETRY];  
.   
.
```

The Signaller

When a signal gets "*raised*", normal execution is suspended and control is transferred to the *Signaller*, which is part of Mesa's run-time support. It is the Signaller's responsibility to find and execute bodies of code that recognize the signal that was raised. The bodies of code that recognize the signal are called *catch phrases*.

Catch phrases are visited in reverse order of the standard scope rules; that is, inner blocks are visited first, then outer blocks. The order at the procedure level goes from the most recently called procedure to the least recently called procedure, beginning with the procedure that raised the signal.

The path that the Signaller follows in order to find catch phrases that recognize a signal is called the *propagation path*.

3 Signal Operations

- Declaring a signal
- Generating a signal (a.k.a. Raising a signal)
- Recognizing a signal

Declaring a Signal

If you are going to use a particular signal only within one module, you should declare it only within that module. If you are going to use the signal across many modules, the signal must be declared in an Interface. (Unless otherwise specified, the term *signal* may stand for both SIGNAL and ERROR.)

Syntax for Signals:

```
<signalname>: SIGNAL[<args if any>] RETURNS[<results if any>];
```

Example:

```
mySignal: SIGNAL[s: LONG STRING] RETURNS[ns: LONG STRING];
```

Syntax for Errors:

```
<errorname>: ERROR[<args if any>] RETURNS[<results if any>];
```

Example:

```
myError: ERROR[type: ErrorType];
```

```
ErrorType: TYPE = {...,maxSizeExceeded, invalidParameters,  
    invalidSize, insufficientSpace, otherError....};
```

**** ERRORS cannot be RESUMEd.**

Initializing a signal

In the implementation module, initialize the body of the signal to the keyword `CODE`:

Example:

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

The actual code for the signal will be dynamically bound at runtime.

Generating a Signal

If the signal doesn't return results:

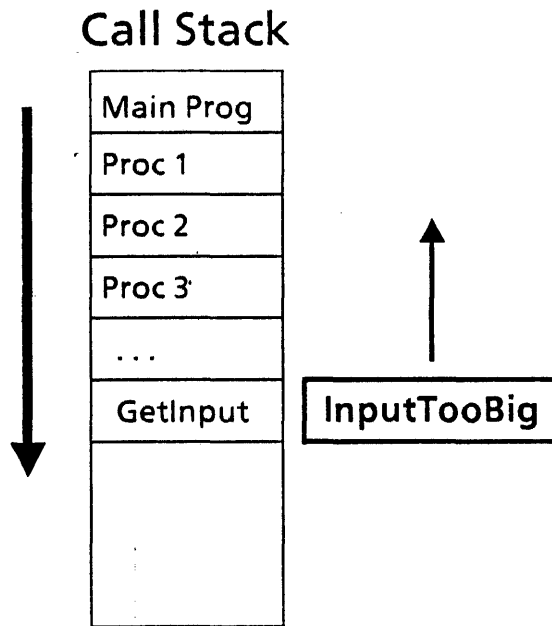
```
SIGNAL <signalname>[<arguments if any>];  
ERROR <errorname or signalname>[<arguments if any>];  
RETURN WITH ERROR <errorname or signalname>  
  [<arguments if any>];  
ERROR;
```

If the signal returns results:

```
<leftside> ← SIGNAL <signalname>[<arguments, if any>];  
<leftside> ← ERROR <errorname or signalname>[<args, if any>];
```


Generating a Signal

```
GetInput: PROCEDURE RETURNS[inputNumber: INTEGER] = {  
  .  
  .  
  IF inputNumber NOT < 1000 THEN  
    SIGNAL InputTooBig;  
  .  
  .  
};
```



Recognizing Signals With Catch Phrases

A propagating signal can be recognized by catch phrases. Catch phrases, in general, contain 2 parts:

- 1) a recognition clause (specifying the signal of interest) and
- 2) an action clause (specifying what should happen if that signal is recognized). Its syntax is similar to a SELECT statement.

Catch phrases can be located in 2 places:

- 1) during a procedure call (tucked in with the argument list) and
- 2) at the beginning of any block (including loops).

Methods of Recognizing a Signal (in an Argument List)

A catch phrase can be included in an argument list as the last item. That includes any procedure call, signal call, START, RESTART, STOP, JOIN, FORK, and WAIT. A ! (exclamation point) in the argument list marks the beginning of a catch phrase.

Examples:

```

.
.
n ← GetInput[!InputTooLittle => BEGIN ... END;
             InputTooBig => RESUME]; -- example without parameters

```

```

OpenChannel[size, handle !InputTooBig, NoChannels => RETRY];
             -- example with parameters

```

Scope: If the signals `InputTooLittle` or `InputTooBig` are raised within `GetInput` they will be recognized by the catch phrase attached to the `GetInput` procedure call.

Similarly, if the signals `InputTooBig` or `NoChannels` are raised within `OpenChannel` they will be recognized by the catch phrase attached to the `OpenChannel` procedure call.

More Examples of Recognizing a Signal (in an Argument List)

Note that the catch phrases can be different each time that a procedure is called.

Example:

```
.  
.
n ← GetInput[!InputTooLittle => BEGIN ... END;
           InputTooBig => RESUME];
.  
.
n ← GetInput[!InputTooBig => RETRY];
.  
.
```

Another Method of Recognizing a Signal (With ENABLE Clauses)

A catch phrase can be included at the beginning of any BEGIN - END block. This is done with an ENABLE Clause.

Example:

```
PlayAGame: PROCEDURE = {  
  IF ... THEN
```

```
    BEGIN
```

```
      ENABLE OutOfMoney => {  
        GetMoreMoney[];  
        RESUME;  
      };
```

```
      <statement 1>;
```

```
      <statement 2>;
```

```
      ...
```

```
      END;
```

```
    <statement 3>;
```

```
    ...
```

```
  };
```

Scope: Signals raised in <statement 1> or <statement 2> will be recognized with this catch phrase. Signals raised in <statement 3> will not be recognized.

More Examples with ENABLE Clauses

An ENABLE Clause can also occur at the beginning of any DO loop.

Example:

THROUGH [1..n]

```
DO
  ENABLE {
    Sig1 => LOOP;
    Sig2, Sig3 => RETRY;
  };
  <statement 1>;
  <statement 2>;
  ...
  ENDLLOOP;
```

<statement 3>;

Scope: Signals raised in <statement 1> or <statement 2> will be recognized with this catch phrase. Signals raised in <statement 3> will not be recognized.

The Scope of ENABLE

```
BEGIN
  <ENABLE clause>
  <Declarations>
  <Statements>
  <ExitsClause>
END;
```

Catch phrases in an ENABLE clause cannot access variables declared in the enclosing BEGIN-END block.

Example:

```
BEGIN
  ENABLE OutOfMoney => {
    IF i < 0 THEN ... -- i is undefined at this point
    GetMoreMoney[];
    RESUME;
  }; -- of the Enable Clause

  i: CARDINAL ← 0; -- i is declared here
  ...
  i ← i + 1;
  ...
END;
```

The Scope of ENABLE (con't)

To get around this, use 2 sets of BEGIN and END:

```
BEGIN
  <Declarations>
  BEGIN
    <ENABLE clause>
    <Statements>
  END;
  <ExitsClause>
END;
```

Example:

```
BEGIN
i: CARDINAL ← 0; -- declare i in an outer block
  BEGIN
    ENABLE OutOfMoney => {
      IF i < 0 THEN ... -- valid to reference i here
      GetMoreMoney[];
      RESUME;
    };
    ...
    i ← i + 1;
    ...
  END;
END;
```


Results of Entering a Catch Phrase

A signal that has been recognized in a catch phrase can either be *rejected* (which means that the signal will continue to propagate up the call stack) or it can be *caught* which means that the signal will stop its propagation).

A catch phrase can reject the signal three different ways:

- 1) explicitly with a REJECT statement,
- 2) implicitly by not being recognized,
- 3) implicitly by falling off the end (not explicitly stopping propagation).

Methods of "*catching*" a signal will be forthcoming.

Rejection Examples

For the signal `InputTooBig`:

1) `InputTooBig` is recognized but *explicitly rejected*:

```
n ← GetInput[!InputTooBig => {  
  IF ... THEN REJECT;  
  ...}];
```

2) `InputTooBig` is *not recognized* therefore, it is implicitly not caught:

```
n ← GetInput[!InputTooLittle => BEGIN .. END];
```

3) `InputTooBig` *fell off the end* because there was no explicit command about where control should go (just 'recognized', but not 'caught'):

```
n ← GetInput[!InputTooBig => {  
  ...  
}];
```

Catchy Statements

RESUME
CONTINUE
RETRY
LOOP
EXIT
GOTO

If you want to “catch” a signal (stop its propagation), you must have one of these six statements at the end of a catch phrase. These statements tell the Signaller where to transfer program control. If none of these statements are used in a catch phrase, it is assumed that the Signaller is to remain in control and should continue along the propagation path looking for more catch phrases.

Any signal that is not *caught* within the call stack will be caught by the debugger. The message that you will see is “Uncaught Signal . . .”.

RESUME Statement

Treat the signal call as a procedure call; return to the point the signal was raised.

Example:

```

BadData: SIGNAL[...] RETURNS[...];

GetData: PROC[...] RETURNS[...] = {
  ...
  IF ... THEN
    n ← SIGNAL BadData[...]; -- control returns here after the RESUME
    ...                       -- with n getting assigned the returned value.
  }.

-- mainline code
...
n ← GetData[... !BadData =>          can put code here
  RESUME[ -- with some number that will get assigned to n -- ]];

```

Control returns to where the signal was raised, but with a new value (if there is one).

Note: There are two more-detailed examples of RESUME in later slides.

CONTINUE Statement

Goes to the statement following the one to which the catch phrase belongs.

- Catch phrase in an argument list:

```
...
temp ← MostRecentTemp[!NoTemp => CONTINUE];
<statement 1>
...
```

Control passes to <statement 1>.

- Catch phrase in an Enable Clause:

```
... -- code fragment in a program
IF ... THEN
```

```
BEGIN
ENABLE NoInterest => CONTINUE;
<statement 1>;
n ← GetData[]; -- NoInterest may be raised here
<statement 2>;
END;
```

```
<statement 3>;
```

```
...
```

Control passes to <statement 3>.

CONTINUE Statement (con't)

- Catch phrase in a Loop:

... -- code fragment in a program

```
WHILE ... DO
```

```
    ENABLE NoInterest => CONTINUE;  
    <statement 1>;  
    n ← GetData[]; -- NoInterest may be raised here.  
    <statement 2>;  
    ...
```

```
ENDLOOP;
```

```
<statement 3>;
```

```
...
```

Control passes to the next iteration.

RETRY Statement

Go back to the beginning of the statement to which this catch phrase belongs.

- Catch phrase in an argument list:

```
... -- code fragment in a program
tries ← 0;
n ← GetHandle[!NoneLeft => {
    tries ← tries + 1;
    IF tries < 8 THEN RETRY ELSE GOTO errorMsg; }];
...
```

The call to GetHandle is executed again.

- Catch phrase in an Enable Clause:

```
... -- code fragment in a program
IF ... THEN {
    ENABLE IncorrectResults => RETRY;
    <statement 1>;
    <statement 2>; -- IncorrectResults may be raised here.
    <statement 3>;
    ...
};
<statement 4>;
...
```

The block (starting with <statement 1>) is started over.

RETRY Statement (con't)

- Catch phrase in a Loop:

```
... -- code fragment in a program  
WHILE ... DO
```

```
ENABLE Sig1 => RETRY;  
<statement 1>;  
n ← GetData[]; --Sig1 could be raised here  
<statement 2>;  
...
```

```
ENDLOOP;  
<statement 3>;  
...
```

The iteration in which the SIGNAL was raised (the current iteration) is started over (starting with <statement 1>).

LOOP, EXIT and GOTO (revisited)

LOOP and EXIT are only meaningful within loops.

- Example of LOOP:

```
... -- code fragment in a program
WHILE ... DO
    ENABLE Sig1 => LOOP;
    <statement 1>;
    n ← GetData[]; -- Sig1 could be raised here
    <statement 2>;
    ...
ENDLOOP;
<statement 3>;
...
```

Control passes to the next iteration.

- Example of EXIT:

```
... -- code fragment in a program
WHILE ... DO
    ENABLE Sig1 => EXIT;
    <statement 1>;
    n ← GetData[]; -- Sig1 could be raised here
    <statement 2>;
    ...
ENDLOOP;
<statement 3>;
...
```

Control passes to <statement 3>.

LOOP, EXIT and GOTO (con't)

- Example of GOTO in a BEGIN - END block:

```

... -- code fragment in a program
IF .. THEN {
    ENABLE IncorrectResults => GOTO punt;
    <statement 1>;
    <statement 2>; -- IncorrectResults may be raised here.
    ...
};
<statement 4>;
...
EXITS
    punt => {...};
...

```

Control jumps to the EXITS clause looking for the arm labelled punt.

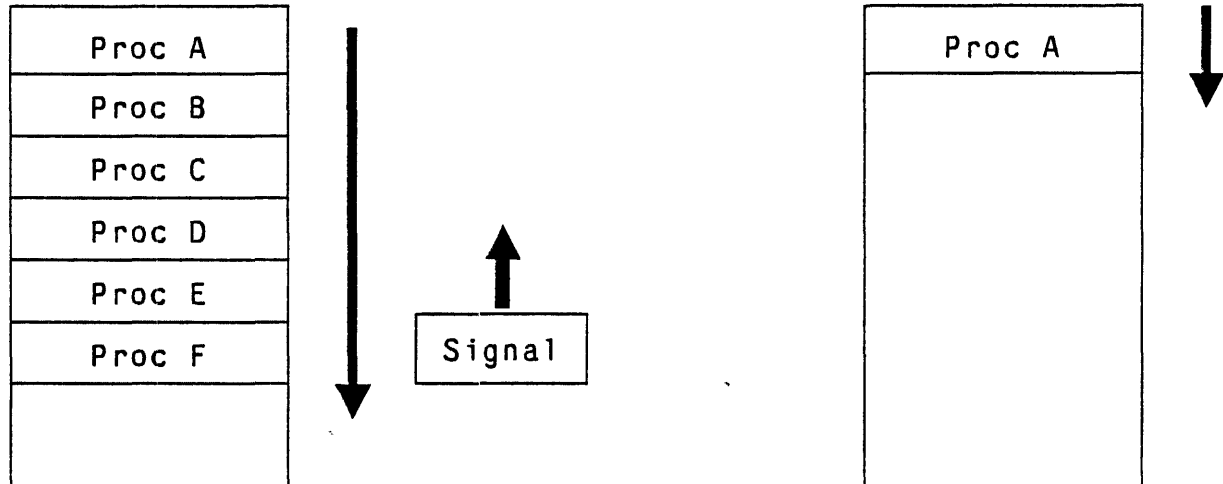
- Example of GOTO in a Loop:

```

... -- code fragment in a program
WHILE ... DO
    ENABLE NoInterest => GOTO getOut;
    <statement 1>;
    n ← GetData[]; -- NoInterest may be raised here.
    <statement 2>;
    ...
REPEAT
    getOut => NULL;
ENDLOOP;
<statement 3>;
...

```

UNWIND



In the example above, assume procedure F raises a signal. Procedure A catches that signal and does a CONTINUE (which resumes control in the frame which the signal was caught), so procedures B, C, D, E, and F will be removed from the call-stack.

A procedure that is going to be removed from the call-stack should be notified of that fact so that it may have a chance to clean up after itself (e.g. close files, deallocate storage, etc.).

Mesa provides the signal UNWIND as a mechanism of notifying procedures that they will be removed from the call-stack. This allows a procedure to restore invariants if that procedure suddenly finds itself being removed as the result of a lower level signal being caught and handled by a catch phrase in a procedure higher in the call-stack.

The path of UNWIND

When a catch phrase has recognized a signal, and is about to do a GOTO, EXIT, LOOP, RETRY, or CONTINUE, the UNWIND signal is raised at the point the original signal was raised.

The propagation path for UNWIND is the same as that of the original signal. After UNWIND is passed to the last catch phrase in a given procedure frame, the frame is deallocated. When UNWIND reaches the "scoping boundary" of the catch frame that accepted the original signal, it stops and the jump corresponding to the statements above takes place, destroying the last catch frame in the process. (The UNWIND does not enter the catch phrase that accepted the original signal.)

Since an UNWIND signal is just like any other signal, it may be recognized by procedures that wish to restore invariants before being removed from the call-stack.

The propagation of UNWIND should never be stopped by one of the six magic "catchy" keywords.

```
... -- Code fragment
ENABLE UNWIND =>
  BEGIN
    ... -- close files, etc --
  END;
...
```

Summary

Declare the signal (or use a pre-declared signal)

Generate / Raise the signal

Recognize the signal (at which point you can perform some action or make some test)

Catch or Reject the signal (IF the signal is caught with something other than RESUME, then the Signaller raises the signal UNWIND *before* transferring control.)

If UNWIND was raised, it will follow the original propagation path.

Transfer program control to the pre-specified place.

Full Signal Example

```
InputTooBig: SIGNAL[number: INTEGER]
  RETURNS[newNumber: INTEGER] = CODE;

n: INTEGER;

GetInput: PROCEDURE RETURNS[inputNumber: INTEGER] = {
  ...
  inputNumber ← ... --inputNumber gets some value--
  IF inputNumber NOT < 1000 THEN
    inputNumber ← SIGNAL InputTooBig[inputNumber];
  ... --do something interesting
};

-- start of mainline code

... -- code fragment
n ← GetInput[!InputTooBig =>
  IF number < 1100 THEN
    RESUME[999]
  ELSE {
    -- Message user about range of Input--
    RETRY }];
...

```

Another Signal Example

-- Just FYI (it is already declared in the String Interface)

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

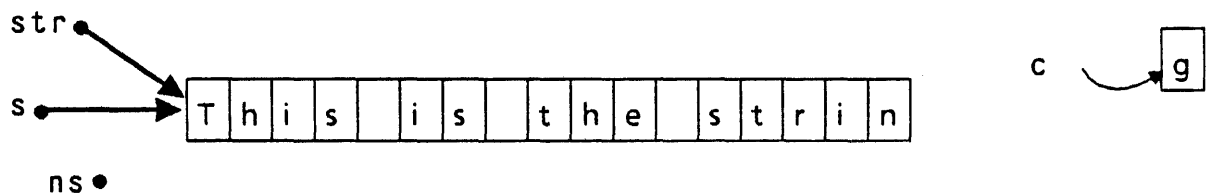
... -- code fragment in a program

```
str: LONG STRING ← GetStringFromSomewhere[];
c: CHARACTER ← GetCharFromSomewhereElse[];
```

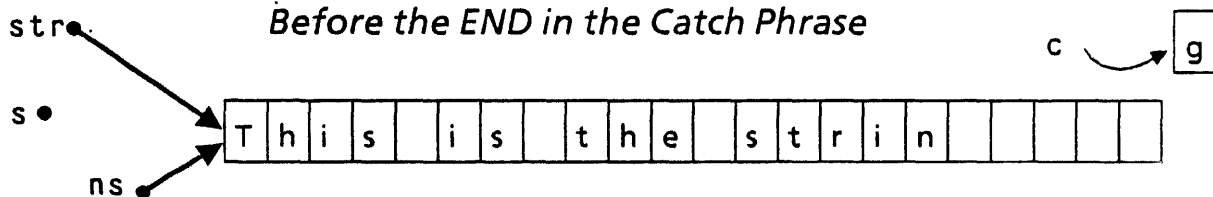
```
String.AppendChar[str, c !String.StringBoundsFault =>
  BEGIN
    ns ← String.MakeString[z, s.maxlength + 5]; -- temporary
    String.Copy[from:s, to:ns]; -- Copy old string to new string
    String.FreeString[z, s]; -- Deallocate old string
    str ← ns; -- set global pointer to new string
    RESUME[str]; -- RESUME with the new string
  END];
```

...

After the BEGIN in the Catch Phrase



Before the END in the Catch Phrase



References

J. J. Horning, *Programming Languages*, in T. Anderson and B. Randell (eds.) *COMPUTING SYSTEMS RELIABILITY*, Cambridge University Press, 1979.

Roy Levin, *Program Structures for Exceptional Condition Handling*, Ph. D. thesis (available as technical report), Department of Computer Science, Carnegie-Mellon University, 1977.

J. B. Goodenough, *Exception handling: Issues and a proposed notation*, COMM. ACM 18, no. 12, 1975.

F. Christian, *Exception Handling and Software Fault Tolerance*, 10th Symposium on Fault Tolerant Computing Systems, Kyoto, 1980.

Mesa Streams

Outline

1. Streams
 - a. Streams overview
 - b. Declaring Streams and initializing them
 - c. Possible Errors
 - d. I/O Procedures
 - e. Stream Blocks
 - f. Deleting Streams

Mesa Streams

Streams provide *sequential* access to data (a way to read or write a "stream" of bytes, words, etc.) [There are other facilities for handling random access of files by mapping segments of files to virtual memory. See the MSegment interface documented in the Mesa Programmer's Manual for more information.]

Streams are device independent. (Floppy, tape drive, disk file, etc.) However, for the purposes of this lecture, a stream is a connection from a program to a file on the local disk in order to read and write that file.

Necessary Stream Operations

- Declare a stream variable
- Acquire the stream handle (initialize the connection)
- Use the stream (get data and put data)
- Delete the stream

Declaring Streams

```
inStream: MStream.Handle;
```

```
inStream: Stream.Handle;
```

MStream.Handle and Stream.Handle are equivalent types. You can use them interchangeably. This is because:

```
MStream.Handle: TYPE = Stream.Handle;
```

MStream and Stream

MStream is a Tajo interface (documented in the *Mesa Programmer's Manual*.)

Stream is a Pilot interface (documented in the *Pilot Programmer's Manual*.)

To attach a stream to a local disk file, use the **MStream** interface. All device-specific information will be encapsulated in the stream handle. **MStream** is usually used in conjunction with the **MFile** interface, also documented in the *Mesa Programmer's Manual*.

Once you have set up the stream, use the **Stream** interface to do your input and output. Once you have a connection, a stream to a disk file is just like a stream to any other device.

Acquiring Stream Handles

Two step process:

- Acquire a handle to the file (MFile procedures)
- Attach a stream to that file (MStream procedures)

1) To acquire a handle to a specified file, use this MFile procedure (remember, this is just the definition!):

```
MFile.Acquire: PROCEDURE [
  name: LONG STRING,
  access: MFile.Access,
  -- MFile.Access: TYPE = {readOnly, writeOnly, readWrite, ...}
  release: MFile.ReleaseData, -- explained later
  mightWrite: BOOLEAN ← FALSE,
  initialLength: MFile.InitialLength ← MFile.dontCare,
  type: MFile.Type ← unknown]
  RETURNS [MFile.Handle];
```

2) To attach a stream to that file, use this MStream procedure:

```
MStream.Create: PROCEDURE [
  file: MFile.Handle,
  release: MStream.ReleaseData, -- explained later
  options: Stream.InputOptions ← Stream.defaultInputOptions,
  streamBase: File.PageNumber ← 0]
  RETURNS [stream: MStream.Handle];
```

Example:

```
fileName: LONG STRING ← "junk"L;
file: MFile.Handle ← MFile.Acquire[
  name: fileName, access: readWrite,
  release: [NIL, NIL] !MFile.Error => GOTO exit];
stream: MStream.Handle ← MStream.Create[
  file: file, release: [NIL, NIL] !MStream.Error => GOTO exit];
```

Acquiring Stream Handles to Temp Files

The MFile interface provides a mechanism of providing a temporary, anonymous file, if that is what your program needs. Once you have acquired a file handle to a temporary file, use the same MStream procedure shown earlier to attach a stream to it.

```
MFile.AcquireTemp: PROCEDURE [  
  type: MFile.Type, -- MFile.Type: TYPE = {text, binary, directory, ...}  
  initialLength: MFile.InitialLength ← MFile.dontCare,  
  volume: Volume.ID ← [0, 0, 0, 0, 0]]  
  RETURNS [MFile.Handle];
```

Example:

```
tempFile: MFile.Handle ← MFile.AcquireTemp[type: text  
  !MFile.Error => GOTO exit];  
tempStream: MStream.Handle ← MStream.Create[file: tempFile,  
  release: [NIL, NIL] !MStream.Error => GOTO exit];
```


ReleaseData

The acquire procedures in both MStream and MFile have a parameter of type `ReleaseData`. (Definitions are below.) `ReleaseData` is used to specify how your program intends to share files with other programs.

