V12

™

MAINSAIL®

Tutorial, Volume I

MAINSAIL® Tutorial, Part I:

A Beginner's Guide to the MAINSAIL Language

24 March 1989

xidak™

Table of Contents

List of Examples

List of Exercises

List of Figures

List of Tables

1. Introduction

This document is designed to teach the MAINSAIL programming language and the use of some of the utilities in the MAINSAIL environment to someone with computer experience. It is assumed that you have at least taken an introductory computer programming course in some other language. This tutorial makes no attempt to explain the overall philosophy behind computers and computer software, but rather concentrates on the specifics of writing programs in MAINSAIL.

1.1. Relevant Documentation

If you have not already done so, you should consult the "MAINSAIL Documentation User's Guide and Master Index" to see what documents are available on MAINSAIL and the MAINSAIL environment. Before beginning this tutorial, you should read (or at least skim) the "MAINSAIL Overview" and the operating-system-specific MAINSAIL user's guide for your operating system.

If you wish to use MAINEDIT, the MAINSAIL text editor, while working through the examples and exercises in this tutorial, you will need to read the "MAINEDIT User's Guide". After reading the first several chapters of the tutorial, you should read the "MAINDEBUG User's Guide" for instructions on the use of the MAINSAIL debugger. You will find MAINDEBUG extremely useful as you write MAINSAIL programs of progressively greater size and complexity.

1.2. Tutorial Overview

Chapters 2 through 20 constitute an introductory self-taught course in MAINSAIL programming. If you have some experience with other ALGOL-style programming languages (e.g., ALGOL, Pascal, Ada), you may want to skim or skip the first few chapters of the tutorial, and any other material that seems familiar. Otherwise, it is strongly recommended that you follow the directions by logging in, entering, compiling, executing, and modifying the sample programs as directed while you read the text.

C programmers will want to read the comparison between MAINSAIL and C in Appendix B of part II of the "MAINSAIL Tutorial".

The tutorial instructions assume you remain connected to the same file directory or catalog while you work your way through the tutorial; if you change directories in the middle, the

MAINSAIL runtime system may be unable to find files produced by the compiler and left on other directories. A discussion of search rules for these files appears in Chapter 16.

Chapter 1 of part II of the "MAINSAIL Tutorial" and subsequent chapters contain suggestions for the more advanced MAINSAIL programmer. They tell how to make your programs more efficient and more portable, and how to take advantage of all the features of the MAINSAIL environment. Programmers who already know MAINSAIL well may prefer to go directly to this part of the tutorial.

Every chapter has an introductory paragraph or two describing its contents. You may find these introductions useful if you are searching for topics of interest.

Answers to most exercises may be found in Appendix A of part II of the "MAINSAIL Tutorial".

1.3. Conventions Used in This Document

Throughout the examples in this document, characters typed by the user (that's you) are underlined. "<eol>" symbolizes the end-of-line key on a terminal keyboard; this key is marked "RETURN" or "ENTER" on most keyboards. In Example 1.3-1, "Prompt:" is written by the computer; the user types "response" and then presses the end-of-line key. "Prompt:" is an example of a "prompt", i.e., something typed by a program to indicate how the user is to respond.

```
Prompt: response<eol>
```

Example 1.3-1. How User Input Is Distinguished

Some figures in this tutorial contain comments separated from the rest of the figure by a line of vertical bars ("|"). Such comments are not entered as part of sample files and do not appear in sample dialogues when programs are actually run; they are present only as an explanation of the main part of the figure. See Example 1.3-2.

Specifications of syntax often contain descriptions enclosed in angle brackets ("<" and ">"). Such descriptions are not typed literally, but are replaced with instances of the things they describe. For example, a specification of the syntax of the address on an envelope might appear as in Example 1.3-3.

```
                     | This is a comment
This is the main     | on the right-hand
part of the figure.  | side of the
                     | figure.
```

Example 1.3-2. How Comments Are Separated from the Main Part of a Figure

```
<name of addressee>
<street number> <street name>
<town or city name>, <state abbreviation>  <zip code>
```

Example 1.3-3. Syntax of a Mailing Address

2. Writing and Running a Simple Program

This chapter uses some simple examples to show the steps involved in writing and executing a simple MAINSAIL program.

2.1. Getting Started

It is assumed that you know how to log into your computer, and that an account has been set up for you. The operating-system-specific user's guide for your operating system tells how to run MAINSAIL on your system. The instructions in the guide may be inaccurate if your MAINSAIL system was installed in a non-standard way; if necessary, consult the person in charge of MAINSAIL on your system for details.