```
MFile.ReleaseData: TYPE = RECORD [  
  proc: MFile.PleaseReleaseProc ← NIL,  
  clientInstanceData: LONG POINTER ← NIL];
```

```
MStream.ReleaseData: TYPE = RECORD [  
  proc: MStream.PleaseReleaseProc ← NIL,  
  clientInstanceData: LONG POINTER ← NIL];
```

Unless you are planning to share the file with other processes, you should just pass the record `[NIL, NIL]` for the `ReleaseData` parameter.

Accelerators

When you know the name of the file that you want to attach a stream to, and you want to have one of the standard accesses, you can use an "accelerator" procedure to acquire the file handle and attach a stream to that file:

```
MStream.ReadOnly: PROCEDURE[
  name: LONG STRING, release: MStream.ReleaseData]
  RETURNS [MStream.Handle];
```

```
MStream.ReadWrite: PROCEDURE[
  name: LONG STRING, release: MStream.ReleaseData,
  type: MFile.Type]
  RETURNS [MStream.Handle];
```

```
MStream.WriteOnly: PROCEDURE[
  name: LONG STRING, release: MStream.ReleaseData,
  type: MFile.Type]
  RETURNS [MStream.Handle];
```

Example:

```
fileName: LONG STRING ← "name"L;
foo: MStream.Handle ← MStream.ReadWrite[
  name: fileName, release: [NIL, NIL], type: text
  !MStream.Error, MFile.Error => GOTO exit];
```

Errors

MStream and MFile procedures can potentially raise signals that you need to handle. These signals are:

From the MFile Interface:

```
MFile.Error: ERROR [file: MFile.Handle, code: MFile.ErrorCode];
```

```
MFile.ErrorCode: TYPE = MACHINE DEPENDENT {noSuchFile,  
conflictingAccess, insufficientAccess, directoryFull,  
directoryNotEmpty, illegalName, noSuchDirectory, ...};
```

From the MStream Interface:

```
MStream.Error: ERROR [  
stream: Stream.Handle, code: MStream.ErrorCode];
```

```
MStream.ErrorCode: TYPE = MACHINE DEPENDENT {invalidHandle,  
indexOutOfRange, invalidOperation, fileTooLong,  
fileNotAvailable, invalidFile, other(LAST[CARDINAL])};
```

Example:

```
foo: MStream.Handle ← MStream.ReadWrite[  
name: "someName", release: [NIL, NIL], type: text  
!MFile.Error => {  
  SELECT code FROM  
    conflictingAccess, insufficientAccess => GOTO AccessProblems;  
    illegalName => GOTO BadName;  
  ENDCASE => GOTO AllOther};  
MStream.Error => GOTO StreamProblems];
```

Stream I/O

Use the **Stream** Interface to perform any I/O with your stream.

For getting information from the stream:

```
Stream.GetByte: PROCEDURE[sH: Stream.Handle]  
  RETURNS [byte: Stream.Byte];
```

```
Stream.GetChar: PROCEDURE[sH: Stream.Handle]  
  RETURNS [char: CHARACTER];
```

```
Stream.GetWord: PROCEDURE[sH: Stream.Handle]  
  RETURNS [word: Stream.Word]; -- word = 2 bytes
```

For sending information to the stream:

```
Stream.PutByte: PROCEDURE[sH: Stream.Handle,  
  byte: Stream.Byte];
```

```
Stream.PutChar: PROCEDURE[sH: Stream.Handle,  
  char: CHARACTER];
```

```
Stream.PutWord: PROCEDURE[sH: Stream.Handle,  
  word: Stream.Word]; -- word = 2 bytes
```

```
Stream.PutString: PROCEDURE[sH: Stream.Handle,  
  string: LONG STRING, endRecord: BOOLEAN ← FALSE];
```

Reaching the End of the Stream

All of the `Stream.Get*` I/O procedures will raise a signal if the end of the stream (end of file) is reached. You should catch this signal.

```
Stream.EndOfStream: SIGNAL[nextIndex: CARDINAL];
```

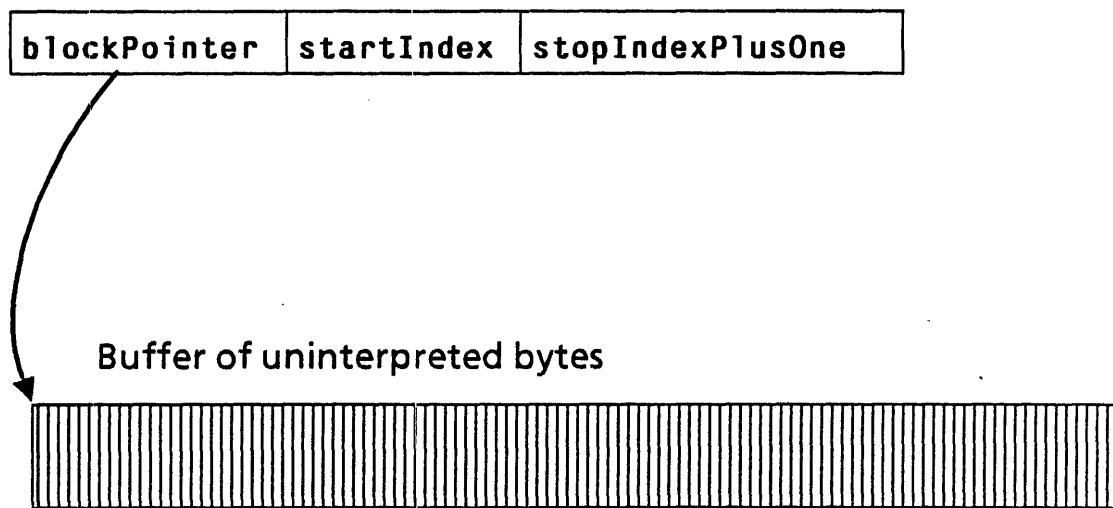
Example:

```
DO
  ch: CHARACTER;
  ch ← Stream.GetChar[inStream !Stream.EndOfStream => EXIT];
  Stream.PutChar[outStream,ch];
ENDLOOP;
```

Stream Blocks

Stream Blocks provide a method of transferring arbitrary data structures. These structures are read into or out of a buffer of uninterpreted bytes. You can then LOOPHOLE the information into the structure of your choice.

The `Stream.Block` is a record that contains a pointer to a buffer, the starting position within that buffer, and the stopping position.



Stream Block Definitions

Stream.Block: TYPE = Environment.Block;

Environment.Block: TYPE = RECORD[
 blockPointer: LONG POINTER TO PACKED ARRAY [0..0)
 OF Environment.Byte,
 startIndex, stopIndexPlusOne: CARDINAL];

Stream.CompletionCode: TYPE = {normal, endRecord,
 sstChange, endOfStream, attention, timeout};

Stream.GetBlock: PROCEDURE[sH: Stream.Handle, block: Stream.Block]
 RETURNS [bytesTransferred: CARDINAL, why: Stream.CompletionCode,
 sst: Stream.SubsequenceType];

Stream.PutBlock: PROCEDURE[sH: Stream.Handle,
 block: Stream.Block, endRecord: BOOLEAN ← FALSE];

Block Example

```

EmployeeInfo: TYPE = MACHINE DEPENDENT RECORD[
  name(0): PACKED ARRAY[0..30) OF CHARACTER,
  ssn(15): LONG CARDINAL,
  employeeNumber(17): LONG CARDINAL,
  gradeLevel(19): CARDINAL];

RecordSize: CARDINAL = SIZE [EmployeeInfo] * 2;-- SIZE returns # of words
rec: LONG POINTER TO EmployeeInfo;

buffer: PACKED ARRAY[0..RecordSize) OF Environment.Byte;
block: Stream.Block ← [@buffer, 0, RecordSize];

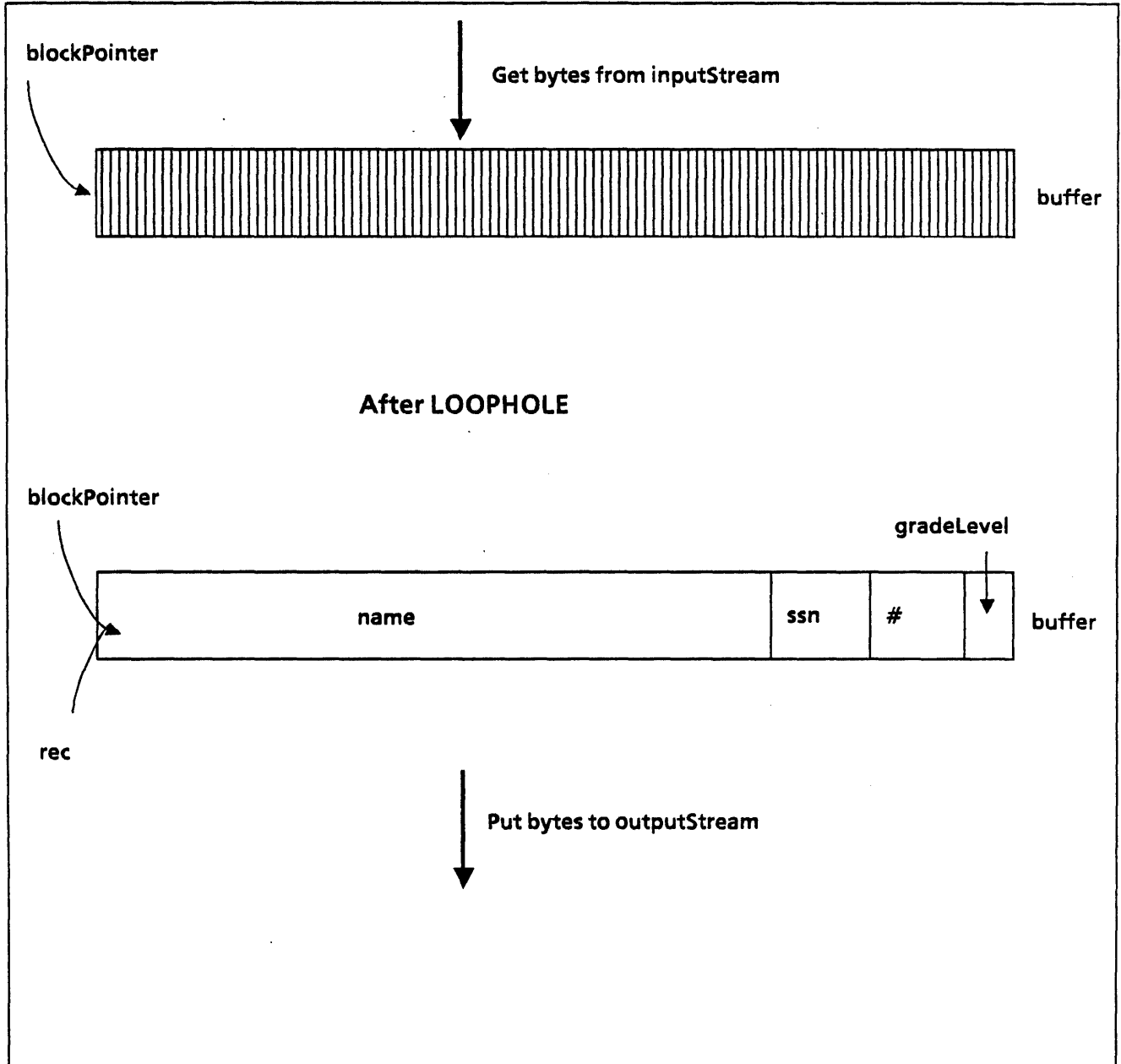
inputStream: Stream.Handle ← MStream.ReadOnly[name: "input",
  release: [NIL, NIL] !MStream.Error, MFile.Error => GOTO exit];

outputStream: Stream.Handle ← MStream.ReadWrite[name: "output",
  release: [NIL, NIL]!MStream.Error, MFile.Error => GOTO exit];

... -- code fragment
completionCode: Stream.CompletionCode ← normal;
UNTIL completionCode = endOfStream DO
  [block.stopIndexPlusOne, completionCode, ] ←
    Stream.GetBlock[inputStream, block];
  rec ← LOOPHOLE[block.blockPointer];
  rec.gradeLevel ← rec.gradeLevel + 3;
  Stream.PutBlock[outputStream, block, TRUE];
ENDLOOP;
...

```


Block Example



Random Access

Stream.Position: TYPE = LONG CARDINAL; -- zero is the beginning of the file

Stream.GetPosition: PROCEDURE [sH: Stream.Handle]
RETURNS [position: Stream.Position];

Stream.SetPosition: PROCEDURE [sH: Stream.Handle,
position: Stream.Position];

Example:

```
Advance: PROC[stream: Stream.Handle, amount: Stream.Position]
  RETURNS[newPos: Stream.Position] = {
    position: Stream.Position ← Stream.GetPosition[sH: stream];
    newPos ← position + amount;
    Stream.SetPosition[sH: stream, position: newPos];
  };
```

Deleting Streams

Always delete the stream handle when you are through with it. This will release the file handle, so that other processes may use the file. Regardless of how the stream handle was acquired, use this procedure:

```
Stream.Delete: PROCEDURE[sH: Stream.Handle];
```

Example:

```
myStream: Stream.Handle ← MStream.ReadOnly[...];  
-- Do I/O --  
Stream.Delete[myStream];  
myStream ← NIL;
```

Stream Example

This example copies the contents of one file to another.

```
... -- code fragment
inStream, outStream: MStream.Handle;

inStream ← MStream.ReadOnly[
  name: "letter.txt"L, release: [NIL, NIL]
  !MFile.Error, MStream.Error => GOTO exit];

outStream ← MStream.WriteOnly[
  name: "letter.out"L, release: [NIL,NIL], type: text
  !MFile.Error, MStream.Error => GOTO exit];

DO
  Stream.PutChar[outStream,
    Stream.GetChar[inStream !Stream.EndOfStream => EXIT]];
  ENDLOOP;

Stream.Delete[inStream];
Stream.Delete[outStream];
inStream ← outStream ← NIL;
...
```

INDEX

A	
Accelerators	4-44
Acquire	4-38, 4-41
AcquireTemp	4-42
B	
Block	4-48 - 4-51
Blocks	4-36, 4-48
C	
Catching	4-2
Catchy	4-21
CompletionCode	4-49 - 4-50
CONTINUE	4-21, 4-23 - 4-24, 4-29 - 4-30
Create	4-41 - 4-42
D	
Declaring	4-2, 4-7 - 4-8, 4-36, 4-39
Delete	4-38, 4-53 - 4-54
Deleting	4-36, 4-53
E	
ENABLE	4-15 - 4-18, 4-23 - 4-28, 4-30
EndOfStream	4-47, 4-54
Error	4-4, 4-8, 4-10, 4-41 - 4-42, 4-44 - 4-45, 4-50, 4-54
Errors	4-8, 4-36, 4-45
Example	4-32 - 4-33, 4-50 - 4-54
EXIT	4-21, 4-27 - 4-28, 4-30, 4-47, 4-54
G	
Generating	4-2, 4-7, 4-10 - 4-11
GetBlock	4-49 - 4-50
GetByte	4-46
GetChar	4-46 - 4-47, 4-54
GetPosition	4-52
GetWord	4-46
GOTO	4-21, 4-25, 4-27 - 4-28, 4-30, 4-41 - 4-42, 4-44, 4-50, 4-54
H	
Handle	4-39, 4-41 - 4-42, 4-44 - 4-46, 4-49 - 4-50, 4-52 - 4-54

I	
Initializing	4-9
L	
Loop	4-16, 4-21, 4-24, 4-26 - 4-28, 4-30
M	
MFile	4-40 - 4-45, 4-50, 4-54
MStream	4-39 - 4-45, 4-50, 4-53 - 4-54
O	
Operations	4-7, 4-38
P	
Polling	4-5
Position	4-52
PutBlock	4-49 - 4-50
PutByte	4-46
PutChar	4-46 - 4-47, 4-54
PutString	4-46
PutWord	4-46
R	
Raising	4-7
Random	4-52
ReadOnly	4-44, 4-50, 4-53 - 4-54
ReadWrite	4-44, 4-50
Recognizing	4-7, 4-12 - 4-15
REJECT	4-19 - 4-20
Rejection	4-20
ReleaseData	4-41, 4-43 - 4-44
Results	4-19
RESUME	4-13 - 4-15, 4-17 - 4-18, 4-20 - 4-22, 4-31 - 4-33
RETRY	4-5, 4-13 - 4-14, 4-16, 4-20 - 4-21, 4-25 - 4-26, 4-30, 4-32

S

Scope 4-13, 4-15 - 4-18
SetPosition 4-52
SIGNAL 4-8 - 4-11, 4-22, 4-26, 4-32 - 4-33, 4-47
Signaller 4-6, 4-21, 4-31
Signals 4-1 - 4-2, 4-4 - 4-5, 4-8, 4-12, 4-15 - 4-16
Stack 4-11
Stream 4-36, 4-38 - 4-42, 4-45 - 4-50, 4-52 - 4-54
Streams 4-35 - 4-37, 4-39, 4-53

U

UNWIND 4-2, 4-29 - 4-31

W

WriteOnly 4-44, 4-54

Processes and Concurrency

Outline

1. Processes and Concurrency
 - a. Concurrent execution
 - b. New language features for processes
 - c. Monitors
 - d. Synchronization -- Condition Variables
 - e. More general forms of monitors
 - f. Signals

Concurrency

The Mesa architecture is designed for applications that expect a large amount of concurrent activity. Support for concurrent execution of multiple processes is provided in the Mesa Language. Mesa's goal is to reduce the complexities associated with concurrent programming.

Concurrency

In order to support the notion of dynamic processes, the Mesa Language must provide the ability to create new processes and also allow communication between the processes.

There must also be a way to control the scheduling of these processes so that they are not manipulating shared data at the same time.

To create new processes and synchronize their results, Mesa has provided to language operators:

FORK/JOIN

To control the scheduling of these processes, Mesa has provided the synchronization mechanism of:

Monitors

Processes

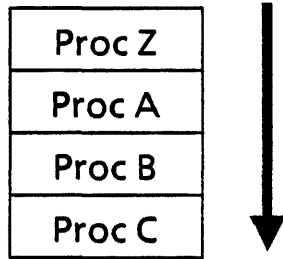
In Mesa, the creation of a new process is simply a special procedure activation that executes concurrently with its caller.

Mesa allows any procedure (except an internal procedure -- to be discussed shortly) to be invoked in this way, at the caller's discretion.

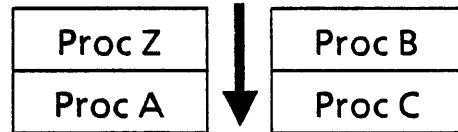
It is possible to later retrieve the results returned by this procedure.

Call Stack

Synchronous call



Asynchronous call
From A to B



New Language Operations and Data Types

There are two new language operators and one new data type for supporting concurrent execution of multiple processes.

PROCESS-- the data type of the value returned by invoking **FORK** and used to identify the process in future operations, including **JOIN**.

FORK -- the operator that is used to create a new process.

JOIN -- the operator that is used to synchronize the return of a process.

The Process Interface

Pilot also implements some routines that are used less frequently. These routines are defined in the Process interface (in the PPM).

Process.Detach: PROCEDURE[PROCESS];

-- the operator that is used to let a new process run by itself (that is, it will never be JOINed)

Process.Abort: PROCEDURE[process: UNSPECIFIED];

-- notifies a specific process that it should abort

Process.Pause: PROCEDURE[ticks: Process.Ticks];

-- puts a process to sleep for a specified amount of time

Process.Yield: PROCEDURE;

-- the calling process gives up control of the processor

Process.SetTimeout: PROCEDURE[

condition: LONG POINTER TO CONDITION, ticks: Process.Ticks];

-- sets the time limit a process will wait on a condition variable

Process.Ticks is a tick of the process timer clock. There are procedures to convert from milliseconds $\leq \geq$ ticks and seconds $\leq \geq$ ticks. Consult the PPM.

Process Representation

In the Mesa architecture, a process corresponds to either

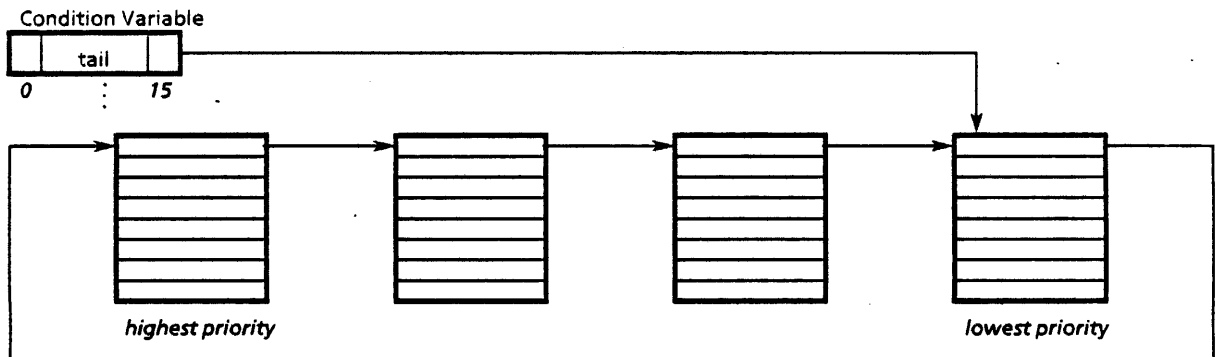
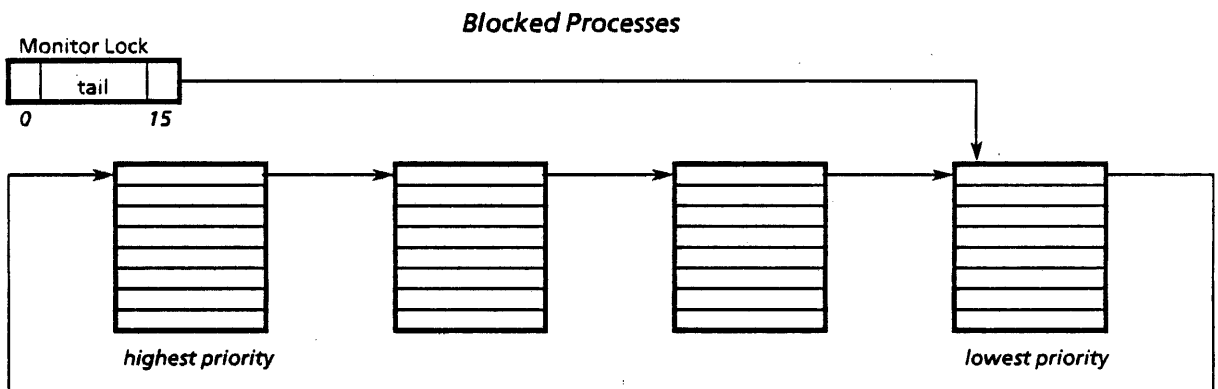
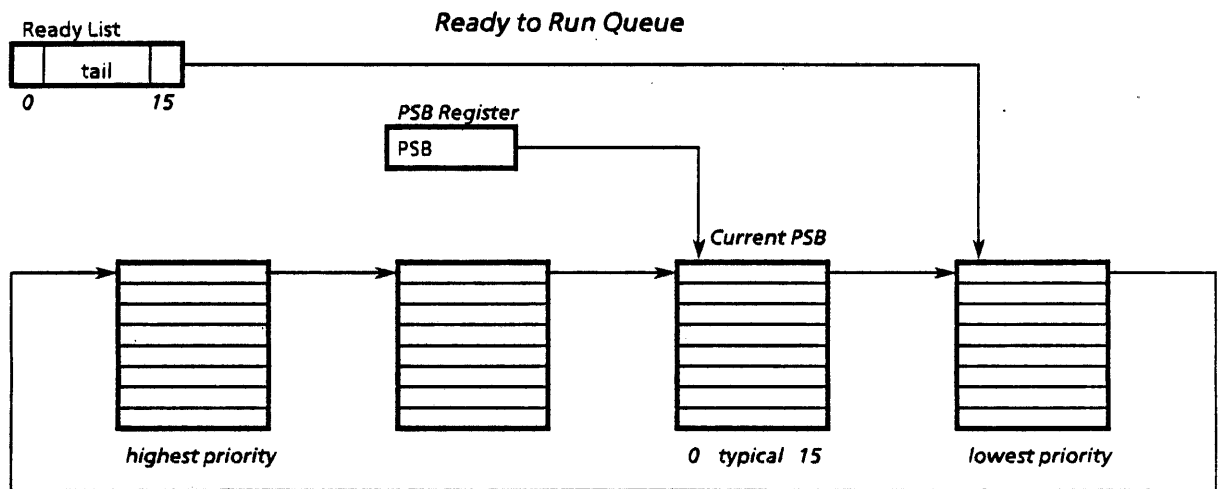
- 1) an instance of execution of a PROGRAM module or
- 2) a procedure call that runs concurrently with its caller.

Every process is represented as a Process State Block (PSB) residing on some operating system queue. A PSB contains information that allows the system to find the MDS in which the process is executing, its context (GF, LF, CB, and PC), its priority, and other information.

Operating system queues are maintained in sorted order.

Each non-running process will either be ready to run or blocked.

Process Representation (con't)



Example: Synchronous Call

The unrelated computation will have to wait for the return of ReadLine even though it does not need the results returned from that operation.

```
GetInput: PROCEDURE[buffer: LONG POINTER TO Buffer]
  RETURNS[bytesRead: CARDINAL] = {
    ...
    bytesRead ← ReadLine[buffer];
    ...
    << unrelated computation >>
    ...
  };
```

```
ReadLine: PROCEDURE[buffervar: LONG POINTER TO Buffer]
  RETURNS[numberOfBytes: CARDINAL] = { ...};
```

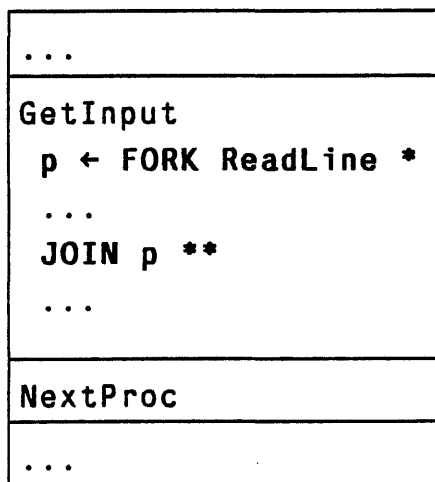
Example: Asynchronous Call

The new process is synchronized at termination.

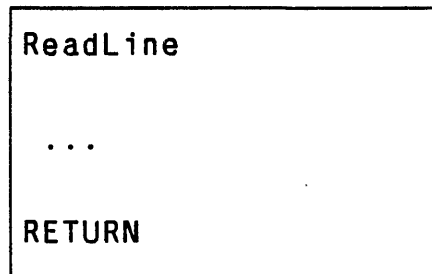
```
GetInput: PROCEDURE[buffer: LONG POINTER TO Buffer]
  RETURNS[bytesRead: CARDINAL] = {
  p: PROCESS RETURNS [CARDINAL];
  ...
  p ← FORK ReadLine[buffer];
  ...
  << concurrent computation >>
  ...
  bytesRead ← JOIN p;
  };
```

PROGRAM Module Execution.

Stack for Process N



Stack for Process N + 1



- * The FORK creates a new PSB for ReadLine and places it on the ready queue.
- ** When the JOIN is reached and ReadLine RETURNS, the results are retrieved and the call stack for Process N + 1 is deleted.

Example: Asynchronous Call

If the new process is intended to function independently and the never be JOINed then that process should be detached.

Example:

```
LookForMail: PROCEDURE[mailBox: MailBox] = {  
  ...  
  Process.Detach[FORK RealLookForMail[mailBox]];  
  ...  
};
```

RealLookForMail is a procedure that has been declared elsewhere. The actual searching takes place in this procedure.

Summary

Mesa's goal is to reduce the complexities associated with concurrent programming by providing high-level language support.

The method for passing parameters to a new process and retrieving its results is exactly the same as the corresponding method for procedures.

No special declaration is needed for a procedure that is FORKed as a process.

The cost of creating and destroying a process is moderate, and the cost in storage is only twice the minimum cost of a procedure instance.

The cost of a context switch is roughly twice the cost of a procedure call.

Therefore, ...

You are *encouraged* to build systems that use many processes with a high rate of interaction.

Monitors -- Overview

When several processes interact by sharing data, care must be taken to properly synchronize access to the data.

The belief is that processes, in general, do not interact. When they do it is in small segments of code that manipulate shared data values.

Safe communication implies only one process is operating in these critical sections at a time.

Processes not only need to ensure mutual exclusion to data, but may also wish to enforce scheduling decisions.

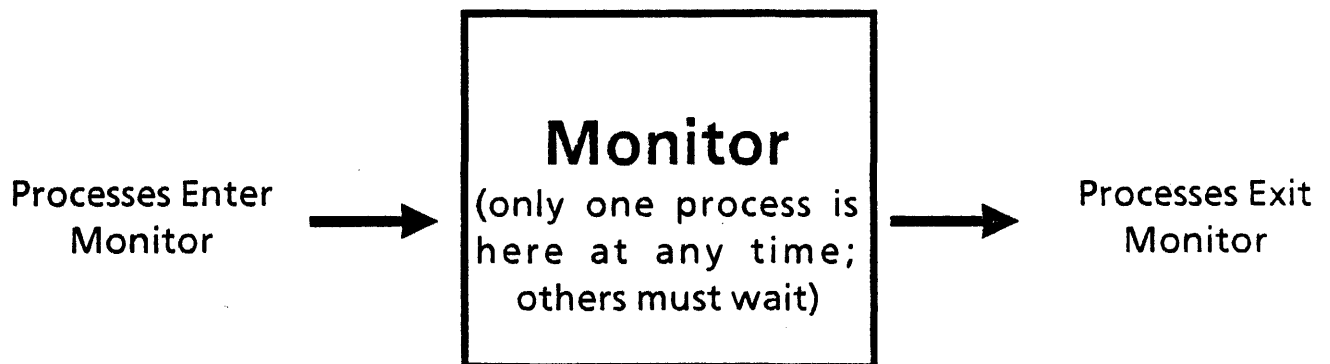
The idea behind monitors is that a proper vehicle for this interaction is one that unifies:

- 1) the synchronization
- 2) the shared data
- 3) the body of code that performs the accesses
- 4) the scheduling needs

Monitors -- Overview

For consistency, Mesa groups these components in a MONITOR module.

Access to shared data is controlled by a MONITOR lock. One lock is associated with the monitor. When the monitor's code is entered the lock must be acquired. When a process leaves the monitor, the lock is released.



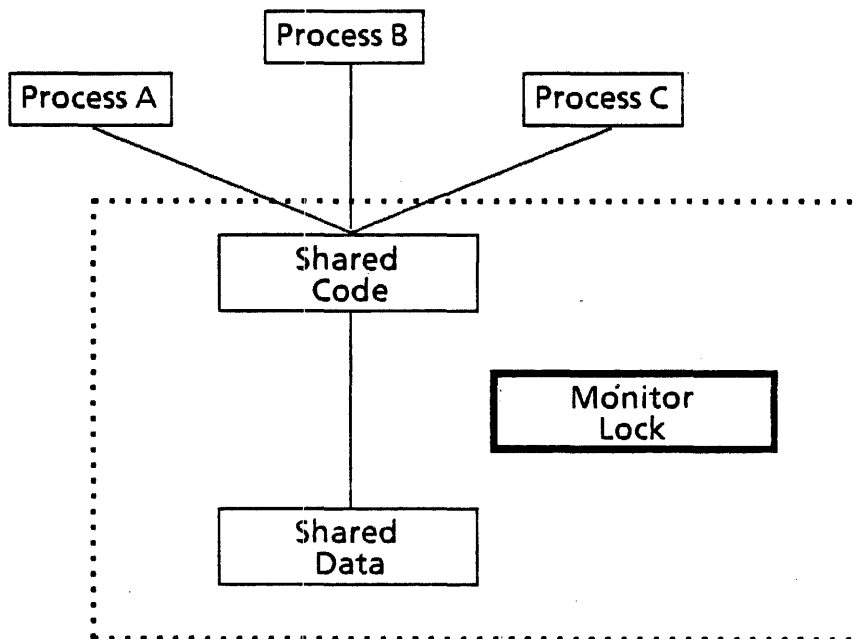
Access to the module are controlled by three types of procedures:

- external procedures** -- procedures that reside in a MONITOR module but execute outside of the monitor.
- ENTRY procedures** -- entry points into the monitor.
- INTERNAL procedures** -- shared monitor routines that are only called from other ENTRY or INTERNAL procedures

Monitoring Global Data

Most common form of a monitor is to package all the data and procedures of the monitor within a single monitor instance.

The monitor lock, in this case, is declared automatically in the global frame of the module. Thus, all of the global variables are monitored.



Monitor Module Structure

Monitor modules are declared much like program modules, except with the keyword **MONITOR** instead of **PROGRAM**.

Example:

```
DIRECTORY
  ...;

SampleMonitor: MONITOR IMPORTS ... EXPORTS ... = {

  <<declaration of shared variables>>
  ...
  <<code>>
}.

```

External procedures are declared as normal non-monitor procedures:

```
P: <PUBLIC> PROCEDURE[args] RETURNS[results] = ...
```

Every monitor has *at least one* entry procedure; these are declared as:

```
Q: <PUBLIC> ENTRY PROCEDURE[args] RETURNS[results] = ...
```

Internal procedures are declared as:

```
R: INTERNAL PROCEDURE[args] RETURNS[results] = ...
```

Note that external procedures should not reference global data as only monitored procedures should reference the global data (external procedures are logically outside the monitor).

Interfaces to Monitors

In Mesa, the attributes ENTRY and INTERNAL are associated with a procedure's body, not with its type. Therefore, from the client side of an interface, a monitor appears to be a normal program module.

For example, a monitor M with entry procedures P and Q might have an interface that appears as:

```
MDefs: DEFINITIONS = {  
  P: PROCEDURE[args] RETURNS[results];  
  Q: PROCEDURE[args] RETURNS[results];  
};
```

Mutual Exclusion

Mutual exclusion in a monitor module is ensured by the monitor lock:

The lock can only be held by one process at a time.

The lock is implicitly acquired upon entry to an ENTRY procedure.

The lock is implicitly released when the ENTRY procedure returns.

Once the lock is held, other processes attempting to enter the monitor are blocked.

Once the lock is released, one of the processes waiting for the lock or a new process that is attempting to enter the monitor can acquire the lock.

Mutual Exclusion Example

Mutual exclusion to a structure that keeps count of the number of active and inactive objects in an environment:

```
MutexDefs: DEFINITIONS = {
  CounterType: TYPE = RECORD[
    active: INTEGER,
    inactive: INTEGER];

  Activate: PROCEDURE;
  Deactivate: PROCEDURE;
  Inspect: PROCEDURE RETURNS[counter: Countertype];
}.
```

Mutual Exclusion Example (con't)

DIRECTORY

MutexDefs;

```
Mutex: MONITOR EXPORTS MutexDefs = {  
  counter: MutexDefs.CounterType ← [0,0];  -- this is the monitored data
```

```
  Activate: PUBLIC ENTRY PROCEDURE = {  
    ENABLE UNWIND => NULL;  -- explained later  
    counter.active ← counter.active + 1;  
    counter.inactive ← counter.inactive - 1;  
  };
```

```
  Deactivate: PUBLIC ENTRY PROCEDURE = {  
    ENABLE UNWIND => NULL;  
    counter.active ← counter.active - 1;  
    counter.inactive ← counter.inactive + 1;  
  };
```

```
  Inspect: PUBLIC ENTRY PROCEDURE  
    RETURNS[counter: MutexDefs.CounterType] = {  
      ENABLE UNWIND => NULL;  
      RETURN[counter];  
    };
```

```
};
```

Synchronization

Synchronization among cooperating processes is expressed explicitly through operations on condition variables:

Condition variables are declared as:

c: CONDITION;

Operations on condition variables:

WAIT condition -- puts process to sleep. The process that executes this statement exits the monitor and waits.

NOTIFY condition -- wakes up *first* process that is waiting

BROADCAST condition -- wakes up *all* waiting processes

CONDITION Variables

Condition variables are always associated with some boolean expression describing the desired state of the monitor data. This yields the general pattern:

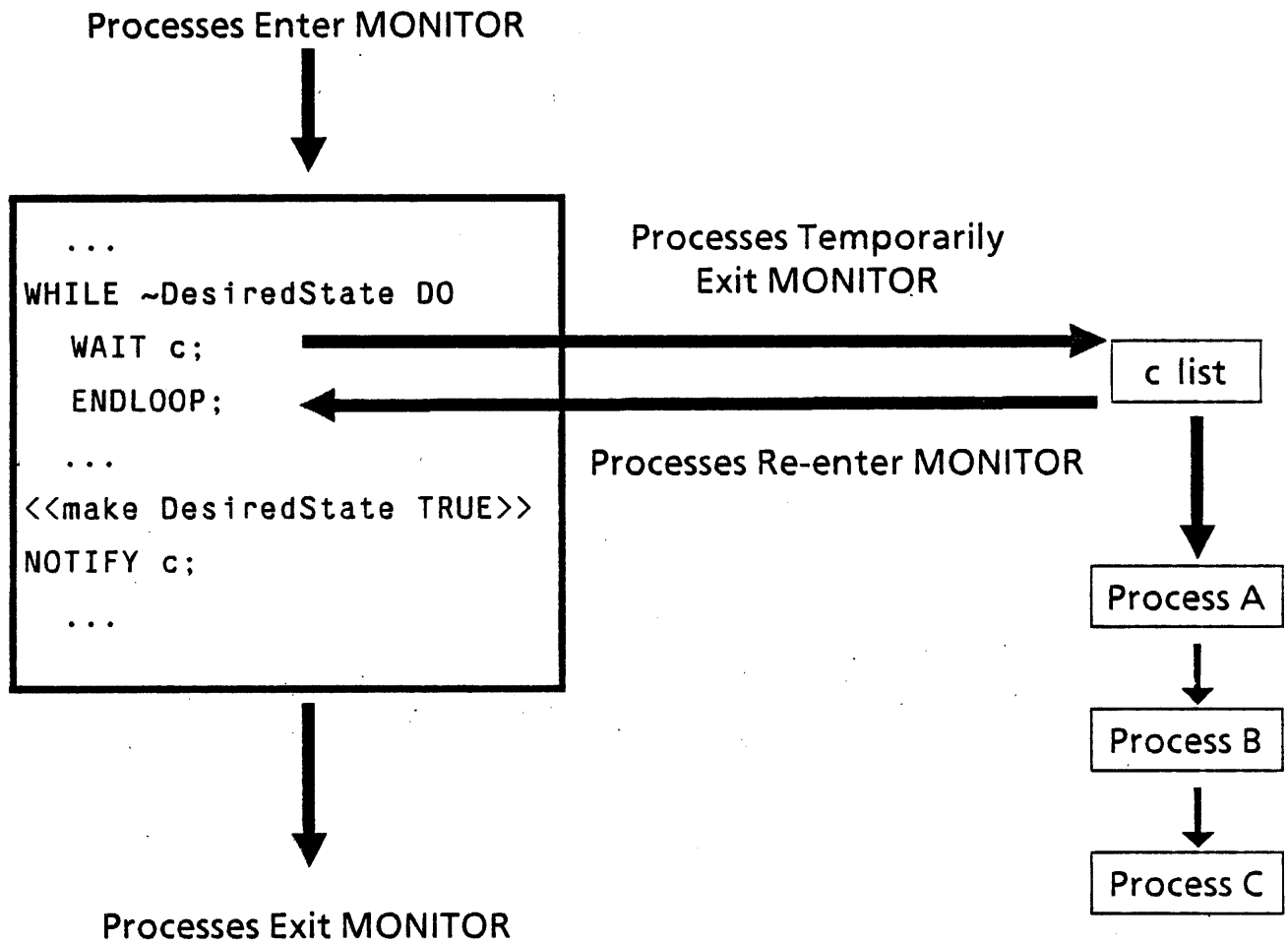
Example of a process (in the monitor) waiting for condition:

```
... -- code fragment
WHILE ~DesiredState DO
    WAIT c;    -- release lock
    ENDLOOP;  -- reacquire lock
<<execute with monitor lock held>>
...
<<RETURN>>    -- release lock
```

Example of a process (in the monitor) making condition true:

```
... -- code fragment
<<make DesiredState TRUE>> -- maybe as side effect of modifying global data
NOTIFY c;
<<continue execution>>
...
<<RETURN>>    -- release lock
```


CONDITION Variables



Note that *only one* process can be in the MONITOR at a time.

NOTIFY Example

DIRECTORY

AllocDefs;

StorageAllocator: MONITOR EXPORTS AllocDefs = {

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;

Allocate: PUBLIC ENTRY PROCEDURE RETURNS[p: ListPtr] = {

ENABLE UNWIND => NULL; *-- explained later*

WHILE freeList = NIL DO *-- boolean expression testing for desired state*

 WAIT StorageAvailable;

 ENDLOOP;

 p ← freeList;

 freeList ← p.next;

};

Free: PUBLIC ENTRY PROCEDURE[p: ListPtr] = {

ENABLE UNWIND => NULL;

p.next ← freeList;

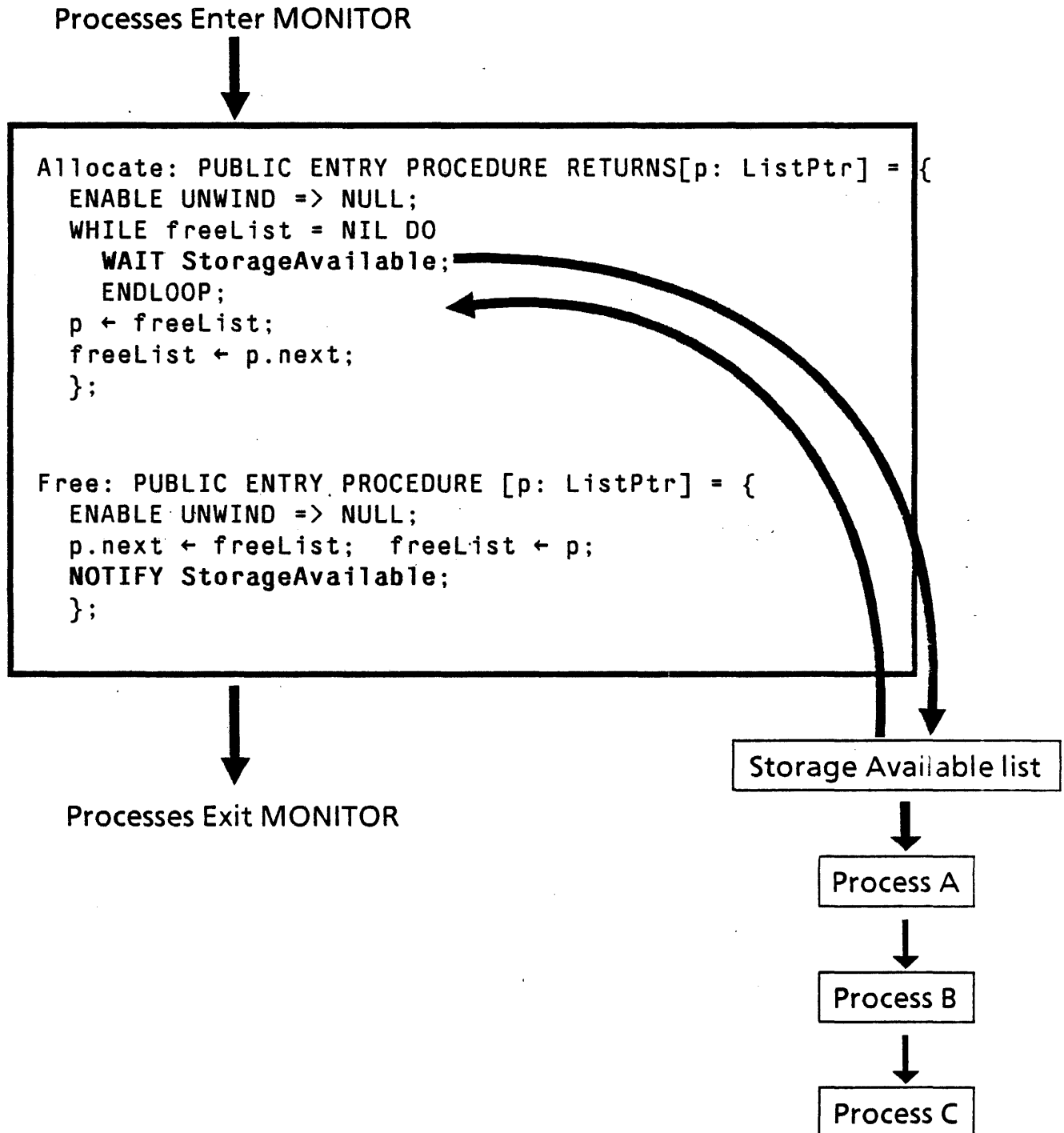
freeList ← p; *-- changes the desired state*

NOTIFY StorageAvailable;

};

};

NOTIFY Example Graphics



BROADCAST Example

DIRECTORY

```
AllocDefs;
```

```
StorageAllocator: MONITOR EXPORTS AllocDefs = {
  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;
```

```
Allocate: PUBLIC ENTRY PROCEDURE [size: CARDINAL]
  RETURNS [p: ListPtr] = {
  ENABLE UNWIND => NULL;
  UNTIL << chunk of size words is available >> DO
    WAIT StorageAvailable;
  ENDLOOP;
  p ← << remove chunk of size words >>;
};
```

```
Free: PUBLIC ENTRY PROCEDURE [p:ListPtr,size: CARDINAL] = {
  ENABLE UNWIND => NULL;
  ...
  << put back storage of size words >>
  ...
  BROADCAST StorageAvailable;
};

}.
```

Other Forms of Notification

Since notification is a hint, a process can be awakened for reasons other than a NOTIFY or BROADCAST on a CONDITION.

Timeouts -- wakes up a process after a specified period of time.
(The process must determine why it was awakened).

Abort a process -- wakes up a process and generates the error ABORTED.

Example: Timeouts

DIRECTORY

```
RemResDefs USING [Print];
```

```
RemoteResource: MONITOR IMPORTS RemResDefs EXPORTS RemResDefs = {
  ResourceAvailable: CONDITION;
  available: BOOLEAN ← TRUE;
```

```
ConnectAndWork: PUBLIC ENTRY PROCEDURE[work: Work]= {
  ENABLE {
    ABORTED => GOTO return;  --see next example
    UNWIND => NULL};
  UNTIL available DO
    Process.Detach[FORK RemResDefs.Print[<<note to user>>]];
    WAIT ResourceAvailable;
  ENDLOOP;
  available ← FALSE;
  <<do work with connection>>;
  EXITS
    return => NULL;
};
```

```
Disconnect: PUBLIC ENTRY PROCEDURE[res: Resource]= {
  ENABLE UNWIND => NULL;
  ...
  available ← TRUE;
  NOTIFY ResourceAvailable;
};
```

--mainline code

```
Process.SetTimeout[@ResourceAvailable, RemResDefs.oneMin];
Process.EnableAborts[@ResourceAvailable];

}.
```

Example: Abort

DIRECTORY

RemResDefs;

SomeImpl: PROGRAM IMPORTS RemResDefs = {
 p: PROCESS;

MakeConnection: PROCEDURE[work: Work]= {
 p ← FORK RemResDefs.ConnectAndWork[work];
};

Print: PUBLIC PROCEDURE[message: LONG STRING]= {
 <<print the message>>
 IF <<user does not want to continue>> THEN
 Process.Abort[p]; -- raises error ABORTED[p]
};

};

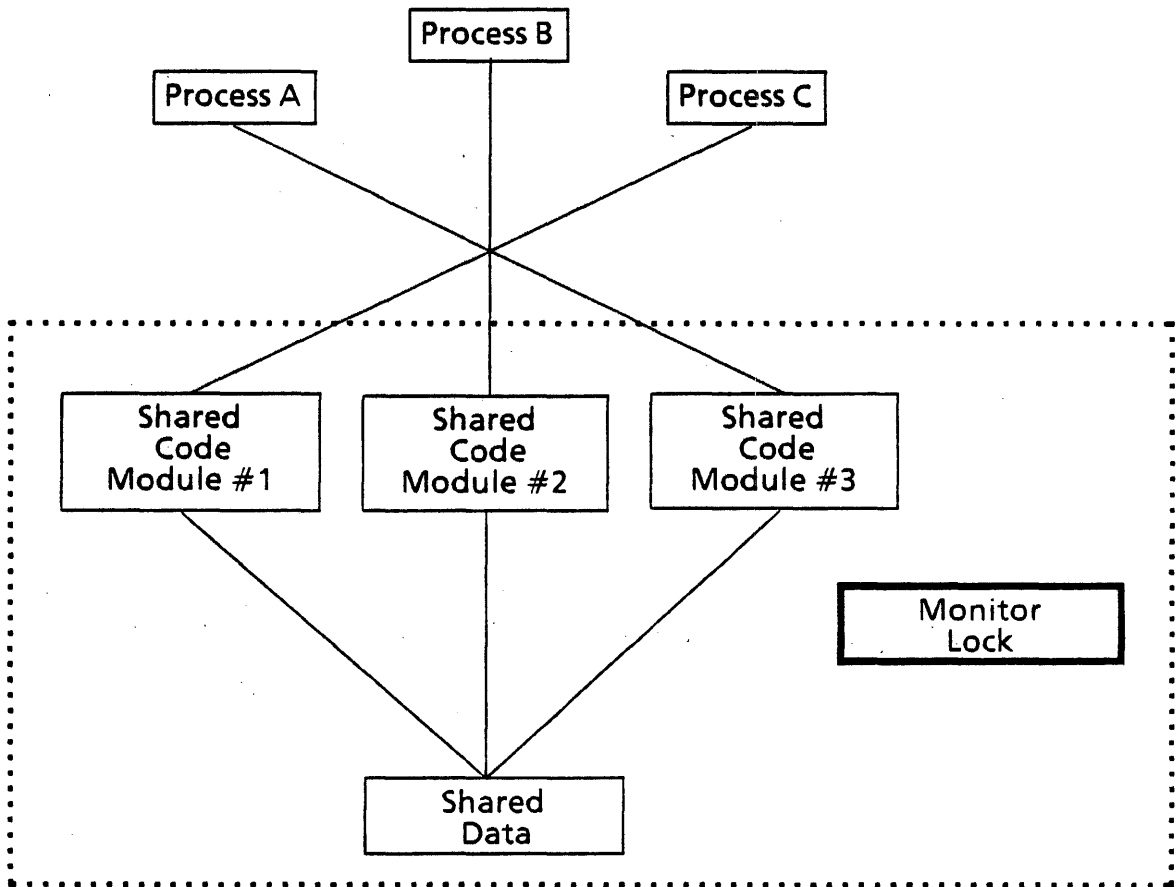
More General Forms of Monitors

We've covered the basic form of a monitor, however, there are situations when one wants to provide a different form of monitoring.

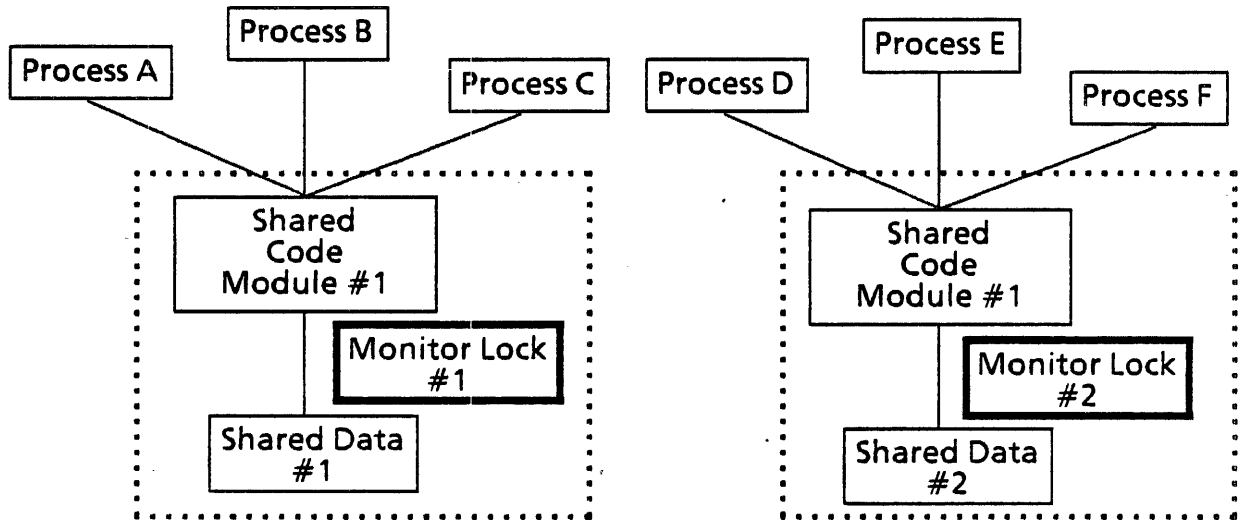
There may be times when the monitor code may be too large to fit inside one programming module, so it needs to be broken into several modules. This is called *Multi-Module Monitors*. The idea is that you have many programming modules but only one Logical Monitor and therefore, need only one Monitor lock. This requires an explicit declaration of the monitor lock.

There also may be a need to not monitor the code but rather just the shared data structures. This is called *Object Monitors* and is implemented with monitored records.

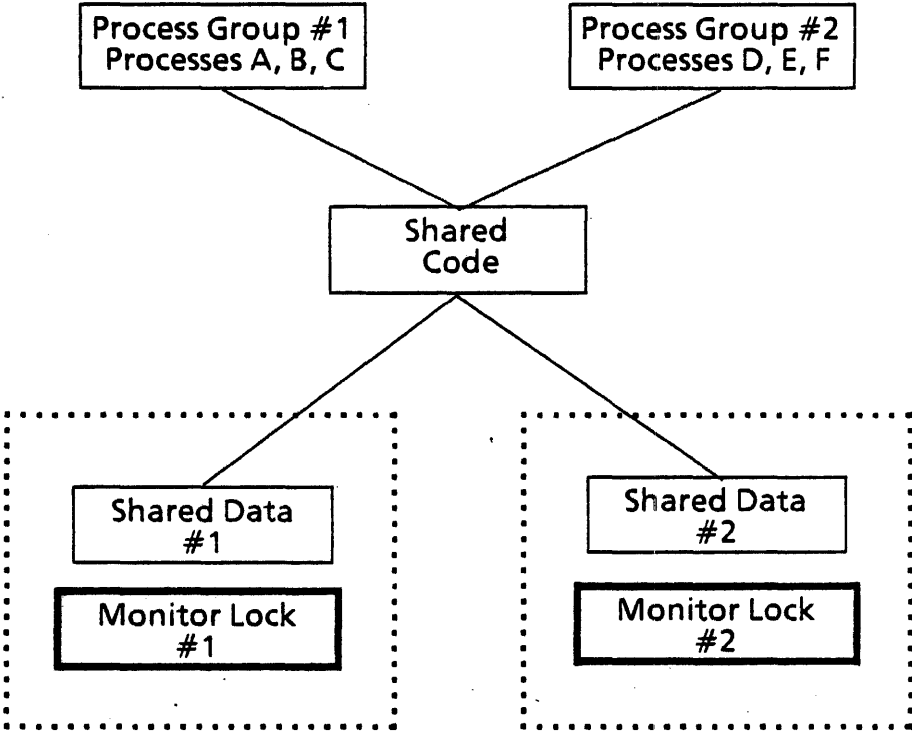
Multi-Module Monitor



Object Monitor



Object Monitor



Explicit Declaration of Monitor Locks

To have the compiler use a programmer declared lock instead of inserting a anonymous global lock, you must include a LocksClause. The LocksClause goes just after the naming of a MONITOR module.

General form of a LocksClause:

```
LocksClause := empty |
              LOCKS Expression |
              LOCKS Expression USING
              identifier: TypeSpecification
```

A monitor lock must also be explicitly declared in the global frame of the Monitor:

```
myLock: MONITORLOCK;
```

And since the lock is going to be shared among several modules, it will need to be declared in a DEFINITIONS module, too.

Example of the Monitor Module:

```
--SampleMonitor.mesa
DIRECTORY
  ...;

SampleMonitor: MONITOR LOCKS myLock = {
  myLock: MONITORLOCK;
  ...
  Proc1: ENTRY PROC[...] = {...};
  ...
}.
```

Multi-Module Monitor Example

-- Definitions Module

```
MultiModuleInternalDefs: DEFINITIONS = {  
  sharedLock: MONITORLOCK;  
  P, Q, R, S: PROCEDURE[...];  
  <<other necessary definitions>>  
}.
```

-- Monitor Module #1

```
DIRECTORY  
  MultiModuleInternalDefs;  
MultiModuleImplA: MONITOR LOCKS sharedLock  
  EXPORTS MultiModuleInternalDefs = {  
  
  sharedLock: PUBLIC MONITORLOCK;  
  ...  
  P: PUBLIC ENTRY PROCEDURE[...] = {...};  
  ...  
}.
```

-- Monitor Module #2

```
DIRECTORY  
  MultiModuleInternalDefs;  
MultiModuleImplB: MONITOR LOCKS MultiModuleInternalDefs.sharedLock  
  IMPORTS MultiModuleInternalDefs  
  EXPORTS MultiModuleInternalDefs = {  
  ...  
  Q: PUBLIC ENTRY PROCEDURE[...] = {...};  
  ...  
}.
```

Multi-Module Monitor Example (con't)

-- Monitor Module #3

DIRECTORY

MultiModuleInternalDefs;

MultiModuleImp1C: MONITOR LOCKS MultiModuleInternalDefs.sharedLock

IMPORTS MultiModuleInternalDefs

EXPORTS MultiModuleInternalDefs = {

...

R: PUBLIC ENTRY PROCEDURE[...] = {...};

...

}.

-- Monitor Module #4

DIRECTORY

MultiModuleInternalDefs;

MultiModuleImp1D: MONITOR LOCKS MultiModuleInternalDefs.sharedLock

IMPORTS MultiModuleInternalDefs

EXPORTS MultiModuleInternalDefs = {

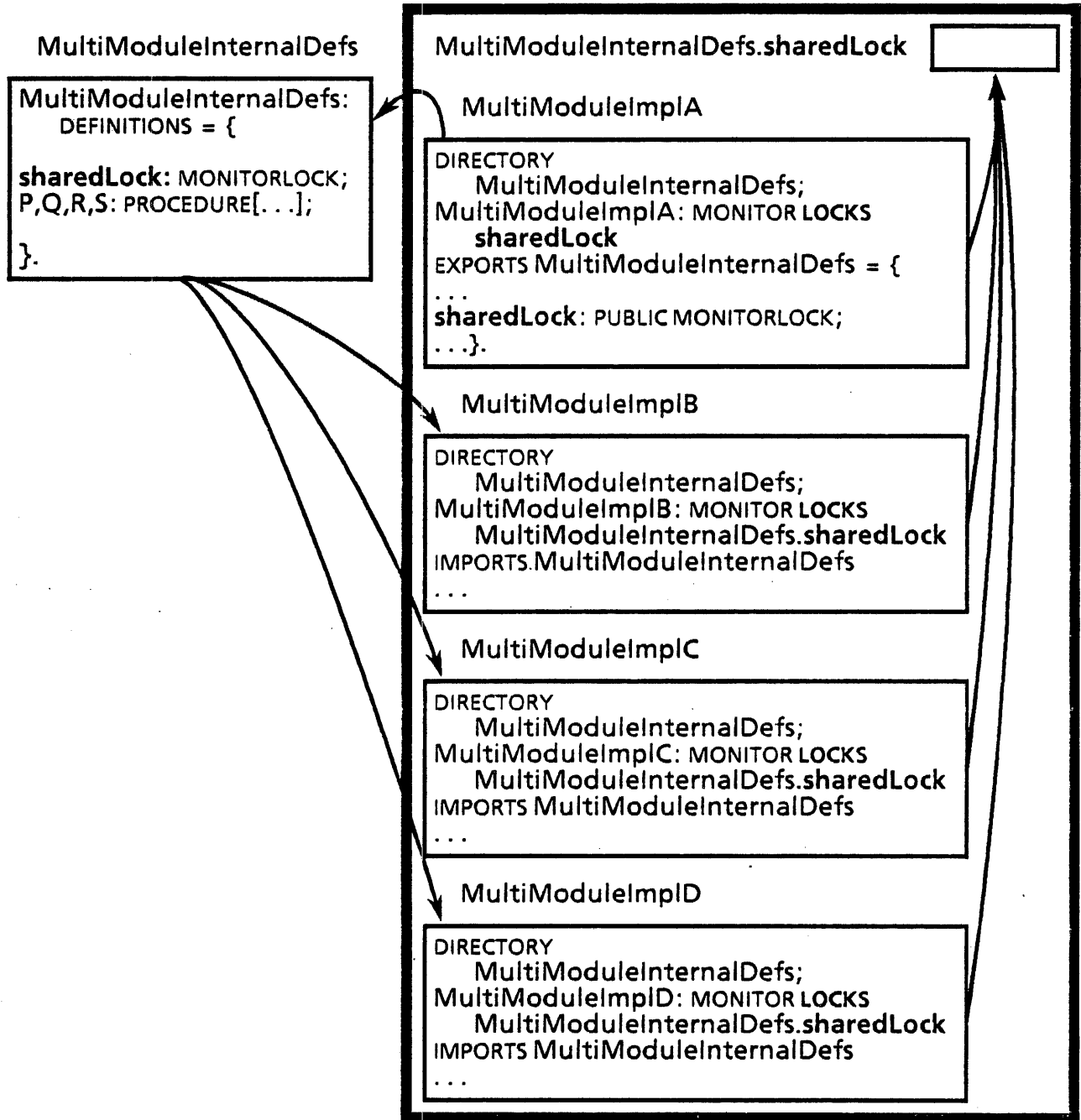
...

S: PUBLIC ENTRY PROCEDURE[...] = {...};

...

}.

Multi-Module Monitor Example (con't)



Note: All Modules use the same lock.

Monitored Records

For situations in which the monitor data cannot simply be the global variables of the monitor module, a monitored record can be used. They are declared as normal records with the key word MONITORED describing it:

```
r: MONITORED RECORD[x: INTEGER, ...];
```

Characteristics of Monitored Records:

A Monitor Lock is automatically inserted in the record.

Fields of the monitored record must not be accessed except from within a monitor which first acquires a lock.

If a monitored record is passed as an argument to a procedure, it must only be done by reference.

Object Monitor Example

```

StackDefs: DEFINITIONS = {

    Handle: TYPE = LONG POINTER TO Object;
    Object: TYPE;  -- An Opaque Type! This will be a monitored record

    Pop: PROCEDURE[stack: Handle] RETURNS[value: LONG CARDINAL];
    Push: PROCEDURE[stack: Handle, value: LONG CARDINAL];
    IsEmpty: PROCEDURE[stack: Handle] RETURNS[itIs: BOOLEAN];
    Create: PROCEDURE[size: LONG CARDINAL] RETURNS[stack: Handle];
    Destroy: PROCEDURE[stack: Handle];

}.  -- End of StackDefs

```

DIRECTORY

```

    Heap,
    StackDefs;

```

```

ObjectImpl: MONITOR LOCKS stack USING stack: StackDefs.Handle
EXPORTS StackDefs = {

```

```

    myZone: UNCOUNTED ZONE ← Heap.Create[initial: 5];

```

```

    Object: PUBLIC TYPE = MONITORED RECORD[
        << some representation of a stack >> ];

```

Object Monitor Example (con't)

```

Pop: PUBLIC ENTRY PROCEDURE[stack: StackDefs.Handle]
  RETURNS[value: LONG CARDINAL] = {
    ENABLE UNWIND => NULL;
    << can access the record's fields in here >> };

Push: PUBLIC ENTRY PROCEDURE[stack: StackDefs.Handle,
  value: LONG CARDINAL] = {
    ENABLE UNWIND => NULL;
    << can access the record's fields in here >> };

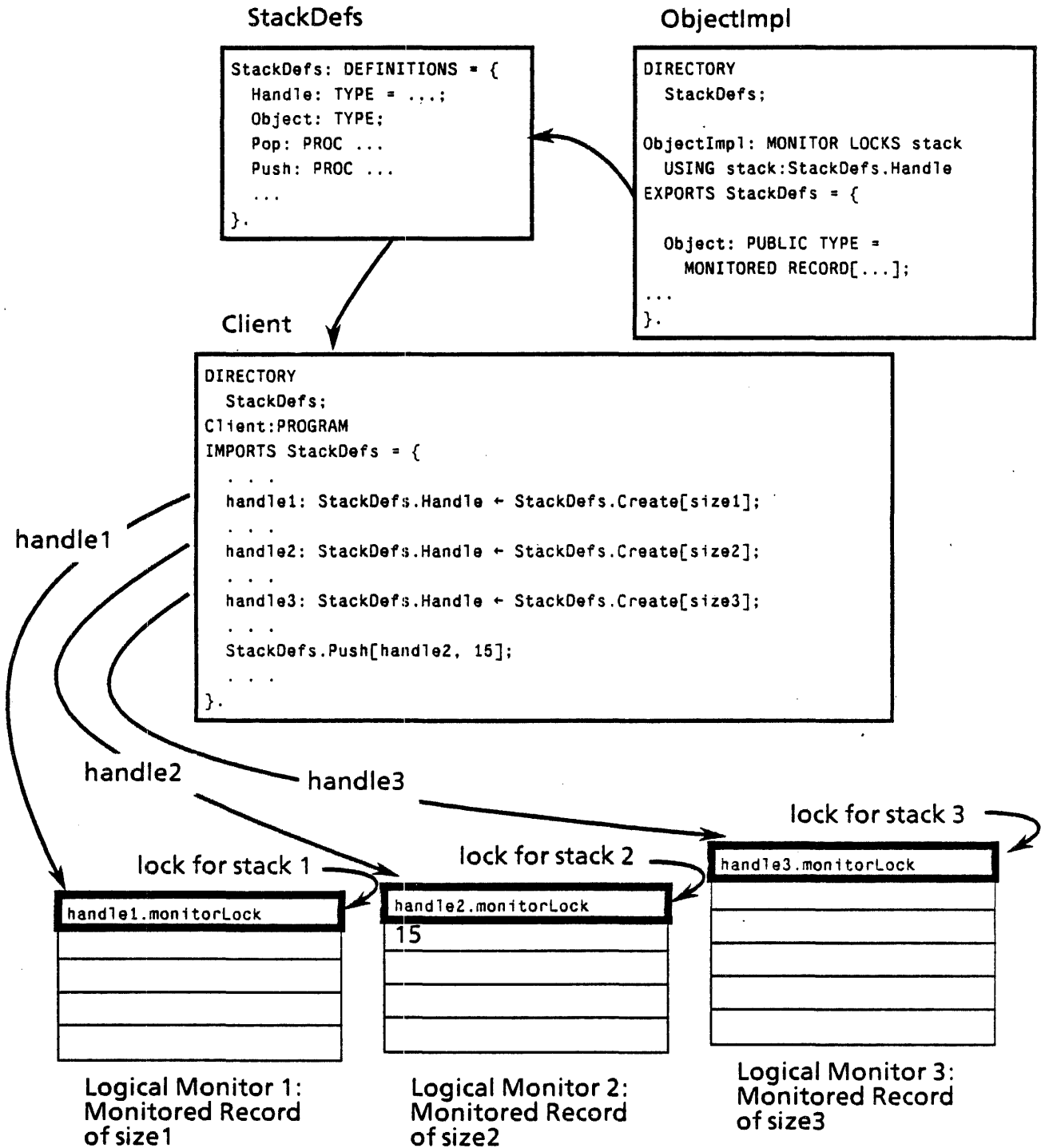
IsEmpty: PUBLIC ENTRY PROCEDURE[stack: StackDefs.Handle]
  RETURNS[itIs: BOOLEAN] = {
    ENABLE UNWIND => NULL;
    << can access the record's fields in here >> };

Create: PUBLIC PROCEDURE[size: LONG CARDINAL]
  RETURNS[stack: StackDefs.Handle] = {
    << allocate the stack object and initialize here >>
    stack ← myZone.NEW[Object[]];
  };

Destroy: PUBLIC ENTRY PROCEDURE[stack: StackDefs.Handle] = {
  ENABLE UNWIND => NULL;
  << free the stack object here>>
  myZone.FREE[@stack];
};
...
}.

```

Object Monitor Example (con't)



General Issues

Signals and Processes

Signals and Monitors

Deadlock conditions

Signals and Processes

Each process has its own call stack.

FORKing a process, therefore creates a new call stack.

Recall that signals propagate up a call stack.

This implies that signals cannot propagate across the gap created by FORKing a procedure.

The only suitable targets for a FORK, therefore, are procedures that catch any signals they incur, and that never generate signals of their own that are expected to be handled by another process.

Signals and Monitors

Signals within the body of an entry procedure require special attention:

Monitor locks are not released when entry frames are unwound.

Raising a signal from within an entry procedure does not release the monitor lock.

Releasing the Lock Using UNWIND

To ensure the monitor lock is released when an entry procedure is being unwound, include an UNWIND catch in the outermost block of the procedure body.

Example:

```
Proc: ENTRY PROCEDURE[...] = {  
  ENABLE UNWIND => { << restore invariant >> };  
  ...  
};
```

Or:

```
Proc: ENTRY PROCEDURE[...] = {  
  ENABLE UNWIND => NULL;  -- Even if you don't need to do anything special,  
  ...                    -- you must include this to release the lock  
};
```

RETURN WITH ERROR

When raising a signal from within an entry procedure you can use the RETURN WITH ERROR construct if you want to release the monitor lock.

```
Failure: ERROR[kind: CARDINAL] = CODE;
```

```
Proc: PUBLIC ENTRY PROCEDURE[...] RETURNS[c1, c2: CHAR] = {  
  ENABLE UNWIND => NULL;  
  ...  
  IF cond1 THEN ERROR Failure[1];  
  IF cond2 THEN RETURN WITH ERROR Failure[2];  
  ...  
};
```


Deadlocks

Three common cases of pairwise deadlock:

Two processes in a monitor WAITing, expecting the other to wake it up.

Cyclic calls between two monitors.

Embedded levels of monitors

INDEX

A	
Abort	5-8, 5-30, 5-32
Asynchronous	5-6, 5-12, 5-14
B	
BROADCAST	5-24, 5-29 - 5-30
C	
Concurrency	5-1 - 5-4
CONDITION	5-8, 5-24 - 5-26, 5-27, 5-29 - 5-31
D	
Deadlock	5-45
Detach	5-8, 5-14, 5-31
E	
ENTRY	5-17, 5-19 - 5-21, 5-48 - 5-49
ERROR	5-49
Example	5-14, 5-22 - 5-23, 5-25, 5-27 - 5-29, 5-31 - 5-32, 5-38 - 5-40, 5-42 - 5-44
Exclusion	5-21 - 5-23
External	5-19
F	
FORK	5-4, 5-7, 5-12 - 5-14, 5-31 - 5-32, 5-46
G	
Global	5-18
I	
Interfaces	5-20
INTERNAL	5-17, 5-19 - 5-20
J	
JOIN	5-4, 5-7, 5-12 - 5-13
L	
Lock	5-10, 5-18, 5-34 - 5-36, 5-41, 5-48
M	
Monitors	5-2, 5-4, 5-16 - 5-17, 5-20, 5-33, 5-45, 5-47
Multi-Module	5-33 - 5-34, 5-38 - 5-40

N
Notification 5-30
NOTIFY 5-24 - 5-26, 5-27 - 5-28, 5-30 - 5-31

O
Object 5-33, 5-35 - 5-36, 5-42 - 5-44

P
Pause 5-8
Process 5-7 - 5-10, 5-12 - 5-14, 5-18, 5-26, 5-28, 5-31 - 5-32, 5-34 - 5-36
Processes 5-1 - 5-2, 5-5, 5-10, 5-16 - 5-17, 5-26, 5-28, 5-36, 5-45 - 5-46

R
Records 5-41
Representation 5-9 - 5-10

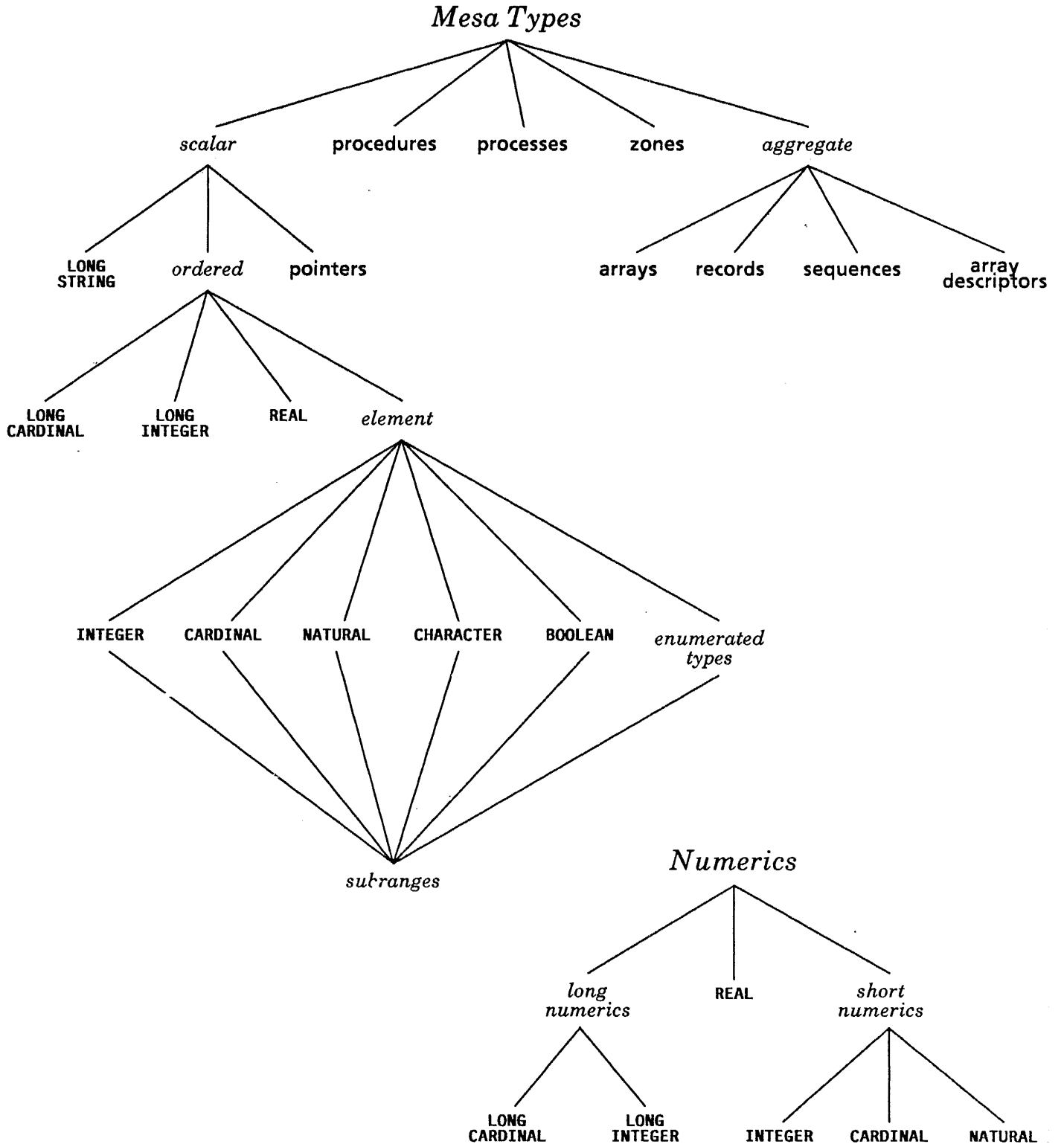
S
SetTimeout 5-8, 5-31
Signals 5-2, 5-45 - 5-47
Stack 5-6, 5-13
Structure 5-19
Synchronization 5-2, 5-24
Synchronous 5-6, 5-11

T
Ticks 5-8
Timeouts 5-30 - 5-31

U
UNWIND 5-23, 5-27 - 5-29, 5-31, 5-43, 5-48 - 5-49

W
WAIT 5-24 - 5-26, 5-27 - 5-29, 5-31

Y
Yield 5-8



Type Determination of Numeric Expressions

Word Length Rules for Expressions (Balancing)

In determining what type of operation (INTEGER, CARDINAL, LONG INTEGER, LONG CARDINAL, REAL) should be performed in an expression, a common word length must first be found.

In general, the operation requiring the fewest automatic type conversions will be the one used. So, for numerics:

1. If all (both) operands are short numerics, a short numeric operation will be used.
2. If all (both) operands are long numerics, a long numeric operation will be used.
3. If one operand is a long numeric, the other operand will be lengthened and a long operation will be used.
 - a. When an INTEGER is lengthened, its inherent type is LONG INTEGER.
 - b. When a CARDINAL or NATURAL is lengthened, its inherent type is LONG INTEGER and LONG CARDINAL.
4. If one operand is a REAL the other operand is converted and a REAL operation is used.

Determination of Representation (Balancing)

Once a common word length is found for an expression, the following rules will choose whether a signed (INTEGER, LONG INTEGER) or unsigned (CARDINAL, LONG CARDINAL) operation will take place.

1. If the operands have exactly one common inherent representation, the operation defined for that representation is selected (and the target representation is ignored).
2. If the operands have no common inherent representation but the target representation is well-defined, the operation yielding that representation is chosen.
3. If the operands have both inherent representations in common, and if target representation is well-defined, it selects the operation.
4. If the operands have both inherent representations in common but the target representation is ill-defined, the signed operation is chosen.
5. Unary minus converts its argument to a signed representation if necessary and produces a signed result.

If the operands have no representation in common and the target representation is ill-defined, the expression is in error.

About The Lab

The Training Lab is located in room C401, next door. There are 20 machines available, so there should be one for everyone. The machines are already set up to contain all of the software you will need for the week. The XDE Mail Tutorials have also been installed in case you have not completed all of them or you would like to refer back to them. Here are a few other things to keep in mind:

- 1) The afternoon labs are "Free Form". That is, you may come and go as you please, taking breaks and lunch as you wish. We do expect, however, that you do put some effort into working on the day's assignment.
- 2) In order for the instructors to understand how well the information is getting through, we would like each of you to show one of the instructors a running version of your assignment. (We may choose to test a few key things to see how robust your implementation actually is.)
- 3) There is an information card on each keyboard assigning you a logon name and password as well as other information about your machine. You do not need to be logged in to work in XDE, although you do need to be logged in to perform any operations over the net (e.g. Printing something). If you have your own logon name on the Xerox net, you may use it if you like.
- 4) The lab machines have an established SearchPath of directories. These include a working directory (MesaWD), a Mesa Interface directory (MesaDefs), etc. To avoid problems, you should not alter this SearchPath.
- 5) XDE Documentation is located in the back of the room. There are copies of the Mesa Language Manual, Mesa Programmer's Manual, Pilot Programmer's Manual, XDE User's Guide, and HardCopy versions of the XDE Tutorials. You are encouraged to use these throughout the course, although they are to remain here for future students and classes.
- 6) You may wish to personalize the *User.cm* on your workstation by changing the *HardCopy PrintedBy:* option, or the default *Brush*, the *Logon Name*, etc. It is okay to do so, but you do not have to.
- 7) At the end of each day, send a mail message to us mentioning what you liked or disliked about the day's work, any typos you noticed, bugs in the programs (ours not yours), or suggestions you may have. (We are constantly modifying the material based on students' suggestions.) The message may be as brief or lengthy as you like. We have found, though, that students who wait until Day 5 to summarize their thoughts in one message, tend to forget some of the thoughts that they had earlier in the week.
- 8) Before you leave for an extended period of time (especially overnight), be sure to run some sort of DMT on the screen in order to protect them. There are many to choose from (e.g. DMT, BrushDMT, Poly, SpaceOut, KineticFractal, etc.).
- 9) Most importantly: **ASK QUESTIONS!** We are here to help you.

Our electronic addresses: MacKay:OSBU North:Xerox, Herz:OSBU North:Xerox

Compiler Exercise

There are two parts to this assignment to help you learn mesa syntax. In Part I you will fix errors in code already written for you. In Part II you will write some code yourself.

Part I: Syntax Errors

The file `CompilerPractice.mesa` has several intentional compiler errors. Use Command Central (or the executive) to compile `CompilerPractice.mesa`. You can see the results in the Compiler log, which is displayed in the bottom subwindow of Command Central. (If you used the executive to compile, then you will have to load the file `Compiler.log` into a file window.)

When the compiler finds an error, it gives an indication of the error (an error message), the position of the error in the source file, a listing of the offending line, and the fix the Compiler assumed in order to continue (when possible).

The compiler will not go on to the next pass after it detects an error. This means that you must fix the errors and compile the program again. If it finds more errors, then you must fix them and repeat the cycle.

Upon successfully compiling the program, run the program in the Executive by typing '`CompilerPractice`'.

Part II: Simple Programming

If you ran `CompilerPractice`, you will have noticed that the tool is a simple Math tool. You might have noticed that there are five functions (add, subtract, multiply, divide, y^x). The first four functions have already been implemented. You will implement the function y^x .

In the program, each function is implemented in separate procedures. The procedure `YtotheXInternal`, which is where the work should be done, has already been started for you. The code for printing the answer has been provided. You need to calculate what Y to the X power is and store the answer in the variable `answer`. The values of x and y are stored in `data.x` and `data.y`, respectively. (Don't worry about all cases; you need only be concerned with relatively small numbers for x and y. This is just to give you some experience in writing very simple mesa code.)

To test your program, you need to unload the previous version by typing '`Unload CompilerPractice`' in the executive. Then, recompile and then run your new version (like you did in Part I).

Debugger Examples

Learning Some Debugger Commands with MiscProcs

These examples are to give you a little experience with debugger before working on today's debugger assignments.

This is the same **MiscProcs** example that is used in the on-line debugger tutorial. If you are not completely comfortable with the concepts covered in the on-line tutorial, or if you haven't done the debugger tutorial, you should go through this example. Otherwise, feel free to go on to the next example.

The purpose of this example is to introduce you to some of the more useful aspects of the debugger's interpreter. At this point, you should make sure that you have the file **MiscProcs.mesa** on your Copilot volume. Load the file into an empty window. Compile the module either from the Executive (by typing '**Compiler MiscProcs**') or by using the Command Central tool. If it does not compile successfully, stop and ask a lab assistant for help. Otherwise, run the program by typing '**MiscProcs/d**' in the Executive.

This command *loads* the file **MiscProcs.bcd**, but does not *start* it. What this means is that the global frame has been allocated for the program (and certain information has been recorded in the operating system) but none of the mainline code has been executed and none of the variables have been initialized. The switch **/d**, calls the debugger. If there is not a sword window to debug locally (or if it is tiny), then one will be made active for you; otherwise, search the active windows looking for the local debugging window. At this point, you can refer to the module from the debugger (since its global frame has been allocated).

In this example, you should type whatever is underlined into the debugger file subwindow. Let's begin with:

1. >SEt Module context: MiscProcs -- type a carriage return (CR) at the end of the line
2. > A;j -- type a space, then the characters "A;j" followed by a CR
3. > Factorial[5] -- type a space at the beginning of lines 3 thru 9
4. > 170B?
5. > A;j
6. > A[3] ← 30; A[7] ← 70
7. > A
8. > InterChange[3,7]
9. > A

On line 1, you told the debugger the module that you were interested in.

On line 2, you used the interpreter to examine the variables "A" (an array) and "j" (a long cardinal). (The Interpreter is invoked by typing a space at the beginning of a line.) Their values look unfamiliar; they weren't initialized because the module hadn't been started (which explains the warning that you got in the debugger: "{global frame number} is not started").

On line 3, you made an interpreted call to the procedure **Factorial** in module **MiscProcs**. You passed the necessary parameter (in this case, a cardinal), and it returned to you an answer (the factorial of your number). This number may have been in an octal format (denoted by the "B" after the number). (Note again that you were warned, before the procedure call, that the module had not been started).

On line 4, you interpreted the number "170B" by typing a "?" after the number. The reason that you did this was because the answer returned by procedure **Factorial** was probably in octal format, and you wanted to see what the answer was in decimal format. What you got when you interpreted "170B" was its value in octal, hexadecimal, decimal, ascii, and other formats. You can also change the default format by invoking **Options!** in the Sword FormSW and changing the value of the enumerated type for cardinals.

On line 5, you re-examined the variables "A" and "j" and found that they contained values that were assigned to them in the program ("A" initialized to all zeroes and "j" being set by the Factorial call to 120). How did the variables become set? You executed code in the module when you called the procedure Factorial. This caused all the global variables in the module to be initialized.

On line 6, you stuffed your own values into the 3rd and 7th spots in the array "A."

On line 7, you examined "A" to make sure that the array contained your values.

On line 8, you made an interpreted call to the procedure InterChange, which interchanges the two values in the spots in the array that you specified (in this case, the third and seventh spot).

On line 9, you re-examined "A" to check that the values for the 3rd and 7th spot had been interchanged.

Your debugger should look similar to the following:

```
go: {proceed, abort, kill, screen, start}    Client: {local}    destroy!
read: { }  write : { }  processes  configs  attach: {source, symbols}
source!  findModule!  rep?!  showType!  type&bits!  options!
break: {set, clear, clearall, list, attachCond, attachKey}  watch: {off}

You called?
>SEt Module context: MiscProcs
> A; j
112560B is not started!
A = (13)[1, 2, 6400B, 17B, 20B, 20156B, 67564B, 20146B, 67565B, 67144B, 20141B,
67144B, 20000B]
112560B is not started!
j = 4640650441B
> Factorial[5]
112560B is not started!
170B
> 170B?
170B = 78X = 120 = 'x = 7:8
> A; j
A = (13)[0,0,0,0,0,0,0,0,0,0,0,0,0]
j = 170B
> A[3] ← 30; A[7] ← 70
> A
A = (13)[0,0,0,30,0,0,0,70,0,0,0,0,0]
> InterChange [3, 7]
> A
A = (13)[0,0,0,70,0,0,0,30,0,0,0,0,0]
>
```

Now, try the following in the debugger:

```
10. > MakeLinkedList[4]
11. > headNode
12. > headNode↑
13. > headNode.next↑
14. > headNode.next.next↑
15. > headNode.next.next.next↑
```

On line 10, you made a call to the procedure `MakeLinkedList`, which creates a singly-linked list where the size is specified by the caller (in this case, the size is 4). The global variable `headNode` is a pointer variable that acts as the header for this linked list.

On line 11, you examined the value of `headNode` and found the *address* of the record that it points to. You know that it's an address by the up-arrow that follows the returned number.

On line 12, you asked to *dereference* the pointer `headNode` and examined the contents of what `headNode` points to. Notice the field `next` and the fact that it contains a number with an up-arrow after it. This field points to the next element in the linked list. (The other field in this record, `str`, is a LONG STRING of length = 1 and maxlength = 1 [hence the "(1,1)"] that contains the text "D".)

On line 13, you asked to examine the contents of what the `next` field points to. Notice that you did not have to type `headNode↑.next↑`, only `headNode.next↑`, due to the *auto-dereferencing* feature that exists in the Mesa language and is included in the interpreter.

On line 14, you examined the contents of what the next `next` field points to.

On line 15, you looked at the final element in the linked list. Notice that the `next` field for this last element is `NIL`.

The last part of your debugger should look similar to the following.

```
go: {proceed, abort, kill, screen, start}    Client: {local}    destroy!
read: { }  write : { }  processes  configs  attach: {source, symbols}
source!  findModule!  rep?!  showType!  type&bits!  options!
break: {set, clear, clearall, list, attachCond, attachKey}  watch: {off}

> MakeLinkedList[4]
> headNode
headNode = 4021731B↑
> headNode↑
headNode = [str:4021736B↑(1,1)"D", next:4021742B↑]
> headNode.next↑
next = [str:4021747B↑(1,1)"C", next:4021753B↑]
> headNode.next.next↑
next = [str:4021760B↑(1,1)"B", next:4021764B↑]
> headNode.next.next.next↑
next = [str:4021771B↑(1,1)"A", next:NIL]
>
```

Second Example: "Function"

This example takes you through a fairly realistic debugging session. Suppose that you have just written a program called `Function.mesa` and you want to test it to see if it gives the correct answers, and if it catches human errors in input and handles them in a desirable way. Load `Function.mesa` in a window, then compile (your favorite way) and run it from the executive (this time, with no switches.) Also, in the executive type 'help function' to see how to use the program. Test the program by entering the following commands in the Executive:

```
function s/20
function c/5
function s/6 s/10 c/4
function q/
```

At this point, the program fails and the debugger gets called. (If a debugger window is not automatically brought up, search the active windows looking for the local debugging window.) In the debugger, you're told that an uncaught **SIGNAL** caused the debugger to be invoked. **SIGNALs** have names, and if this signal can't be identified, it is because the symbols for the module that raised the signal are not present on your disk. In this case, the module that contains the signal is `StringsImplB`. You may not have the symbols for the module `StringsImplB`, and you, therefore, don't have any information other than the fact that the program `Function.bcd` had made a call into `StringsImplB`. If you don't already have the file `StringsImplB.bcd` on your local disk, ask a lab assistant to give you the file. After you have loaded the file onto your machine, type the letters "rd" in the debugger:

```
>ReDisplay swap reason
```

The debugger will now consider the new information (the file `StringsImplB.bcd`) and tell you what the signal was (`String.InvalidNumber`) that caused the debugger to be called. You can now display the stack and run-time variables to determine what caused the uncaught **SIGNAL**. In the following, type what's underlined into the debugger:

1. >Display Stack -- *Display the first element on the run-time stack*
2. >n -- *Look at the line in the source code that executed*
3. >s -- *Look at the local variables in this procedure*
4. >v

On line 1, you asked to display the most recent call on the run-time stack. It was a procedure in `StringsImplB`, and since we are looking for our code in `Function`, we go on to the next item on the run-time stack. (The format of each line of the stack is this: procedureName, local frame pointer, moduleName, globalFrameNumber.)

On line 2, you asked to see the next call on the run-time stack. This time, you recognized one of your modules on the stack, namely "Function." You saw that the procedure being executed was procedure `Main` when a call was made to a procedure in `StringsImplB`. (Generally, when debugging an error like this, you want to keep using the "next" command until you see a procedure in *your* code.)

On line 3, you displayed the source-code line that called the procedure in `StringsImplB`. It was a call through the `String` interface, and the call was to `String.StringToNumber`. If you look in your *Pilot Programmer's Manual* (version 12.0, p. 7-7), you will see that this procedure interprets an input string as an number and returns its value.

On line 4, you inspected the variables that are local to the procedure on the stack. Notice that the string variable that was passed to `String.StringToNumber`, called `number`, is `NIL`. An educated guess would be that this `NIL` string caused the problem because `String.StringToNumber` wouldn't expect a `NIL` string.

Your debugger should look something like this:

```
go: {proceed, abort, kill, screen, start}    Client: {local}    destroy!  
read: { }  write : { }  processes  configs  attach: {source, symbols}  
source!  findModule!  rep?!  showType!  type&bits!  options!  
break: {set, clear, clearall, list, attachCond, attachKey}  watch: {off}   
  
***  uncaught SIGNAL InvalidNumber (in StringsImp1B, G:37420B)  ***  
>Display Stack  
No symbols for L: 14130B, PC: 4860B (in StringsImp1B, G: 37420B)  >n  
Main, L: 10570B, PC: 337B (in Function, G: 112570B)  >s  <>cardinal ←  
String.StringToNumber[number, 10];  
  >v  
  h = 410720B+  
  clientData = NIL  
  outcome = normal  
  OutputProc = PROCEDURE [10756B] (in module ExecImp1, G: 35004B)  
  operation = 3314464B+(1,100)"q"  
  number = NIL  
  cardinal = 0  
  answer = 0  
  >
```

Now you need to determine how this situation got past your error checking. In the procedure **Main**, the line immediately before the call to **String.StringToNumber** is:

```
IF (operation = NIL) AND (number = NIL) THEN EXIT;
```

Notice that **operation** is not **NIL** in this case, but **number** is. This is the situation that should be caught because it produces an **uncaught SIGNAL** if it isn't. The correct error-catching code should be:

```
IF (operation = NIL) OR (number = NIL) THEN EXIT;
```

Make this change to **Function.mesa**. Before you can recompile your new version, you must **Abort** the current version by clicking the **abort** command in the **Sword** formSW and then unloading **Function** in the **Executive**. Then re-compile and run your new version.

More on Uncaught Signals

This part of the exercise will show you how to stop uncaught signals from entering the debugger. Now try this command in the executive:

```
function s/4k6
```

This should cause the local debugger to be invoked with another `Uncaught SIGNAL` in `StringsImplB`. The signal should be `String.InvalidNumber`. Now display the stack and run-time variables to determine what caused the uncaught signal this time. If you look at the Pilot Programmer's Manual (Ch. 7), you'll see that the procedure `String.StringToNumber` can raise the signal `String.InvalidNumber`. In order to make `Function.mesa` catch the signal you need to change:

```
cardinal ← String.StringToNumber[number, 10];
```

to:

```
cardinal ← String.StringToNumber[number, 10 !String.InvalidNumber => {  
  OutputProc["Bad number...continuing..."];  
  CONTINUE }];
```

You don't really need to understand what an uncaught signal is for now; signals will be covered in-depth later this week. Hopefully, what you should get out of this example is the knowledge of how to debug an uncaught signal.

Remember, if you don't have the symbols for the module that raised the signal, you may want to retrieve them onto your machine and redisplay the reason for the swap (from the debugger).

If you already have the symbols on your machine, you do not have to retrieve anything; the debugger will tell you what the uncaught signal was since it will have all the information it needs. Often, you will be able to debug such errors without ever retrieving the symbols: just proceed up the stack until you find one of your procedures, and then use the `Source` command to find the line of code where it died. This will often be enough information to enable you to debug the problem.

Setting Breakpoints

Now, type the following in the Executive:

```
function q/  
function s/100  
function c/100
```

The answers you got for the last query should seem a little strange. 100 cubed is not 16960. At this point, it is a good idea to set breakpoints in the code to see why the wrong answer occasionally gets returned. To set breakpoints, the program must always be loaded first. Since, we have already run our program, then it must be loaded!

Suppose you suspect that the variables contain the correct information, but they are not being printed out correctly in the procedure `PrintResult`. Find a local debugging window and type the underlined parts into the fileSW to set a breakpoint at the beginning of `PrintResult`:

1. `>SEt Module context: Function`
2. `>BreaK Entry procedure: PrintResult`
3. `>Proceed (Confirm) -- Type a carriage return at the end of this line to confirm`

Now try the following command in the Executive:

```
function c/100
```

When the procedure `PrintResult` is entered, the breakpoint will be executed, thus causing the debugger to be called. In the debugger, examine the parameters for `PrintResult`:

4. >Display Stack
5. >p

The value that's given to `PrintResult` is 16960, so the problem is not in printing out the result; the result itself is incorrect. Now try setting breakpoints on the procedures that actually calculate the results:

6. >g
7. >Clear Break #: 1
8. >Break Xit procedure: CubeInput
9. >Proceed (Confirm)

Again, try "function c/100" in the Executive. You should hit the breakpoint that you set in the procedure `CubeInput`. (If you get a Stack Error first, just proceed and you will eventually hit your breakpoint.) Use the debugger to examine the return parameter (type "r" while in Display Stack mode) for `CubeInput`; it should be equal to 16960. Therefore, you know that the calculation itself is incorrect.

If you remember from the lecture this morning, multiplication of two CARDINALs will yield a CARDINAL. In this case, multiplying the three CARDINALs (100 and 100 and 100) resulted in an answer outside the range of CARDINAL (which is $[0..2^{16}]$). The number 16960 is the modulo of $(100^3 / 2^{16})$. The way to fix this is to make the answer a LONG CARDINAL so that the overflow cases will not occur.

You should change the following two lines of code in `Function.mesa` before re-compiling it and running `Function.bcd` again (remember to abort and unload before re-compiling):

(In procedure `SquareInput...`)

```
RETURN [input * input];          change to...    RETURN [LONG[input] * input];
```

(In procedure `CubeInput...`)

```
RETURN [input * input * input];  change to...    RETURN [LONG[input] * input * input];
```

"LONG" will force one of the multiplicands to be a LONG CARDINAL; hence, the answer will be a LONG CARDINAL because operations involving CARDINALs and LONG CARDINALs result in LONG numbers.

Debugger Exercises

All of the following source files include comments at the beginning of the file explaining what the program does. Read those comments! Compile the programs using your favorite method and then run them each from the executive. Then, follow the instructions below. Each program will crash and you should find and fix the problem causing the crashes. Debugging techniques will be required to fix the programs.

Part I: Hash.mesa

Run the following commands from the Exec:

```
Help Hash
Hash lu mark john
Hash john/d
Hash john/d
```

At this point, you should get an address fault. If there isn't a current local debugging session, a local debugging window will be created for you; otherwise, search through the active windows for the local debugging session. Use the debugger to your advantage. It can help you. Really! After you fix the problem, abort the debugging session and unload the program from the executive. Then re-compile and re-run your new version. Make sure that all the bugs are out by running the following commands:

```
Hash lu mark
Hash john/d
```

Part II: BubbleSortProgram.mesa

Run the following commands from the Exec:

```
Help BubbleSort
BubbleSort 2 7 4 1 10
BubbleSort 9 7 4 12 31 16 4 28 1 32
BubbleSort 5 18 7 22 10 11 63 22 84 24
```

You should get an uncaught signal at some point. After you fix the problem, abort, unload, recompile, and re-run (same steps as above) and then make sure that all the bugs are out by running the following commands:

```
BubbleSort 13 19 34 81 18 56 23 44 46
BubbleSort 28 20 4 17 11 18 19 68 1 42
```

If that input works, then try to input just one number:

```
BubbleSort 13
```



```

    else Put.Line[data.fileSW, "undefined"L];
      ↑ Syntax Error [4047]
Text inserted is: ;
-- Capitolize ELSE

};
↑ Syntax Error [5281]
Text inserted is: ENDCASE
-- Notice the SELECT statement. For every SELECT there should be an ENDCASE. Insert one here.

```

Trial # 3

```

    ELSE Put.Line[data.fileSW, "undefined"L];
      ↑ Syntax Error [4041]
Text deleted is: ELSE
-- There should not be a colon separating the THEN-part from the ELSE-part of the IF-THEN-ELSE
statement.

```

Trial # 4

```

Integer is undeclared, at CompilerPractice[1021]:
  x(6): Long Integer ← 0,
-- Again, ALL Mesa Reserved Words must be Capitolized - so, Capitolize INTEGER

```

```

Long does not name a variant, at CompilerPractice[1021]:
  x(6): Long Integer ← 0,
-- Capitolize LONG

```

```

Integer is undeclared, at CompilerPractice[1049]:
  y(8): Long Integer ← NIL];
-- Capitolize INTEGER

```

```

Long does not name a variant, at CompilerPractice[1049]:
  y(8): Long Integer ← NIL];
-- Capitolize LONG

```

Trial # 5

```

.NIL has incorrect type, at CompilerPractice[1049]:
  y(8): LONG INTEGER ← NIL];
-- A LONG INTEGER should be initialized to zero not NIL.

```

Trial # 6

```

Command: CompilerPractice
CompilerPractice.mesa
lines: 220, code: 1774, links: 22, frame: 11, time: 52

```

```

-- After correcting all of the above errors, CompilerPractice.mesa should compile correctly!!!

```

Compiler Practice Solution

This is the .mesa file with no syntax errors. Following this listing is a listing of the compiler errors that were encountered and the fixes that were made.

```
-- File: CompilerPracticeSolution.mesa - - Last edited by:
-- MacKay 16-May-86 12:26:30
-- Create by FormSWLayoutTool on 15-May-86 10:15A11 rights reserved.
```

DIRECTORY

```
Exec USING [AddCommand, ExecProc, Handle, OutputProc, RemoveCommand],
Format USING [StringProc],
FormSW USING [AllocateItemDescriptor, ClientItemsProcType, CommandItem, line0, line2,
LongNumberItem, ProcType],
Heap USING [systemZone],
Process USING [Detach],
Put USING [CR, Line, LongNumber, Text],
Tool USING [Create, Destroy, MakeFileSW, MakeFormSW, MakeMsgSW, MakeSWsProc, UnusedLogName],
ToolWindow USING [TransitionProcType],
Window USING [Box, Handle],
WindowFont USING [CharWidth];
```

CompilerPractice: MONITOR

IMPORTS

```
Exec, FormSW, Heap, Process, Put, Tool, WindowFont = {
```

```
DataHandle: TYPE = LONG POINTER TO Data;
```

```
Data: TYPE = MACHINE DEPENDENT RECORD [
msgSW(0): Window.Handle ← NIL,
formSW(2): Window.Handle ← NIL,
fileSW(4): Window.Handle ← NIL,
x(6): LONG INTEGER ← 0,
y(8): LONG INTEGER ← 0];
```

```
FormItems: TYPE = {x, add, subtract, ytotheX, y, multiply, divide};
```

```
data: DataHandle ← NIL;
```

```
wh: Window.Handle ← NIL;
```

```
zone: UNCOUNTED ZONE ← Heap.systemZone;
```

```
answer: LONG INTEGER ← 0;
```

```
initialBox: Window.Box = [place: [x:436, y: 82], dims: [w: 512, h: 555]];
```

```
busyBit: BOOLEAN ← FALSE;
```

```
Busy: ENTRY PROCEDURE RETURNS [isBusy: BOOLEAN] = {
```

```
ENABLE UNWIND => NULL;
```

```
isBusy ← busyBit;
```

```
busyBit ← TRUE };
```

```
Done: ENTRY PROCEDURE = {
```

```
ENABLE UNWIND => NULL;
```

```
busyBit ← FALSE};
```

```
Write: Format.StringProc = {Put.Text[data.fileSW, s]};
```

```
Msg: Format.StringProc = {Put.Text[data.msgSW, s]};
```

```
Add: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L];    RETURN};
  Process.Detach[FORK AddInternal[]];
}
```

```
AddInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  answer ← data.x + data.y;
  Put.LongNumber[data.fileSW, data.x, [unsigned: FALSE]];
  Put.Text[data.fileSW, " plus "L];
  Put.LongNumber[data.fileSW, data.y, [unsigned: FALSE]];
  Put.Text[data.fileSW, " is "L];
  Put.LongNumber[data.fileSW, answer, [unsigned: FALSE]];
  Put.CR[data.fileSW];
  Done[] };
}
```

```
Subtract: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L];    RETURN};
  Process.Detach[FORK SubtractInternal[]];
}
```

```
SubtractInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  answer ← data.x - data.y;
  Put.LongNumber[data.fileSW, data.x, [unsigned: FALSE]];
  Put.Text[data.fileSW, " minus "L];
  Put.LongNumber[data.fileSW, data.y, [unsigned: FALSE]];
  Put.Text[data.fileSW, " is "L];
  Put.LongNumber[data.fileSW, answer, [unsigned: FALSE]];
  Put.CR[data.fileSW];
  Done[] };
}
```

```
Multiply: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L];    RETURN};
  Process.Detach[FORK MultiplyInternal[]];
}
```

```
MultiplyInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  answer ← data.x * data.y;
  Put.LongNumber[data.fileSW, data.x, [unsigned: FALSE]];
  Put.Text[data.fileSW, " times "L];
  Put.LongNumber[data.fileSW, data.y, [unsigned: FALSE]];
  Put.Text[data.fileSW, " is "L];
  Put.LongNumber[data.fileSW, answer, [unsigned: FALSE]];
  Put.CR[data.fileSW];
  Done[] };
}
```

```
Divide: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L];    RETURN};
  Process.Detach[FORK DivideInternal[]];
}
```

```

DivideInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  Put.LongNumber[data.fileSW, data.x, [unsigned: FALSE]];
  Put.Text[data.fileSW, " divided by "L];
  Put.LongNumber[data.fileSW, data.y, [unsigned: FALSE]];
  Put.Text[data.fileSW, " is "L];
  IF data.y # 0 THEN {
    answer ← data.x / data.y;
    Put.LongNumber[data.fileSW, answer, [unsigned: FALSE]];
    Put.CR[data.fileSW];
    Done[];}
  ELSE Put.Line[data.fileSW, "undefined"L];
  Done[] };

```

```

YtotheX: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L]; RETURN;
  }
  Process.Detach[FORK YtotheXInternal[]];

```

```

YtotheXInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF data.x = 0 THEN answer ← 1
  ELSE {
    answer ← data.y;
    THROUGH [1..data.x) DO
      answer ← answer * data.y;
    ENDOOP;};
  Put.LongNumber[data.fileSW, data.y, [unsigned: FALSE]];
  Put.Text[data.fileSW, " raised to the power of "L];
  Put.LongNumber[data.fileSW, data.x, [unsigned: FALSE]];
  Put.Text[data.fileSW, " is "L];
  Put.LongNumber[data.fileSW, answer, [unsigned: FALSE]];
  Put.CR[data.fileSW];
  Done[] };

```

```

ClientTransition: ToolWindow.TransitionProcType = {
  SELECT TRUE FROM
    old = inactive =>
      IF data = NIL THEN data ← zone.NEW[Data ← []];
    new = inactive =>
      IF data # NIL THEN {
        zone.FREE[@data];
      }
  ENDCASE;
};

```

-- The rest of the code is used for building the tool window. You will learn about this on Wednesday, so don't worry if you don't understand it. There are not any compiler errors in these procedures.