It is also assumed that you know how to create a text file (a file is sometimes also called a "data set") on your system. Creating a text file is usually done by means of a program called a "text editor" (or just an "editor"). A text editor called MAINEDIT is part of the MAINSAIL programming environment, and some examples of its use are given in this document. If MAINEDIT is available on your system, you may wish to learn how to use it by reading the "MAINEDIT User's Guide", which assumes no knowledge of programming in general or of MAINSAIL in particular.

2.2. Invoking a MAINSAIL Module

Log in now and invoke MAINSAIL. You should see a banner and an asterisk ("*") prompt as shown in Figure 2.2-1 (the number following the word "Version" may be different on your system). The banner and prompt are written by a MAINSAIL utility called MAINEX, which is described in detail in the "MAINSAIL Utilities User's Guide".

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*
```

Figure 2.2-1. The MAINEX Banner and Prompt

MAINSAIL programs are composed of entities called "modules". One way to run (or "invoke" or "execute") a MAINSAIL module is to type its name to the MAINEX asterisk prompt. The utility module CONCHK is shipped with every standard MAINSAIL system; type "CONCHK" (in upper or lower case) to the asterisk prompt now. The result should look as in Example 2.2-2. Don't worry about exactly what CONCHK does right at the moment; the important thing is that you have just invoked CONCHK, it has printed the message "No inconsistencies detected.", and then it has finished executing.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*conchk<eol>
No inconsistencies detected.
*
```

Example 2.2-2. Running the Utility Module CONCHK

Note the second asterisk prompt in Example 2.2-2. When a module finishes executing, control ordinarily returns to MAINEX, which prints an asterisk prompt to allow you to run another module. If you wish to return to the operating system prompt, respond to the MAINEX asterisk prompt by typing the end-of-line key, as shown in Example 2.2-3.

```
*<eol>
(the operating system command
 processor prints its own prompt here)
```

Example 2.2-3. Ending a MAINEX Session

In MAINSAIL, unlike many other languages, it is not necessary to return to the operating system prompt after each program is executed (provided each program executed is a MAINSAIL program). Any number of MAINSAIL programs may be run in a row by typing the name of each module to be run to the MAINEX asterisk prompt.

2.3. Compiling and Executing a Simple Source Module

This section describes how to run a small program; the next section analyzes why the program does what it does.

Create a text file called "simple.msl" with the contents shown in Figure 2.3-1. A text file with human-readable program code in it is called a "source file". MAINSAIL source file names customarily end in ".msl"; you may use a different name if you prefer, since the MAINSAIL system makes no assumption about the names of MAINSAIL source files.

```
BEGIN "simple"

INITIAL PROCEDURE;
BEGIN
write(logFile,"Hello, world." & eol);
END;

END "simple"
```

Figure 2.3-1. A Simple MAINSAIL Source File: "simple.msl"

The source file "simple.msl" contains the source (also called "source code"), or textual form, of an entire MAINSAIL module. The source code for a module may be referred to as a "source module", or just as a "module", in which case context must distinguish it from an "object module", which is described below. It is possible for a source file to contain more than one module, or to contain only part of a module, but the case in which a source module is composed of exactly one file is common for small modules.

In order to execute a MAINSAIL module that exists in source form, it must first be "compiled", i.e., translated into a "machine language" form readable by the computer (but not by human beings). Some programming languages (e.g., certain forms of BASIC) do not require a compilation step, but rather execute source code directly (or nearly directly). Such languages are called "interpreted languages"; MAINSAIL is called a "compiled language" because the compilation step is required. A MAINSAIL compilation translates the source module into a machine language file (the object module, usually called an "objmod") of which the name is derived from the name of the source module. When you type the name of a module to the MAINEX asterisk prompt, you are really telling MAINEX to search for the objmod with the file name corresponding to the module name you have just typed (actually, some modules are not contained in individual files, and there are ways of changing the default correspondence between source and object module names, but these are discussed later).