```

Init: PROCEDURE = {
  Exec.AddCommand["CompilerPractice."L, NoOp, NIL, Unload];
  wh ← Tool.Create[
    makeSWsProc: MakeSWs, initialState: default, initialBox: initialBox,
    clientTransition: ClientTransition, name: "CompilerPractice"L,
    cmSection: "CompilerPractice"L ] };

NoOp: Exec.ExecProc = { };

Unload: Exec.ExecProc = {
  IF Busy[] THEN {
    Exec.OutputProc[h]["Tool is busy. Sorry.\n"L];
    RETURN[error] };
  Tool.Destroy[wh];
  Exec.RemoveCommand[h, "CompilerPractice."L];
  Done[] };

MakeSWs: Tool.MakeSWsProc = {
  logName: LONG STRING ← [23];
  Tool.UnusedLogName[unused: logName, root: "CompilerPractice.log"L];
  data.msgSW ← Tool.MakeMsgSW[window: window];
  data.formSW ← Tool.MakeFormSW[
    window: window, formProc: MakeForm];
  data.fileSW ← Tool.MakeFileSW[window: window, name: logName] };

charWidth: CARDINAL ← WindowFont.CharWidth['0'];
CharPos: PROC[char: CARDINAL] RETURNS [x: INTEGER] = {
  x ← charWidth * char };

MakeForm: FormSW.ClientItemsProcType = {
  OPEN FormSW;
  nItems: CARDINAL = FormItems.LAST.ORD + 1;
  items ← AllocateItemDescriptor[nItems];
  items[FormItems.x.ORD] ← LongNumberItem[
    tag: "X"L, place: [CharPos[1], line0], signed: FALSE, value: @data.x];
  items[FormItems.add.ORD] ← CommandItem[
    tag: "ADD"L, place: [CharPos[34], line0], proc: Add];
  items[FormItems.subtract.ORD] ← CommandItem[
    tag: "SUBTRACT"L, place: [CharPos[47], line0], proc: Subtract];
  items[FormItems.ytotheX.ORD] ← CommandItem[
    tag: "Y to the X"L, place: [CharPos[66], line0], proc: YtotheX];
  items[FormItems.y.ORD] ← LongNumberItem[
    tag: "Y"L, place: [CharPos[1], line2], signed: FALSE, value: @data.y];
  items[FormItems.multiply.ORD] ← CommandItem[
    tag: "MULTIPLY"L, place: [CharPos[34], line2], proc: Multiply];
  items[FormItems.divide.ORD] ← CommandItem[
    tag: "DIVIDE"L, place: [CharPos[47], line2], proc: Divide];
  RETURN[items: items, freeDesc: TRUE] };

-- Mainline code
Init[]; -- this gets string out of global frame *

}...

```

Here is a listing of the 6 different compiler logs that it took to get the program compiled, with error meanings and the fixes that should be made. Keep in mind that depending on how you interpreted the errors, you might not get the same subsequent errors.

Trial # 1

DIRECTORY:

↑ Syntax Error [218]

Text deleted is: :

-- Just delete the colon

Window USING [Box, Handle],

↑ Syntax Error [654]

Text deleted is: USING

Text inserted is: ([

-- Notice the line above this one. The semi-colon indicates the end of the DIRECTORY clause, which shouldn't end here. The semi-colon should be replaced with a comma.

WindowFont USING [CharWidth];

↑ Syntax Error [688]

No recovery found.

-- This error was caused by the misplaced semi-colon, also. When that correction is made, this error will go away.

Trial # 2

Exec, FormSW, Heap, Process, Put, Tool, WindowFont, = {

↑ Syntax Error [799]

Text inserted is: id

-- Delete the comma following "WindowFont"

Put.Text(data.fileSW, " plus "L);

↑ Syntax Error [2025]

Text deleted is: (id . id ,

Text inserted is: ; (

-- Replace the opening AND closing parentheses with square brackets.

Put.CR(data.fileSW);

↑ Syntax Error [2217]

Text inserted is: ;

-- Same error as above

if data.y # 0 then {

↑ Syntax Error [3890]

Text inserted is: _

-- The problem here is caused from 'if' not being in caps, so the compiler doesn't recognize it as a reserved word. Solution: capitalize IF

if data.y # 0 then {

↑ Syntax Error [3901]

Text deleted is: id

Text inserted is: ;

-- Capitalize THEN

Solutions to Debugger Exercises

Part I: Hash.mesa

An address fault occurs when you are searching for an element to be deleted and the element is not in the list. To fix this error, add some error checking to make sure that the element is in the list before you try to delete it.

```
DeleteRec: PROCEDURE [string: LONG STRING + NIL] RETURNS [BOOLEAN + TRUE] = BEGIN
  ptr, ptr2: Handle;
  bucket: CARDINAL;
  [ptr, bucket] ← FindRec[string];
  IF ptr = NIL THEN RETURN[FALSE];
  ptr2 ← table[bucket];
  UNTIL (ptr2.mySibling = NIL) OR (ptr2.mySibling = ptr) DO
    ptr2 ← ptr2.mySibling;
  ENDOLOOP;
  IF ptr2.mySibling = ptr THEN BEGIN
    ptr2.mySibling ← ptr.mySibling;
    RETURN[TRUE];
  END
  ELSE RETURN[FALSE];
END;
```

Part II: BubbleSortProgram.mesa

An Uncaught Signal (BoundsFault) occurs because the comparison statement in the FOR loop tries to access one node larger than the allocated array size. This was caused by a square bracket (which is inclusive) instead of a parenthesis (which is exclusive) in FOR statement.

```
DoTheBubbleSort: PROCEDURE [upperBound: CARDINAL] RETURNS [BOOLEAN, CARDINAL] =
  BEGIN
    exchangeMade: BOOLEAN ← FALSE;
    position: CARDINAL ← 0;
    FOR j: CARDINAL IN [0..upperbound-1] DO
      IF A[j] > A[j + 1] THEN {
        InterChange[j, j+1];
        position ← j;
        exchangeMade ← TRUE;
      };
    ENDOLOOP;
    RETURN[exchangeMade, position];
  END; --DoTheBubbleSort
```

Note: The range [0..upperbound-2] will not work when upperbound < 2.

Program Example

Math Tool			
First Number= 22	Second Number= 17		
Product!	Sum!	Difference!	Quotient!
The product of 22 and 17 is 374			
The sum of 22 and 17 is 39			
The difference of 22 and 17 is 5			
The quotient of 22 and 17 is 1			

```
--Math.mesa  
--John Erskine  
--14-Dec-84 17:30:08
```

```
DIRECTORY
```

```
Window USING [Handle];
```

```
Math: DEFINITIONS =
```

```
BEGIN
```

```
Add: PROCEDURE [output: Window.Handle, n1, n2: INTEGER];
```

```
Divide: PROCEDURE [output: Window.Handle, n1, n2: INTEGER];
```

```
Multiply: PROCEDURE [output: Window.Handle, n1, n2: INTEGER];
```

```
Subtract: PROCEDURE [output: Window.Handle, n1, n2: INTEGER];
```

```
END..
```



```
-- MathControl.mesa - Last Edited on:
-- 13-Aug-87 13:19:23
-- Create by FormSWLayoutTool on 12-Aug-87 17:57
```

DIRECTORY

```
Exec USING [AddCommand, ExecProc, OutputProc, RemoveCommand],
Format USING [StringProc],
FormSW USING [AllocateItemDescriptor, ClientItemsProcType, CommandItem, line0, line2,
  LongNumberItem, ProcType],
Heap USING [Create],
Math USING [Add, Divide, Multiply, Subtract],
Process USING [Detach],
Put USING [Text],
Tool USING [Create, Destroy, MakeFileSW, MakeFormSW, MakeMsgSW, MakeSWsProc, UnusedLogName],
ToolWindow USING [TransitionProcType],
Window USING [Handle];
```

MathControl: MONITOR

IMPORTS

```
Exec, FormSW, Heap, Math, Process, Put, Tool = {
```

```
DataHandle: TYPE = LONG POINTER TO Data;
```

```
Data: TYPE = MACHINE DEPENDENT RECORD [
```

```
  msgSW(0): Window.Handle ← NIL,
```

```
  formSW(2): Window.Handle ← NIL,
```

```
  fileSW(4): Window.Handle ← NIL,
```

```
  firstNumber(6): LONG INTEGER ← 0,
```

```
  secondNumber(8): LONG INTEGER ← 0];
```

```
FormItems: TYPE = {firstNumber, secondNumber, sum, difference, product, quotient};
```

```
data: DataHandle ← NIL;
```

```
wh: Window.Handle ← NIL;
```

```
myZone: UNCOUNTED_ZONE ← Heap.Create[initial: 4];
```

```
busyBit: BOOLEAN ← FALSE;
```

```
Busy: ENTRY PROCEDURE RETURNS [isBusy: BOOLEAN] = {
```

```
  ENABLE UNWIND => NULL;
```

```
  isBusy ← busyBit;
```

```
  busyBit ← TRUE };
```

```
Done: ENTRY PROCEDURE = {
```

```
  ENABLE UNWIND => NULL;
```

```
  busyBit ← FALSE };
```

```
Write: Format.StringProc = {Put.Text[data.fileSW, s]};
```

```
Msg: Format.StringProc = {Put.Text[data.msgSW, s]};
```

```
Sum: FormSW.ProcType = {
```

```
  ENABLE ABORTED => {Done[]; CONTINUE};
```

```
  IF Busy[] THEN {
```

```
    Msg["Tool is busy.\n"L]; RETURN};
```

```
  Process.Detach[FORK SumInternal[]];
```

```
SumInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  Math.Add[data.fileSW, data.firstNumber, data.secondNumber];
  Done[] };

Difference: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L]; RETURN};
  Process.Detach[FORK DifferenceInternal[]]];

DifferenceInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  Math.Subtract[data.fileSW, data.firstNumber, data.secondNumber];
  Done[] };

Product: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L]; RETURN};
  Process.Detach[FORK ProductInternal[]]];

ProductInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  Math.Multiply[data.fileSW, data.firstNumber, data.secondNumber];
  Done[] };

Quotient: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN {
    Msg["Tool is busy.\n"L]; RETURN};
  Process.Detach[FORK QuotientInternal[]]];

QuotientInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  Math.Divide[data.fileSW, data.firstNumber, data.secondNumber];
  Done[] };

ClientTransition: ToolWindow.TransitionProcType = {
  SELECT TRUE FROM
    old = inactive =>
      IF data = NIL THEN data ← myZone.NEW[Data ← []];
    new := inactive =>
      IF data # NIL THEN myZone.FREE[@data];
  ENDCASE;
};

Init: PROCEDURE = {
  Exec.AddCommand["MathTool."L, NoOp, NIL, Unload];
  wh ← Tool.Create[
    makeSWsProc: MakeSWs, initialState: default,
    clientTransition: ClientTransition, name: "MathTool"L,
    cmSection: "MathTool"L];
};
```

```
NoOp: Exec.ExecProc = { };
```

```
Unload: Exec.ExecProc = {  
  IF Busy[] THEN {  
    Exec.OutputProc[h]["Tool is busy. Sorry.\n"L];  
    RETURN[error] };  
  Tool.Destroy[wh];  
  Exec.RemoveCommand[h, "MathTool."L];  
  Done[] };
```

```
MakeSws: Tool.MakeSwsProc = {  
  logName: LONG STRING ← [15];  
  Tool.UnusedLogName[unused: logName, root: "MathTool.log"L];  
  data.msgSW ← Tool.MakeMsgSW[window: window];  
  data.formSW ← Tool.MakeFormSW[  
    window: window, formProc: MakeForm, zone: myZone];  
  data.fileSW ← Tool.MakeFileSW[window: window, name: logName];  
  };
```

```
MakeForm: FormSW.ClientItemsProcType = {  
  nItems: CARDINAL = FormItems.LAST.ORD + 1;  
  items ← FormSW.AllocateItemDescriptor[nItems, myZone];  
  items[FormItems.firstNumber.ORD] ← FormSW.LongNumberItem[  
    tag: "First Number"L, place: [6, FormSW.line0], signed: FALSE,  
    value: @data.firstNumber, z: myZone];  
  items[FormItems.secondNumber.ORD] ← FormSW.LongNumberItem[  
    tag: "Second Number"L, place: [186, FormSW.line0], signed: FALSE,  
    value: @data.secondNumber, z: myZone];  
  items[FormItems.sum.ORD] ← FormSW.CommandItem[  
    tag: "Sum"L, place: [6, FormSW.line2], proc: Sum, z: myZone];  
  items[FormItems.difference.ORD] ← FormSW.CommandItem[  
    tag: "Difference"L, place: [90, FormSW.line2], proc: Difference, z: myZone];  
  items[FormItems.product.ORD] ← FormSW.CommandItem[  
    tag: "Product"L, place: [186, FormSW.line2], proc: Product, z: myZone];  
  items[FormItems.quotient.ORD] ← FormSW.CommandItem[  
    tag: "Quotient"L, place: [294, FormSW.line2], proc: Quotient, z: myZone];  
  RETURN[items: items, freeDesc: TRUE];  
  };
```

```
-- Mainline code
```

```
Init[]; -- this gets string out of global frame
```

```
...  
}
```

--MathImpl.mesa
--John Erskine
--14-Dec-84 17:40:29

DIRECTORY

Window USING [Handle],
Put USING [CR, Decimal, Text],
Math;

MathImpl: PROGRAM IMPORTS Put EXPORTS Math = {

Add: PUBLIC PROCEDURE [output: Window.Handle, n1, n2: INTEGER] = {
 Put.Text[output, "The sum of "L];
 Put.Decimal[output, n1];
 Put.Text[output, " and "];
 Put.Decimal[output, n2];
 Put.Text[output, " is "L];
 Put.Decimal[output, n1+n2];
 Put.CR[output];
};

Divide: PUBLIC PROCEDURE [output: Window.Handle, n1, n2: INTEGER] = {
 Put.Text[output, "The quotient of "L];
 Put.Decimal[output, n1];
 Put.Text[output, " and "];
 Put.Decimal[output, n2];
 Put.Text[output, " is "L];
 IF n2 = 0 THEN
 Put.Text[output, "infinity"]
 ELSE
 Put.Decimal[output, n1/n2];
 Put.CR[output];
};

Multiply: PUBLIC PROCEDURE [output: Window.Handle, n1, n2: INTEGER] = {
 Put.Text[output, "The product of "L];
 Put.Decimal[output, n1];
 Put.Text[output, " and "];
 Put.Decimal[output, n2];
 Put.Text[output, " is "L];
 Put.Decimal[output, n1*n2];
 Put.CR[output];
};

Subtract: PUBLIC PROCEDURE [output: Window.Handle, n1, n2: INTEGER] = {
 Put.Text[output, "The difference of "L];
 Put.Decimal[output, n1];
 Put.Text[output, " and "];
 Put.Decimal[output, n2];
 Put.Text[output, " is "L];
 Put.Decimal[output, n1-n2];
 Put.CR[output];
};

}...

```
--MathTool.config  
--John Erskine  
--14-Dec-84 17:49:17
```

```
MathTool: CONFIGURATION  
  IMPORTS Exec, FormSW, Heap, Tool, Process, Put
```

```
CONTROL MathControl = {
```

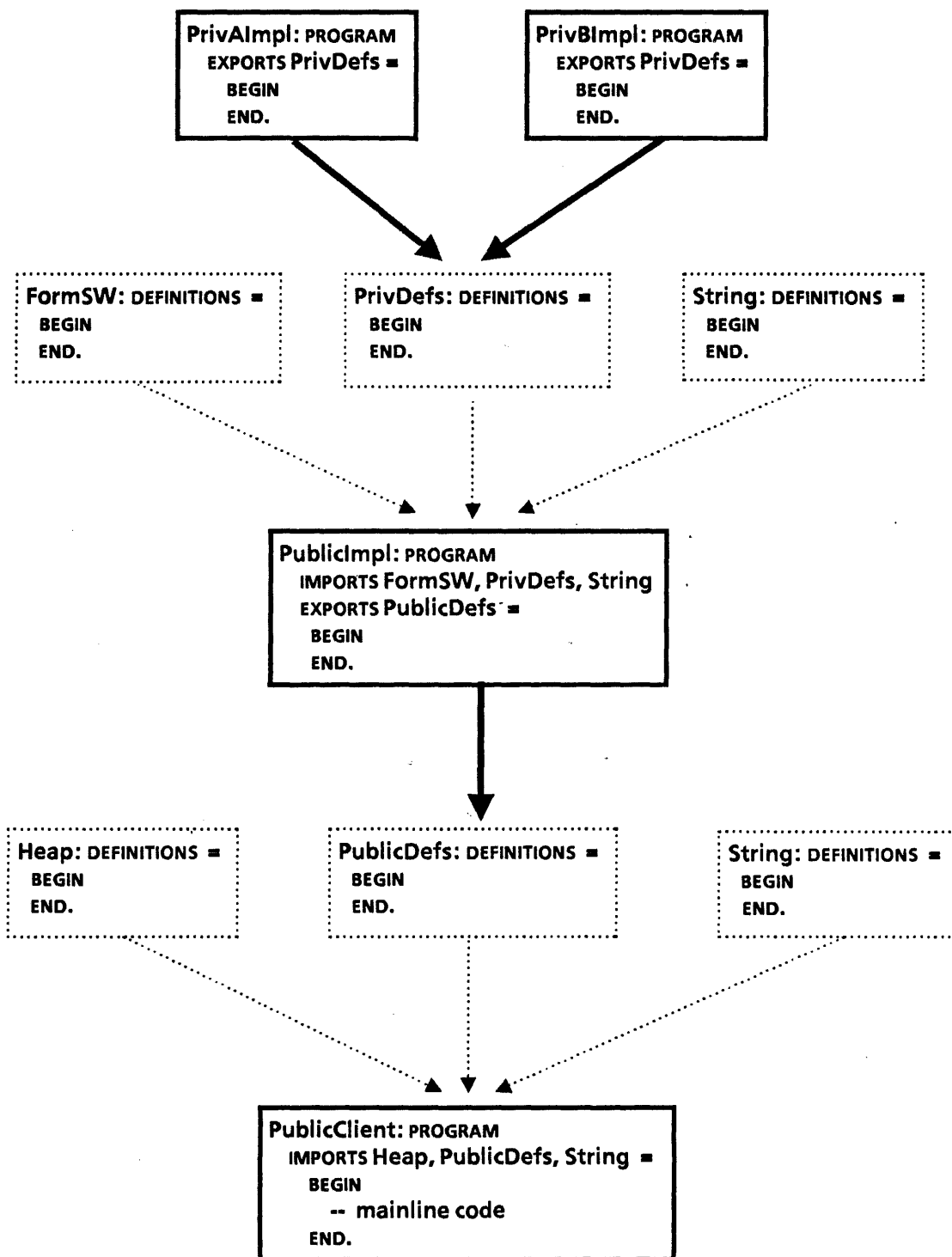
```
  MathControl;  
  MathImpl;
```

```
}...
```

In Class Exercise - Configurations

Write a CONFIGURATION file for a program consisting of the files shown below. (The bold-lined boxes are PROGRAM modules and the dotted-lined boxes are DEFINITIONS modules.)

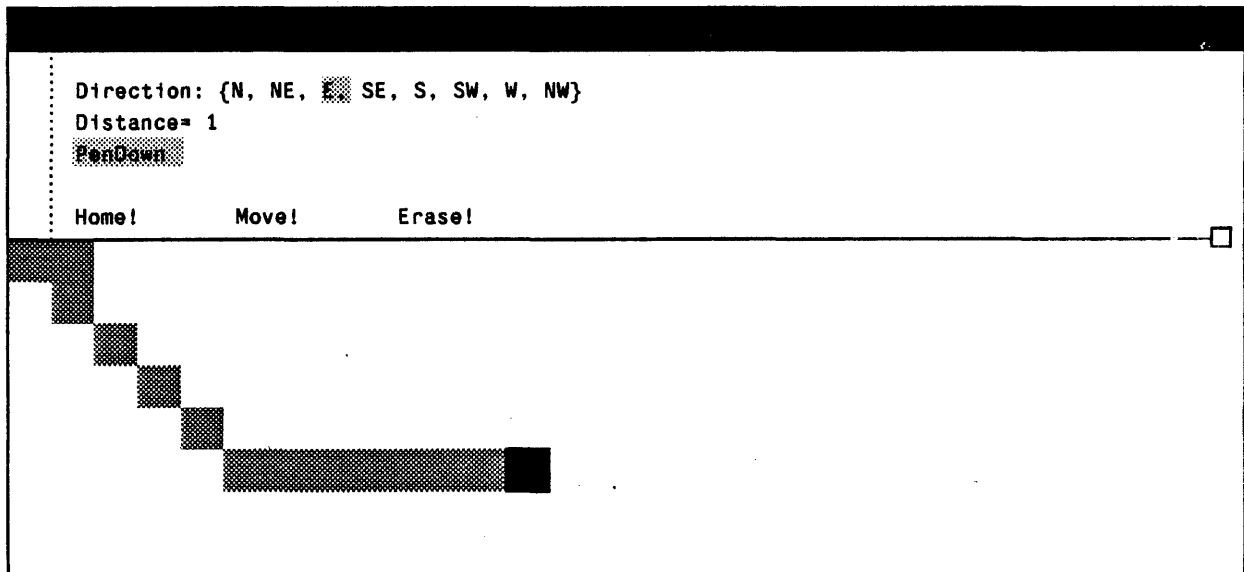
Note: Just write one configuration file for the entire application instead of a nested config within a config.



Interfaces Programming Assignment: Turtle

In this programming assignment, you need to provide the implementation of one interface using procedures defined in another interface. The finished product will implement the Turtle programming language:

Initially, a turtle is located at the top left-hand side of the window (coordinates 0, 0). This turtle carries a pen. You can tell the turtle to move different distances and different directions with his pen either lowered or raised by clicking the appropriate commands. While the pen is lowered, you will see the turtle leave a trail behind him as he moves.



The interface you will be implementing is called Turtle.mesa:

```
--Turtle.mesa
--2-Oct-84 11:14:51
```

```
DIRECTORY
```

```
Window USING [Handle];
```

```
Turtle: DEFINITIONS = {
```

```
  gridSize: CARDINAL = 32;                -- length and width of grid
  Home: PROCEDURE [output: Window.Handle]; -- Puts turtle at (0,0)
  PenUp: PROCEDURE;                       -- Raises the pen.
  PenDown: PROCEDURE;                     -- Lowers the pen.
  Move: PROCEDURE [output: Window.Handle, distance: CARDINAL, direction: [0 .. 8]];
      -- Moves the turtle by the amount in distance in the given
      -- direction. 0 is up (North), 1 is up 1 and right 1 (NorthEast),
      -- 2 is to the right (East), ...
  Erase: PROCEDURE [output: Window.Handle]; -- Completely clears the grid.
  Redraw: PROCEDURE [output: Window.Handle];
      -- Draws the grid from scratch when the window becomes active
}
```

You need to write an implementation module, `TurtleImpl.mesa`, that implements the procedures listed above. These procedures get called from the module `TurtleControl.mesa` when the user invokes the appropriate commands in the form subwindow. (`Redraw` is called when the user changes the size of the window. For example, `Redraw` will be called if you make the Turtle window tiny and then reactivate it.) You don't have to actually write the code that draws the boxes on the screen; instead, you should import procedures from the `Boxes` interface to do the actual drawing:

--`Boxes.mesa`

DIRECTORY

Window USING [Handle];

`Boxes`: DEFINITIONS = {

`DrawWhite`: PROCEDURE [output: Window.Handle, y, x: CARDINAL]; -- *Draws a white box*

`DrawGray`: PROCEDURE [output: Window.Handle, y, x: CARDINAL]; -- *Draws a gray box*

`DrawBlack`: PROCEDURE [output: Window.Handle, y, x: CARDINAL]; -- *Draws a black box*

};

These procedures will draw a box at the coordinates (y,x). [(0,0) is the top left-hand corner of the window shown in the tool. x increases to the right and y increases downward.] The implementation for the `Boxes` interface has already been written and is provided in the module `BoxesImpl.mesa`.

Note that the procedures `Home`, `Move`, `Erase`, and `ReDraw` are all passed in a parameter of type `Window.Handle`. This parameter is used to specify the window in which the painting is to be done. (Basically, it is just a pointer to the proper subwindow.) You don't have to do anything with this parameter except pass it along to the routines in `Boxes`. (Those routines need to know where to do the painting.)

In your implementation module, you will need a 2-dimensional packed array of `BOOLEANS` in order to determine where the turtle has been. The length of each dimension should be indicated by the constant `gridSize` in the interface `Turtle.mesa`. You'll have to keep track of whether the pen is up or down and where the turtle is at any given time. Your implementation should handle the case where the user instructs the turtle to go off the edge of the grid (outside the bounds of the array). Simply not moving is the easiest implementation. When the pen is down, everytime the turtle moves off a square, that square should be marked with a gray box. When the pen is up, the square should remain in its original condition.

Assignment

1. Write the implementation module `TurtleImpl.mesa` that implements the procedures in `Turtle.mesa`.
2. Write the configuration file `TurtleTool.config`.
3. Verify that your implementation is correct by running the tool.

Extra for Experts

In your implementation module `TurtleImpl.mesa`, change the implementation for the procedure `Home` so that when `Home` is called, the turtle will recursively look for a path home from his current position, and mark that path with all black squares. Limit the turtle to only the gray squares that up to this point have marked his wanderings. Allow for the possibility that there may not be a path home in which case you may either place the turtle home or leave him in his current position. When testing this extension, you may want to increase the constant `gridSize` in the definitions module `Turtle.mesa`.

Tool-Written Factorial Tool

<input type="checkbox"/>	
Number=	Format: {hex, octal, <u>decimal</u>}
Factorial!	<input type="checkbox"/>
The factorial of 4 is 24	

```
-- FactorialTool.mesa
-- Create by FormSWLayoutTool on 16-May-86 10:42
```

DIRECTORY

```
Exec,
Format,
FormSW,
Heap,
Process,
Put,
Tool,
ToolWindow,
Window;
```

```
FactorialTool: MONITOR IMPORTS Exec, FormSW, Heap, Process, Put, Tool = {
```

```
DataHandle: TYPE = LONG POINTER TO Data;
Data: TYPE = MACHINE DEPENDENT RECORD [
  msgSW(0): Window.Handle ← NIL,
  formSW(2): Window.Handle ← NIL,
  fileSW(4): Window.Handle ← NIL,
  number(6): UNSPECIFIED ← 0,
  format(7): UNSPECIFIED ← 0];
```

```
FormItems: TYPE = {number, format, factorial};
```

```
data: DataHandle ← NIL;
wh: Window.Handle ← NIL;
zone: UNCOUNTED_ZONE ← Heap.Create[initial: 4];
busyBit: BOOLEAN ← FALSE;
```

```
Busy: ENTRY PROCEDURE RETURNS [isBusy: BOOLEAN] = {
  ENABLE UNWIND => NULL;
  isBusy ← busyBit;
  busyBit ← TRUE };
```

```
Done: ENTRY PROCEDURE = {
  ENABLE UNWIND => NULL;
  busyBit ← FALSE };
```

```
Write: Format.StringProc = {Put.Text[data.fileSW, s]};
Msg: Format.StringProc = {Put.Text[data.msgSW, s]};
```

```

Factorial: FormSW.ProcType = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF Busy[] THEN { Msg["Tool is busy.\n"]; RETURN};
  Process.Detach[FORK FactorialInternal[]];
}

FactorialInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  Write["Factorial called\n"];
  Done[] };

ClientTransition: ToolWindow.TransitionProcType = {
  SELECT TRUE FROM
    old = inactive => IF data = NIL THEN data ← zone.NEW[Data ← []];
    new = inactive => IF data # NIL THEN { zone.FREE[@data] };
  ENDCASE };

Init: PROCEDURE = {
  Exec.AddCommand["FactorialTool."L, NoOp, NIL, Unload];
  wh ← Tool.Create[makeSWSProc: MakeSWS, initialState: default,
    clientTransition: ClientTransition, name: "FactorialTool"L, cmSection: "FactorialTool"L ];
}

NoOp: Exec.ExecProc = { };

Unload: Exec.ExecProc = {
  IF Busy[] THEN {
    Exec.OutputProc[h]["Tool is busy. Sorry.\n"];
    RETURN[error] };
  Tool.Destroy[wh];
  Exec.RemoveCommand[h, "FactorialTool."L];
  Done[] };

MakeSWS: Tool.MakeSWSProc = {
  logName: LONG STRING ← [20];
  Tool.UnusedLogName[unused: logName, root: "FactorialTool.log"L];
  data.msgSW ← Tool.MakeMsgSW[window: window];
  data.formSW ← Tool.MakeFormSW[window: window, formProc: MakeForm];
  data.fileSW ← Tool.MakeFileSW[window: window, name: logName] };

MakeForm: FormSW.ClientItemsProcType = {
  OPEN FormSW;
  nItems: CARDINAL = FormItems.LAST.ORD + 1;
  format: ARRAY[0..3) OF Enumerated ← [
    ["hex"L, 0], ["octal"L, 1], ["decimal"L, 2]];
  items ← AllocateItemDescriptor[nItems];
  items[FormItems.number.ORD] ← NumberItem[
    tag: "Number"L, place: [8, line0], signed: FALSE, notNegative: TRUE, value: @data.number];
  items[FormItems.format.ORD] ← EnumeratedItem[tag: "Format"L,
    place: [186, line0], feedback: all, choices: DESCRIPTOR[format], value: @data.format];
  items[FormItems.factorial.ORD] ← CommandItem[
    tag: "Factorial"L, place: [8, line1], proc: Factorial];
  RETURN[items; items, freeDesc: TRUE] };

-- Mainline code
Init[]; -- this gets string out of global frame
}...

```

User-Modified Factorial Tool

```
-- FactorialTool.mesa  
-- Create by FormSWLayoutTool on 16-May-86 10:42. Modified on 16-May-86 10:57
```

DIRECTORY

```
Exec USING [AddCommand, ExecProc, Handle, OutputProc, RemoveCommand],  
Format USING [StringProc],  
FormSW USING [AllocateItemDescriptor, ClientItemsProcType, CommandItem, Enumerated,  
EnumeratedItem, line0, line1, NumberItem, ProcType],  
Heap USING [Create],  
Process USING [Detach],  
Put USING [Text],  
Tool USING [Create, Destroy, MakeFileSW, MakeFormSW, MakeMsgSW, MakeSWsProc, UnusedLogName],  
ToolWindow USING [TransitionProcType],  
Window USING [Handle];
```

```
FactorialTool: MONITOR IMPORTS Exec, FormSW, Heap, Put, Process, Tool = {
```

```
DataHandle: TYPE = LONG POINTER TO Data;  
Data: TYPE = MACHINE DEPENDENT RECORD [  
  msgSW(0): Window.Handle ← NIL,  
  formSW(2): Window.Handle ← NIL,  
  fileSW(4): Window.Handle ← NIL,  
  number(8): CARDINAL ← 0,  
  format(7): FormatType ← decimal];
```

```
FormatType: TYPE = {hex, octal, decimal};  
FormItems: TYPE = {number, format, factorial};  
data: DataHandle ← NIL;  
wh: Window.Handle ← NIL;  
zone: UNCOUNTED_ZONE ← Heap.Create[initial: 4];  
busyBit: BOOLEAN ← FALSE;
```

```
Busy: ENTRY PROCEDURE RETURNS [isBusy: BOOLEAN] = {  
  ENABLE UNWIND => NULL:  
  isBusy ← busyBit;  
  busyBit ← TRUE };  
Done: ENTRY PROCEDURE = {  
  ENABLE UNWIND => NULL:  
  busyBit ← FALSE };
```

```
Write: Format.StringProc = {Put.Text[data.fileSW, s]};  
Msg: Format.StringProc = {Put.Text[data.msgSW, s]};
```

```
ClientTransition: ToolWindow.TransitionProcType = {  
  SELECT TRUE FROM  
    old = inactive => IF data = NIL THEN data ← zone.NEW[Data ← []];  
    new = inactive => IF data # NIL THEN { zone.FREE[@data] };  
  ENDCASE };
```

```
Factorial: FormSW.ProcType = {  
  ENABLE ABORTED => {Done[]; CONTINUE};  
  IF Busy [] THEN { Msg["Tool is busy.\n"]; RETURN};  
  Process.Detach[FORK FactorialInternal[]];
```

```
FactorialInternal: PROCEDURE = {
  ENABLE ABORTED => {Done[]; CONTINUE};
  IF data.number > 12 THEN { --Out of range.
    Put.CR[data.msgSW]; Put.Text[data.msgSW, "Too high, try again."L]
  } ELSE {
    result: LONG CARDINAL ← 1;
    temp: CARDINAL ← data.number;
    WHILE temp > 0 DO result ← result * temp; temp ← temp - 1; ENDLOOP;
    Put.CR[data.fileSW];
    Put.Text[data.fileSW, "The factorial of "];
    Put.Decimal[data.fileSW, data.number];
    Put.Text[data.fileSW, " is "];
    SELECT data.format FROM
      hex => Put.LongNumber[data.fileSW, result, [16]];
      octal => Put.LongNumber[data.fileSW, result, [8]];
      decimal => Put.LongNumber[data.fileSW, result, [10]];
    ENDCASE;
    Put.CR[data.fileSW] };
  Done[] };

Init: PROCEDURE = {
  Exec.AddCommand["FactorialTool.~"L, NoOp, NIL, Unload];
  wh ← Tool.Create[makeSwsProc: MakeSws, initialState: default,
    clientTransition: ClientTransition, name: "FactorialTool"L, cmSection: "FactorialTool"L] };

NoOp: Exec.ExecProc = { };

Unload: Exec.ExecProc = {
  IF Busy[] THEN {
    Exec.OutputProc[h]["Tool is busy. Sorry.\n"L];
    RETURN[error] };
  Tool.Destroy[wh];
  Exec.RemoveCommand[h, "FactorialTool.~"L];
  Done[] };

MakeSws: Tool.MakeSwsProc = {
  logName: LONG STRING ← [20];
  Tool.UnusedLogName[unused: logName, root: "FactorialTool.log"L];
  data.msgSW ← Tool.MakeMsgSW[window: window];
  data.formSW ← Tool.MakeFormSW[window: window, formProc: MakeForm];
  data.fileSW ← Tool.MakeFileSW[window: window, name: logName] };

MakeForm: FormSW.ClientItemsProcType = { OPEN FormSW;
  nItems: CARDINAL = FormItems.LAST.ORD + 1;
  format: ARRAY[0..3] OF Enumerated ← [{"hex"L, 0}, {"octal"L, 1}, {"decimal"L, 2}];
  items ← AllocateItemDescriptor[nItems];
  items[FormItems.number.ORD] ← NumberItem[
    tag: "Number"L, place: [6, line0], signed: FALSE, notNegative: TRUE, value: @data.number];
  items[FormItems.format.ORD] ← EnumeratedItem[tag: "Format"L,
    place: [186, line0], feedback: all, choices: DESCRIPTOR[format], value: @data.format];
  items[FormItems.factorial.ORD] ← CommandItem[tag: "Factorial"L, place: [6, line1], proc: Factorial];
  RETURN[items: items, freeDesc: TRUE] };

Init[]; -- this MainLine code gets the string out of global frame
}...
```

Using the FormSWLayoutTool

Description

The FormSWLayoutTool has three subwindows: a message subwindow, a form subwindow, and a file subwindow. The items in the FormSWLayoutTool form subwindow are as follows:

FormType: {bool, command, enum, longNum, num, source, string, tag}

This is an enumerated item that lists the possible items that you can have in a form subwindow. When you are laying out a form subwindow, you select the type of the item that you want from this enumeration.

bool creates a Mesa BOOLEAN, and is video inverted when TRUE.

command creates a command with an associate procedure that will be called when the command is invoked.

enum creates a tag containing an enumerated list of items. The code generated does not use Mesa enumerated types; rather it creates an ARRAY DESCRIPTOR containing strings corresponding to the names in the enumerated tag, and cardinal numbers that correspond to the ORD of the tag item.

longNum generates a LONG UNSPECIFIED, which you can change to a LONG CARDINAL or LONG INTEGER.

num generates an UNSPECIFIED, which you can change to a CARDINAL or INTEGER.

source is not currently supported.

string creates a LONG STRING and in the code sets it to NIL.

tag does not generate a Mesa variable; it simply creates the named tag and places it on your tool. The tag has no functionality; it is for documentation/information.

Tag: This is where you type the name of the tag for the item that you want to place in your form subwindow. Thus, if you want your new form subwindow to have a command called Fred!, you would put Fred in the Tag: field and select Command from the FormType enumeration.

Zone: This specifies the heap that you want your program to use for storage allocation. If you leave this field blank, the default is to use the systemZone.

AlignX is a boolean that causes columns to be defined by the width of the character '0. If you don't use this option, the default is to define one column per bit on the screen. You should use AlignX to ensure that you have straight columns, since it may be difficult to discern if a column is off by one bit.

Usebox causes the generated tool to have the same dimensions and location as the FormSWLayoutTool. Thus you simply manipulate the layout tool to the size and position you like and your new tool will have the same characteristics. Obviously, this only controls the initial size and position of the tool; the user is free to change the window.

Anyfont causes the layout tool to generate code that will have proportional spacing rather than absolute. This means that the form subwindow will look right regardless of the font that the

user chooses. Otherwise, if the tool is displayed in a large font, the letters may distort and overlap.

Root: is where you specify the name that you want your source file to have. Don't include the .mesa extension; this is added automatically by the tool.

Dolt! causes the layout tool to generate code for the form subwindow.

SetDefaults! allows you to set the defaults for the property sheets of the different form items.

Save! saves the contents of the form subwindow that you are creating in a file named root .by. This can be useful if you are creating a complex tool and want to ensure that a system crash won't destroy your work.

Load! loads a .by file into the layout tool so you can continue work (the .by is automatically appended onto the root.)

Plagiarize! lets you copy a form subwindow from another tool into the layout tool's window. Just invoke **Plagiarize!** and then select the form subwindow that you want to plagiarize. You can then edit the plagiarized window, as described below.

Operation

The layout tool has two modes of operation: initial layout and editing. When there is text in the Tag: field, you are in initial layout; otherwise, you are editing.

When you are in initial layout mode, the mouse pointer becomes a brush (a string of characters that represents the tag). To add an item to your new form subwindow, select the type of the item from the **FormType** enumeration, put the appropriate tag in the Tag: field, move the mouse into the bottom subwindow and click over the desired location. When you are through laying out your new form subwindow, remove the text from the Tag: field and you are ready to edit the form.

In edit mode, you can use the DELETE, MOVE, COPY and PROPS keys to edit the items in your form subwindow. DELETE, MOVE, and COPY have the obvious meanings; PROPS allows you to change the properties of an item. Each form item that you create has associated properties, which you can display by selecting the item and pressing the PROP'S key on your keyboard. You can use this property sheet to change various aspects of the item, such as the name of the Mesa variable that an item represents. When you create an enumerated item, you will have to use the property sheet to set the values that the enumeration can have.

Dynamic Storage Allocation Exercise: Letter Groups

In this exercise, you will 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 files:

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

```
Input: James! Where are you?!
Number of Queues: {four}
Group!

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
```

The tool as it appears when LetterTool.bcd is executed.

When the user invokes **Group!**, the **CommandItem** procedure **Group** (in **LetterControl**) passes the input string and desired the number of queues to procedure **ProcessInput** (in **LetterImpl**). **ProcessInput** calls five procedures: **InitQueues**, **CutUpAlphabet**, **StoreLetters**, **PrintResults**, and **DeallocateQueues**. **InitQueues** creates and initializes the queues; **CutUpAlphabet** determines which characters in the alphabet each queue will handle; **StoreLetters** actually puts the characters into the queues; **PrintResults** (in **LetterControl**) displays the results; and **DeallocateQueues** deallocates the storage that the queues used.

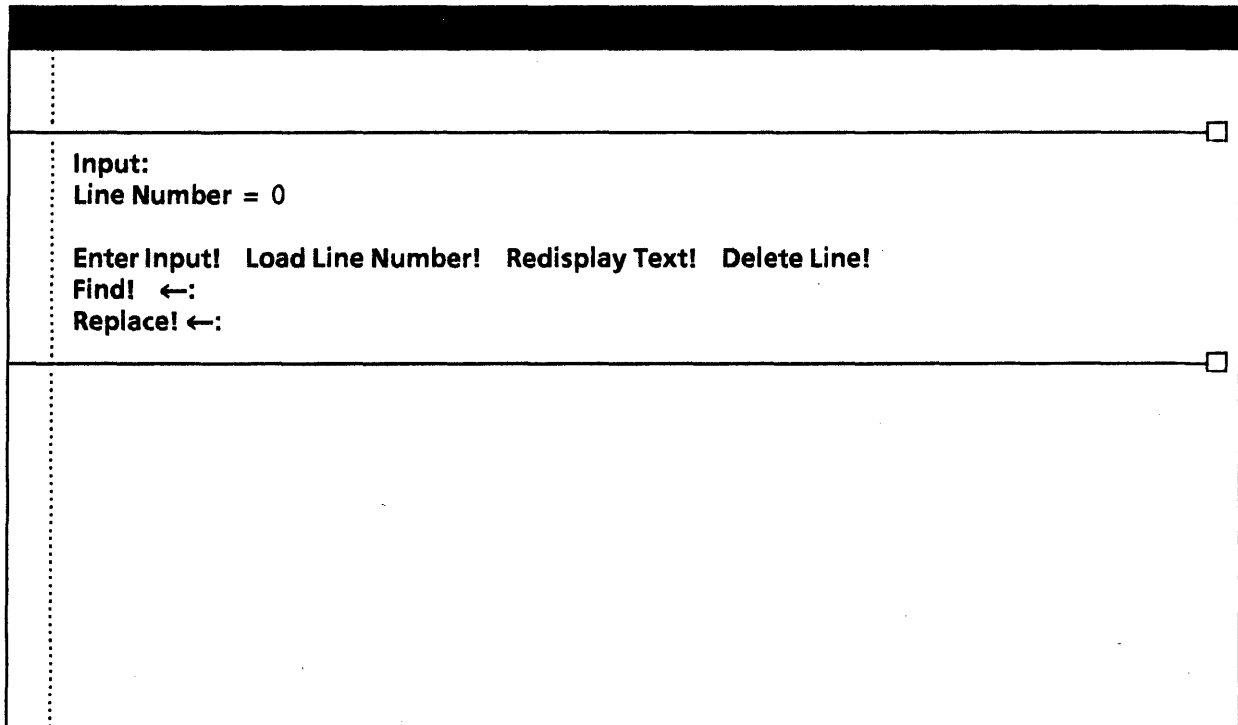
There are two instances where dynamic storage allocation must be considered. First, there is the initial allocation from a heap, where two factors are variable: the number of queues and the size of each queue. Secondly, there is the expansion of a queue when the sequence that represents the queue is 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.

Assignment

Modify the file **LetterImpl.mesa** and fill in the dynamic storage allocation code in the indicated places. The procedures that you need to write are listed at the top of the **LetterImpl.mesa**. You will need to use some of the types that are declared in **LetterDefs.mesa**.

Extra Programming Assignment: Editor

In this exercise, you will modify a line editor that runs in a tool window. To use this tool, you use the **Enter Input!** command to enter input in the line editor, and then use the other commands to change that input.



The line editor currently calls the following string manipulation procedures in the **String** interface:

- String.CopyToNewString**
- String.DeleteSubString**
- String.EqualSubString**
- String.ExpandString**
- String.FreeString**
- String.InsertString**
- String.Replace**

Your assignment is to implement the same procedures through another interface called **String2**. We have provided the interface; you need to 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**

Before you start writing your implementation module, you might want to run the working version (**Editor.bcd**) to get an idea of how it works. Once you are ready to start writing code, you need to do the following:

- 1) Change all **String** references to **String2** in the module **EditorImpl**.
- 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 the implementations for the procedures listed in **String2**.
- 6) Change the configuration **Editor2.config** to reflect the new usage of program modules.
- 7) Test your program.

Note that in order to write the implementations for the **String2** procedures, you will have to read the **String** documentation in the Pilot Programmer's Manual to get an idea of what the procedures are supposed to do, and how a substring works.

DSA Extra Exercise: Tree Traversal Tool

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.

```
Number= 5
Enter Input!      Clear Tree!
PreOrder!      InOrder!      PostOrder!

>>>>>>><<<<<<<<
PreOrder is 7 4 2 5 9 8 12
>>>>>>><<<<<<<<
```

Your assignment is to write the procedures `Init`, `EnterNumber`, and `ClearTree` in the module `TreeTraversalProblem.mesa`. The comments in this module provide a more complete explanation of what you need to do.

You will also need the following modules:

- `TreeTraversalDefs.mesa`
- `TreeTraversalImp.mesa`
- `TreeTraversalTool.config`

Where Does Control Go After a SIGNAL is Caught?

Statement	Frame and Block Where Control is Resumed After a Catch	UNWIND raised ?	
		YES	NO
RESUME	Control is resumed in the frame where the signal was raised, at the point where the signal was raised. View this situation as a return from a procedure call.		X
CONTINUE	Control is resumed in the frame (and block) where the signal is caught, not in the frame where the signal was raised (if these frames are different). <i>Control is given to the statement following the statement to which the catch phrase belongs.</i> If Catch phrase is in a(n): <ol style="list-style-type: none"> 1. Argument list: Go to the statement following the call. 2. BEGIN - END block: Go to the statement following the BEGIN - END block most narrowly enclosing the ENABLE clause. 3. Loop: Go to the "next" iteration if any. 	X	
RETRY	Control is resumed in the frame (and block) where the signal is caught. <i>Control passes to the beginning of the statement to which the catch phrase belongs.</i> If Catch phrase is in a(n): <ol style="list-style-type: none"> 1. Argument list: Execute the call again. 2. BEGIN - END block: Go to the first statement of the BEGIN - END block most narrowly enclosing the ENABLE clause. 3. Loop: Start the current iteration again from the beginning. 	X	
LOOP	Control is resumed in the frame (and block) where the signal is caught. <i>Control passes to the "next" iteration.</i>	X	
EXIT	Control is resumed in the frame (and block) where the signal is caught. <i>Control passes to the first statement outside the loop containing the EXIT statement.</i>	X	
GOTO	Control is resumed in the frame where the signal is caught. <i>Control passes to the EXITS clause of the block containing the catch phrase or to some surrounding block of the block containing the catch phrase.</i> Note that control need not resume in the block where the signal is caught.	X	

Complex Signal Example

Consider the code below and name the statements that will be executed when the following call is made:
Proc1[0];

```
Sig1: ERROR = CODE;  
Sig2: ERROR = CODE;  
Sig3: SIGNAL = CODE;
```

```
Proc1: PROCEDURE[x: CARDINAL] =  
  BEGIN  
    ENABLE  
    . BEGIN  
      Sig1 => GOTO punt;  
      Sig2 => <statement 1>;  
      UNWIND => <statement 2>;  
    END;  
    <statement 3>;  
    <statement 4>;  
    IF TRUE THEN  
      BEGIN  
        ENABLE  
        Sig1 => <statement 5>;  
        <statement 6>;  
        <statement 7>;  
        Proc2[x!]  
        Sig2, Sig3 => <statement 8>;  
        UNWIND => <statement 9>;  
      END;  
      <statement 10>;  
    EXITS  
    punt => <statement 11>;  
  END;
```

→ Catch Phrase 1

→ Catch Phrase 2

→ Catch Phrase 3

```
Proc2: PROCEDURE[x: CARDINAL] =  
  BEGIN  
    Proc3[x!]  
    Sig1 => <statement 12>;  
    Sig2 => <statement 13>;  
    UNWIND => <statement 14>;  
  END;
```

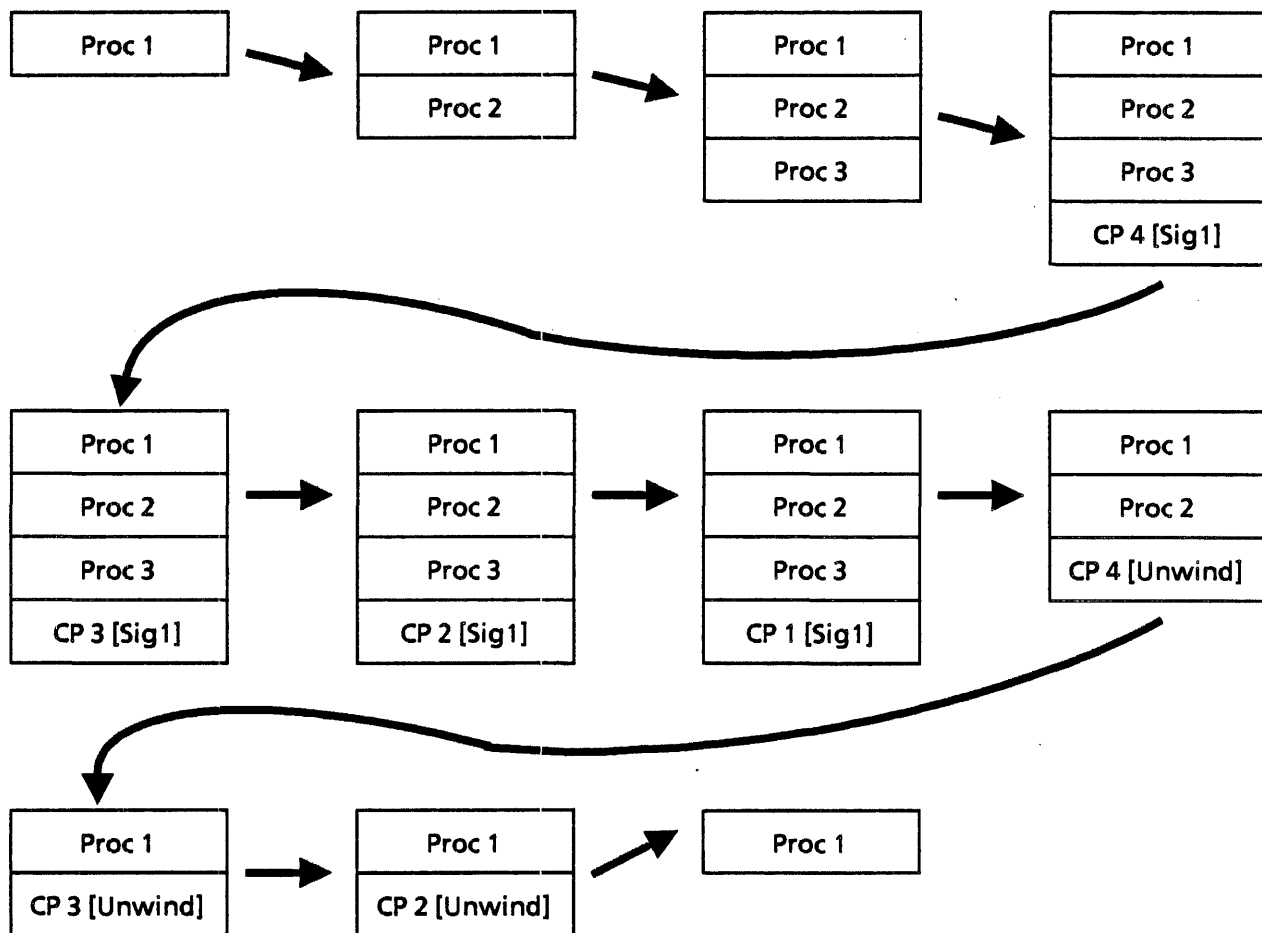
→ Catch Phrase 4