To compile "simple.msl", you must invoke the MAINSAIL compiler, which is itself a MAINSAIL module called COMPIL. COMPIL prints out a herald and a prompt ("compile (? for help):"), then allows you to type the name of the first source file to be compiled. Perform the steps shown in Example 2.3-2.

```
*compil<eol>                               | COMPIL is the
 -----------                               | module to run.


MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): simple.msl<eol> | "simple.msl"
                      ---------------  | (containing source
Opening intmod for $SYS...             | module SIMPLE) is
                                       | the first file to
simple.msl 1                           | compile.
Objmod for SIMPLE on simple-xyz.obj    | "xyz" will be
Intmod for SIMPLE not stored           | replaced by a
                                       | different string
compile (? for help): <eol>            | on your operating
                      -----            | system.
*
```

Example 2.3-2. Compiling "simple.msl" with the MAINSAIL Compiler

Whenever it compiles, the compiler issues the "opening intmod for $SYS..." message; this means it is looking up the standard symbols built into MAINSAIL. It then prints the names and page numbers of all the source files it uses to produce the compiled version of the module, the object module. Since the module SIMPLE is contained entirely within the file "simple.msl", which has only one page, the MAINSAIL compiler prints "simple.msl 1". When it has finished producing the object file, it writes a line showing the name of the module compiled ("SIMPLE") and the name of the objmod file ("simple-xyz.obj" is shown in Example 2.3-2; in real life, the "xyz" part is replaced with some characters that vary from operating system to operating system). If an intmod (another kind of file produced by the compiler) was not requested (see Section 20.3 for a discussion of intmods), the compiler prints that the intmod was not stored. The compiler then returns to the compiler prompt, to which you may enter another source file name. Typing the end-of-line key to the compiler prompt terminates the execution of COMPIL.

To execute (or "run" or "invoke") the resulting object module, you need only type its name to the MAINEX asterisk prompt. Unlike many programming languages, MAINSAIL does not require an explicit "link" or "bind" step following compilation. SIMPLE writes the line "Hello, world.", then exits. See Example 2.3-3.

A module or collection of modules that performs some well-defined task is frequently referred to as a "program". The MAINSAIL compiler and MAINEX do not distinguish between

```
*simple<eol>
Hello, world.
*
```

Example 2.3-3. Executing the Compiled Module SIMPLE

modules that constitute entire programs in and of themselves and those that are viewed as fragments of programs. Indeed, the same module may sometimes be used in a fashion the programmer views as "independent", or "program-like", and sometimes in a "dependent" fashion. This is more fully explained in Chapter 16. Since SIMPLE by itself performs a relatively well-defined task, it may be thought of as a program.

2.4. Errors and the "Error Response:" Prompt

If you mistyped the contents of the file "simple.msl" when you created it, you may have gotten an error message from the compiler. Such an error message ends with the prompt "Error response:". Such error messages are commonly given by the MAINSAIL runtime system and by MAINSAIL utility programs. The correct thing to do when such an error occurs is usually to hit the <eol> key (unless you know of something more appropriate to do, which may be the case in some circumstances, particularly if the error message itself instructs you to do something different). In this case, the MAINSAIL program that issued the error message usually continues, patching up the error as best it can.

You can cause MAINSAIL to issue an error message by trying to compile a file that is not a valid MAINSAIL program. Try compiling a file named "xxx.msl" that contains just the line "xxx". The result should look likc Example 2.4-1.

2.5. An Analysis of the Module SIMPLE

The module SIMPLE has the parts shown in Figure 2.5-1. The extra blank lines and the line break after "logFile," do not matter to the MAINSAIL compiler; the source code in Figure 2.5-1 is the "same" as the one in Figure 2.3-1 in the sense that if the two files were compiled they would produce identical "simple-xyz.obj" files, which means they would perform the same actions when executed. A change in a source module that does not result in a change in the actions performed by its objmod is said "not to alter the meaning" of the module. The MAINSAIL compiler ignores extra blanks and ends of lines, except within identifiers or numbers or between pairs of double quotes. Blanks or ends of lines are required to separate two adjacent identifiers (words in a program), of course.

```
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): xxx.msl
Opening intmod for $SYS...

xxx.msl 1

ERROR: file xxx.msl page 1 line 1 at ##
xxx##
Expected BEGIN
Error Response: <eol>


ERROR: file xxx.msl page 1 line 1 at ##
xxx##
Module END expected here
Error Response: <eol>

Objmod not generated
Intmod not stored

compile (? for help): <eol>
*
```

Example 2.4-1. An Error Message and Response

The line labeled "Initial "BEGIN"" is the first line of SIMPLE. Every source module begins with the word "BEGIN" followed by the module name in double quotes. The word "BEGIN" is a "keyword" or "reserved word"; i.e., it plays a special role in the structure of MAINSAIL source code. The words "INITIAL", "PROCEDURE", and "END" in Figure 2.5-1 are also keywords. MAINSAIL has a number of keywords, which will be introduced in this tutorial as they are encounted. Keywords are customarily written entirely in upper case, but may be written in lower or mixed case.