```
Proc3: PROCEDURE [x: CARDINAL] =  
  BEGIN  
    IF x = 0 THEN  
      ERROR Sig1  
    ELSE  
      ERROR Sig2;  
    END;
```

Statements are executed in the following order:

- Statement 3
- Statement 4
- Statement 6
- Statement 7
- Proc2[0]
- Proc3[0]
- Statement 12
- Statement 5
- Statement 14
- Statement 9
- GOTO punt;
- Statement 11

Snapshots of the call-stack when it changes state are shown below (CP = Catch Phrase):



Streams Programming Assignment: Madlibs™

In this programming assignment, you will write a program that will play the game of Madlibs™. In Madlibs™, the user is asked to think of some words at random. Typically he is asked for adjectives, nouns, verbs, etc. Once these words are collected, they are placed into an incomplete story that has specific destinations for adjectives, nouns, verbs, etc. For instance the incomplete story line might look like this:

Dear Mom and Dad:

How are things? My classes aren't hard; they're just a little <adjective>.
I'm having fun though. My teachers are very <adjective> and they
all like to <verb>. Last night, my English teacher showed me how to
<verb>. She also gave me a <noun> and told me to <verb> every day.
My girlfriend is fine. Yesterday we went to <place> and <past tense verb>.
well, I should go. Send <plural noun>.

love, <name>

The finished story might read:

Dear Mom and Dad:

How are things? My classes aren't hard; they're just a little slimy.
I'm having fun though. My teachers are very green and they
all like to roller skate. Last night, my English teacher showed me how to
climb trees. She also gave me a twig and told me to sleep every day.
My girlfriend is fine. Yesterday we went to Hoover Tower and sat.
well, I should go. Send trees.

love, John

In this version of Madlibs™, the incomplete story lines will exist as a local file. Your program will read through the story letter by letter and copy most of the file to a temporary file. Whenever a left bracket, <, is encountered, the program will read the word between the two brackets, and display the word to the tool's message subwindow asking the user to input this type of word. You then think of a word to type into the input field and click over the Read Input! command. Your program will then write the string, contained in the backing store of the input field, to the temporary file. These steps should be repeated until the input file is empty. When the file is empty your program should display the completed story to the tool's file subwindow.

When the tool is run, the user should type in a file name. After a game has started, the tool will query for words:

Type in a verb	<input type="checkbox"/>
Filename: Mad1.txt	Input: ran
Play Madlibs!	Read Input!
	<input type="checkbox"/>

Finally, when the end-of-file has been reached and the new file has been created the text is displayed to the file subwindow.

Filename: Mad1.txt		Input:
Play Madlibs!		Read Input!
Dear Mom and Dad:		
How are things? My classes are ok; sometimes they're a little slimy.		
I'm having fun though. My teachers are very green and they		
all like to roller skate. Last night, my English teacher showed me how to		
climb trees. She also gave me a twig and told me to sleep every day.		
My girlfriend is fine. Yesterday we went to Hoover Tower and sat.		
well, I should go. Send trees.		
Love, John		

Programming Assignment

Your assignment is divided into two parts:

- Part 1: Create a tool with the `FormSWLayoutTool` that contains 4 tags: 2 *strings* and 2 *commands*. Modify the generated code so when commands are invoked the tool will call the procedures defined in `MadlibsDefs.mesa`. The generated code should be placed in the file `MadlibsControl.mesa`.
- Part 2: Modify the file `MadlibsImpl.mesa` which implements the `Play Madlibs!` and `Read Input!` commands. (The two procedures that implement those commands, `PlayMadlibs` and `GetInput`, should be called from the `MadlibsControl` module whenever the user clicks on those commands in the window.) You will have to implement three procedures in `MadlibsImpl` according to the comments that describe the procedures. Bind the 2 program modules `MadlibsImpl` and `MadlibsControl` using the configuration file `Madlibs.config`. There are 3 files with which to test your program; `Mad1.txt`, `Mad2.txt`, and `Mad3.txt`.

Signal Exercises

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

In all code fragments, assume the following declaration:

```
Sig1: SIGNAL = CODE;
```

1. 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>;
2. 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>;
3. 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>;
4. 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>;
5. 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>;
6. Proc1: PROCEDURE =
 BEGIN
 SIGNAL Sig1;
 END;

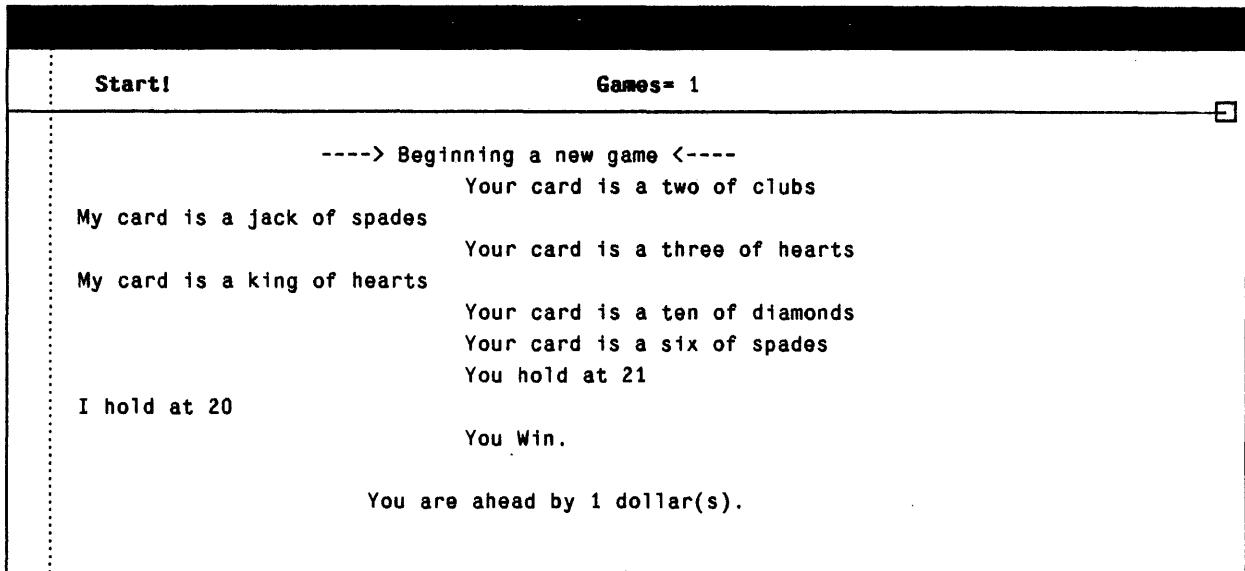
 IF TRUE THEN
 BEGIN
 ENABLE
 Sig1 => RESUME;
 <statement 1>;
 Proc1[!Sig1 => CONTINUE];
 <statement 2>;
 Proc1;
 <statement 3>;
 END;
 <statement 4>;

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

8. 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;

Extra Signals Programming Assignment: Blackjack

In this programming assignment, you will alter a program which 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; both hands will be output to a file sub-window as each card is played. When all of the games are finished, the total number of dollars won will also be output to a file sub-window:



```
Start! Games= 1
-----> Beginning a new game <-----
Your card is a two of clubs
My card is a jack of spades
Your card is a three of hearts
My card is a king of hearts
Your card is a ten of diamonds
Your card is a six of spades
You hold at 21
I hold at 20
You Win.
You are ahead by 1 dollar(s).
```

In this game of blackjack, the player bets 1 dollar on every hand. If he gets blackjack, then he wins 2 dollars. Otherwise if the dealer gets blackjack, the player loses. If the game continues, the player receives hits according a conservative strategy based on his hand, and the dealer's face card. If he busts, he loses. Otherwise, the dealer receives hits until his total is a hard 17 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 **Start!** is clicked, 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.
playerTotal, dealerTotal: CARDINAL;
playerHasAce, dealerHasAce: BOOLEAN;
dealerHole, dealerFace: CardType;

THROUGH [1..gamesToBePlayed] DO
  IntializeDeckForNewGame;
  Put.CR[output]; Put.CR[output];
  Put.Line[output, "          ---->Beginning a new game <----"L];
  [playerTotal,dealerTotal,playerHasAce,dealerHasAce,dealerHole,dealerFace] ← Deal[output];
  IF playerHasAce AND (playerTotal = 11) THEN {
    winnings ← winnings + 2; --Player has Blackjack
    Put.Line[output, "          *** You Have A Blackjack!!!! ***"L];
  }
  LOOP };
```

```
IF dealerHasAce AND (dealerTotal = 11) THEN {
  winnings ← winnings - 1; --Dealer has Blackjack
  Put.Line[output, "*** I Have A Blackjack!!!! ***"L];
  LOOP };
[playerTotal] ← HitPlayer[output, playerHasAce, playerTotal, dealerFace];
IF playerTotal > 21 THEN {
  winnings ← winnings - 1; --Player busted.
  Put.Line[output, "                                You Busted"L];
  LOOP }
ELSE {
  Put.Text[output, "                                You hold at "L];
  Put.LongDecimal[output, playerTotal];
  Put.CR[output] };
dealerTotal ← HitDealer[output, dealerHasAce, dealerTotal];
IF dealerTotal > 21 THEN {
  winnings ← winnings + 1; --Dealer busted.
  Put.Text[output, "I Busted"L];
  LOOP }
ELSE {
  Put.Text[output, "I hold at "L];
  Put.LongDecimal[output, dealerTotal];
  Put.CR[output] };
SELECT playerTotal FROM
  < dealerTotal => {
    winnings ← winnings - 1;
    Put.Line[output, "I Win"L] };
  > dealerTotal => {
    winnings ← winnings + 1;
    Put.Line[output, "                                You Win"L] };
  ENDCASE => Put.Line[output, "                                We're even. Tie Game."L];
ENDLOOP;
Put.CR[output];
IF winnings < 0 THEN Put.Text[output, "                                You are behind by "L]
ELSE Put.Text[output, "                                You are ahead by "L];
Put.LongDecimal[output, ABS[winnings]];
Put.Line[output, " dollar(s)."L];
}; --PlayBlackJack
```

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.
  IF freeCard = 53 THEN [deck, firstCard, freeCard] ← Shuffled[deck, firstCard];
  card ← deck[freeCard];
  freeCard ← freeCard + 1;
}; --NewCard
```

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 instance 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 that will 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 currently not 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.

Assignment

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 (like 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 the assignment might have been completed without using a signal.
2. Signals could have been used to indicate `DealerBlackjack`, `DealerBusted`,... From an efficiency point of view, why isn't this such a great idea?

Signal Exercises Solutions

In all code fragments, assume the following declaration:

Sig1: SIGNAL = CODE;

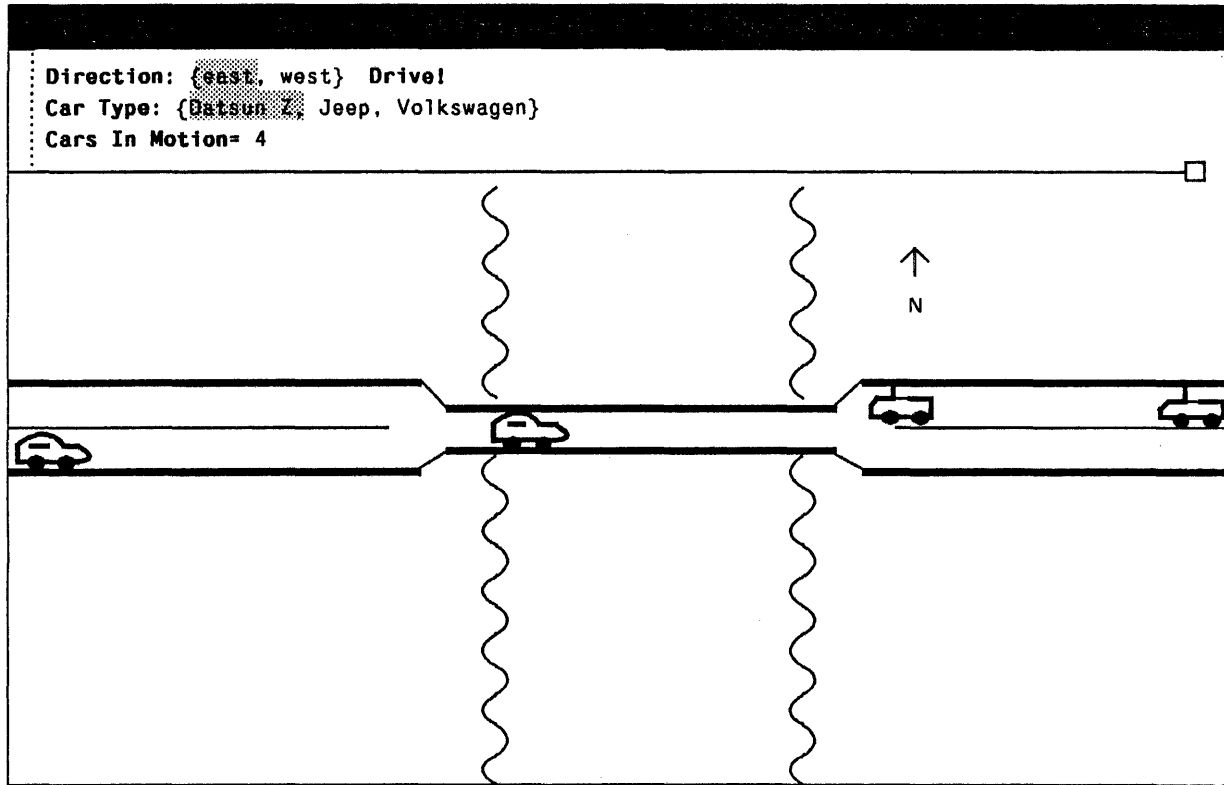
- | | | | |
|--------------------------------------|---|--------------------------------------|---|
| 1. FOR counter: INTEGER IN [1..2] DO | 1 | 4. FOR counter: INTEGER IN [1..2] DO | 1 |
| BEGIN | 1 | ENABLE | 1 |
| ENABLE | 2 | Sig1 => LOOP; | 2 |
| Sig1 => LOOP; | 3 | <statement 1>; | 3 |
| <statement 1>; | 4 | IF counter = 1 THEN SIGNAL Sig1; | 4 |
| IF counter = 1 THEN SIGNAL Sig1; | | <statement 2>; | |
| <statement 2>; | | <statement 3>; | |
| END; | | ENDLOOP; | |
| <statement 3>; | | <statement 4>; | |
| ENDLOOP; | | | |
| <statement 4>; | | | |
| | | | |
| 2. FOR counter: INTEGER IN [1..2] DO | 1 | 5. FOR counter: INTEGER IN [1..2] DO | 1 |
| BEGIN | 3 | ENABLE | 1 |
| ENABLE | 1 | Sig1 => CONTINUE; | 2 |
| Sig1 => CONTINUE; | 2 | <statement 1>; | 3 |
| <statement 1>; | 3 | IF counter = 1 THEN SIGNAL Sig1; | 4 |
| IF counter = 1 THEN SIGNAL Sig1; | 4 | <statement 2>; | |
| <statement 2>; | | <statement 3>; | |
| END; | | ENDLOOP; | |
| <statement 3>; | | <statement 4>; | |
| ENDLOOP; | | | |
| <statement 4>; | | | |
| | | | |
| 3. FOR counter: INTEGER IN [1..2] DO | 1 | 6. Proc1: PROCEDURE = | 1 |
| BEGIN | 4 | BEGIN | 2 |
| ENABLE | | SIGNAL Sig1; | 3 |
| Sig1 => EXIT; | | END; | 4 |
| <statement 1>; | | | |
| IF counter = 1 THEN SIGNAL Sig1; | | IF TRUE THEN | |
| <statement 2>; | | BEGIN | |
| END; | | ENABLE | |
| <statement 3>; | | Sig1 => RESUME; | |
| ENDLOOP; | | <statement 1>; | |
| <statement 4>; | | Proc1[!Sig1 => CONTINUE]; | |
| | | <statement 2>; | |
| | | Proc1; | |
| | | <statement 3>; | |
| | | END; | |
| | | <statement 4>; | |

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

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


Monitors Programming Assignment: One Lane Bridge

In this programming assignment, you need to write part of a program that emulates the traffic on a one lane bridge. As shown in the diagram below, there are two lanes of traffic on each side of a bridge that has only one lane. You will write a monitor to handle the traffic on the bridge (and prevent head-on collisions). Keep in mind that if a car is on the bridge going in some direction (i.e. east), then other cars going in the same direction can follow it. Only the cars going in the opposite direction would have to wait.



When you select a direction (east or west) and type of car (Datsun Z, Jeep, or Volkswagen) and then invoke `Drive!`, a new car of the chosen type going in the chosen direction will be created, and the variable `Cars In Motion` will be incremented. `Cars In Motion` cannot be changed from the window - only from within the program. When you start a car, it will move toward the bridge, then wait, if necessary, until it can cross the bridge, and then continue driving. When it drives past the window, it parks somewhere and `Cars In Motion` will be decremented to reflect that the car is no longer in motion.

The entire user interface has been written for you. This includes the code to create the window and also the code to draw the moving cars. You need to implement the procedure `Bridge.Drive`, which is called when the user invokes `Drive!`. This procedure is defined in the interface `Bridge.mesa`. You should implement this procedure in `BridgeMonitorImpl.mesa`. A template has been provided for you.

Notice that this module is a MONITOR, but the procedure `Drive` is not an entry procedure - it is an external procedure. But, since this is a MONITOR module, you can put all of your procedures for monitoring the bridge in this module, too.

The interface `Bridge.mesa` (shown below) contains many declarations that you will need. Each of the items in this interface are explained below **DO NOT ALTER OR RE-COMPILE THIS INTERFACE**. There are many modules that are dependent upon the particular version of `Bridge.bcd` that is on your disk, and you do not have the source to recompile those other modules.

--Bridge.mesa
-- 16-Jan-86 16:07:48

DIRECTORY

Supervisor USING [SubsystemHandle],
Window USING [Handle];

Bridge: DEFINITIONS = {

DataHandle: TYPE = LONG POINTER TO Data;
Data: TYPE = MACHINE DEPENDENT RECORD[
 msgSW(0): Window.Handle ← NIL,
 formSW(2): Window.Handle ← NIL,
 bridgeSW(4): Window.Handle ← NIL, --the subwindow where the cars are displayed
 wh(6): Window.Handle ← NIL,
 direction(8): Direction ← east, --is mapped to the choice in the formSW
 car(9): CarType ← DatsunZ, --is mapped to the choice in the formSW
 carsInMotion(10): CARDINAL ← 0, --is mapped to the variable in the formSW
 agent(11): Supervisor.SubsystemHandle]; --is used to control deactivation of the tool

Direction: TYPE = {east, west}; --Direction and CarType are enumerated
CarType: TYPE = {DatsunZ, Jeep, Volkswagen}; --types that are used in the data record

BridgeType: TYPE = RECORD[
 carsOnBridge: CARDINAL ← 0, --the number of cars on the bridge at any time
 direction: Direction ← east]; --the direction of the cars on the bridge

Drive: PROCEDURE[instanceData, DataHandle];
BeginDriving: PROCEDURE[sw: Window.Handle, car: CarType, dir: Direction];
CrossBridge: PROCEDURE[sw: Window.Handle, car: CarType, dir: Direction];
ContinueDriving: PROCEDURE[sw: Window.Handle, car: CarType, dir: Direction];
}...

In your monitor, you should create an instance of the record type **BridgeType** to keep track of the number of cars on the bridge at any given time and the direction of those cars. The procedure **Drive** is the procedure that you will implement in **BridgeMonitorImpl.mesa**. The three procedures **BeginDriving**, **CrossBridge**, and **ContinueDriving** are the procedures that you will call when you want to send a car off in a given direction. They have already been implemented for you. These procedures display the moving car along the specified part of the road and return control to their caller when they have completed. So, when you want to start a car driving in a given direction, a call like **BeginDriving[data.bridgeSW, which car, which direction]** (where *which car* and *which direction* have the values of a particular **CarType** and **Direction**) would do the display part for you. When you know that a particular car is ready to cross the bridge, call **CrossBridge[...]**, and when the car has left the bridge, call **ContinueDriving[...]**.

Here is the template for `BridgeMonitorImpl.mesa`:

```
--BridgeMonitorImpl.mesa  
-- 16-Jan-86 16:10:52
```

```
DIRECTORY
```

```
Bridge USING [DataHandle],  
Process USING [MsecToTicks, Pause];
```

```
BridgeMonitorImpl: MONITOR
```

```
IMPORTS Process  
EXPORTS Bridge = {
```

```
data: Bridge.DataHandle ← NIL;
```

```
Drive: PUBLIC PROCEDURE[instanceData: Bridge.DataHandle] = {  
  data ← instanceData;  
  Process.Pause[Process.MsecToTicks[500]];
```

```
-- You need to write the rest --
```

```
}; --Drive
```

```
}...
```

The parameter `instanceData` is a pointer to a record of type `Bridge.Data`. `data` is immediately assigned the value of `instanceData`. Now `data` will always have the most current data and you can access any of the fields in `data` (i.e. `data.direction`) from anywhere within this module. `Process.Pause[...]` will suspend the process for a specified time, in this case a half second. This puts a little time in between cars so that they don't start so close to each other that they appear one on top of another. You need to finish writing this procedure and all other 'help' procedures that you may need.

Every time that you change the value of `CarsInMotion`, you need to redisplay the new value. You can do this with a call to `FormSW.DisplayItem[data.formSW, 3]`. The parameters of `FormSW.DisplayItem` are the window that the item we want to redisplay is in (`data.formSW`) and the number of the item that we want to redisplay. Each item in the `formSW` has a corresponding item number. The item number of `CarsInMotion` is 3.

There is a maximum number of co-existing processes allowed. To prevent problems, you should catch the error `Process.TooManyProcesses`, display a message to the message subwindow, and prevent any new cars from starting (at least until other cars have parked). For information on how to catch the error, refer to the Pilot Programmer's Manual, pp. 2-26 through 2-30.

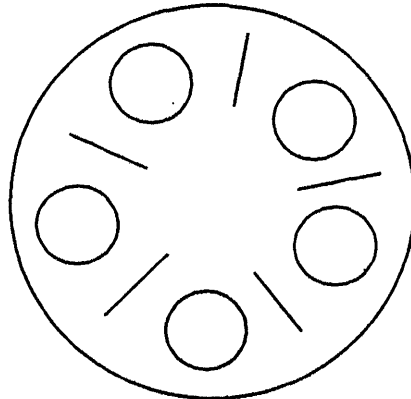
The configuration module, `OneLaneBridge.config`, has been written for you.

Assignment

1. Finish the implementation module `BridgeMonitorImpl.mesa`.
2. Verify your implementation by running the tool.

Extra Monitors Programming Assignment: Dining Philosophers (Hard Version)

Your assignment 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.



The tool has an enumerated item to represent each philosopher:

```
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.
```

Initially, all philosophers are thinking. The user can tell a philosopher to start eating by changing the value of the enumerated from thinking to eating. If the philosopher can eat, then the state will change to eating, and appropriate feedback will appear in the file subwindow. If the philosopher must wait to eat, then the state will change to waiting, and an appropriate message will appear.

Thus, in the program, you need to write a procedure that reacts to changes in the state of the enumerated "philosophers." When the user asks a philosopher to eat, you should check to see if he can eat (see if his chopsticks are available.) If the philosopher can eat, then you need to update data to indicate that his chopsticks are in use, and display a message. Otherwise, the process should wait until it can eat. While the

process is waiting, the state of the philosopher should be "waiting." Note that your implementation is the only way to change the state to "waiting"; the user should not be able to explicitly change the state to or from waiting.

If the user changes the state of a philosopher from eating to thinking, then you should change the data so that his chopsticks are no longer in use, and inform other philosophers that they might be able to eat.

You are on your own for this assignment; we do not provide any of the code for you.