The module name must be an "identifier", which means that it must consist entirely of letters and digits, and its first character must be a letter (unlike some programming languages, MAINSAIL does not consider the underbar ("_") to be a letter for the purpose of constructing identifiers). Case is not distinguished in identifiers; the words "simple", "write", and "logFile",

```
BEGIN "simple"                    | Initial "BEGIN"
                                  |
INITIAL PROCEDURE;                | Initial procedure header
                                  |
BEGIN                             | Initial procedure body
write(logFile,                    |
    "Hello, world." & eol);       |
END;                              |
                                  |
END "simple"                      | Final "END"
```

Figure 2.5-1. Parts of the Module SIMPLE

which are all identifiers, could be written as "Simple", "WRITE", and "logfile", respectively, without changing the meaning of the module SIMPLE. Non-keyword identifiers are customarily written in lower case, or in a mixture of upper and lower case so as to be easier to read (e.g., "logFile" is written as it is because it is composed of the words "log" and "file").

Certain predefined identifiers (which will be encountered later) begin with a dollar sign ("$") character. You must never create such an identifier in your own programs.

A module name is required to be no longer than six characters. This is a restriction not placed on most identifiers; most identifiers created by the programmer may be as long as the programmer desires, provided that no line of source code exceeds 32,766 characters. Some programming languages ignore trailing characters in identifiers exceeding a certain length, but in MAINSAIL every character in an identifier is significant. For example, the identifiers in Example 2.5-2 are considered to be different identifiers by the MAINSAIL compiler.

```
thisIsAVeryVeryLongIdentifierItIsLongIdentifierNumber1
thisIsAVeryVeryLongIdentifierItIsLongIdentifierNumber2
```

Example 2.5-2. These Identifiers Are Not the Same

The line in Figure 2.5-1 labeled "Initial procedure header" signifies the beginning of what is called the "initial procedure". The initial procedure contains code that is executed when a module is invoked. A module may or may not contain an initial procedure, but if a single module is to be executed as a program, it must contain an initial procedure (otherwise it does nothing when you try to run it). A module may contain at most one initial procedure, although it may contain other sorts of procedures; procedures are described in more detail in Chapter 5.

The "Initial procedure body" is what SIMPLE executes when it is run. The body begins with the keyword "BEGIN" and ends with the keyword "END", followed by a semicolon. Between the "BEGIN" and the "END" is a call to the procedure "write"; the word "write" is followed by a pair of parentheses in which is enclosed a description of what to write and where to write it. "logFile" is where to write what is written; "Hello, world.", followed by an end-of-line ("& eol"), is what is written. "logFile" is usually your terminal; i.e., things written to logFile appear on your terminal screen. Much more will be explained about "write" in subsequent chapters; for now, just accept that you may cause a program to print something to your terminal by surrounding it with double quotes, preceding it with "write(logFile,", and following it with ");". An end-of-line may be added to something in double quotes by means of "& eol".

The line labeled "Final "END"" in Figure 2.5-1 terminates the source module. The module name in quotes after the keyword "END" must be the same identifier as the module name after the initial "BEGIN". Every source module must have a final "END".

2.6. Errors in a Source File

When the MAINSAIL compiler compiles a module, it checks to make sure that the module conforms to the rules for MAINSAIL text. If the module does not conform, the compiler prints an error message to logFile, and waits for you to type a command indicating how to deal with the error. A common response to the compiler error prompt is <eol>, which means to continue the compilation as well as the compiler can. The compiler does not produce an objmod if it encounters errors in the source text. A source file construct containing errors is said to be "illegal".

As an example, alter "simple.msl" to look like the version shown in Figure 2.6-1. The closing parenthesis following "write" has been removed.

```
BEGIN "simple"                          |
                                        |
INITIAL PROCEDURE;                      |
BEGIN                                   |
write(logFile,"Hello, world." & eol;    | The closing
END;                                    | parenthesis is
                                        | missing.i
END "simple"                            |
```

Figure 2.6-1. A Version of "simple.msl" Containing an Error

Attempt to compile the erroneous version of "simple.msl" by following the dialogue shown in Example 2.6-2. The compiler marks the place where it thinks the error is with "##". It also informs you of the page and line numbers in the source file where the error is. Compiler error messages try to be as informative as possible, although the compiler may sometimes misconstrue your intent. Also, an error may confuse the compiler and cause it to issue error messages beyond the point of the original error in source code that you feel is correct. It is sometimes necessary to take compiler error messages with a grain of salt. In this case, the compiler seems to have done pretty well; it says it was expecting to see a ")" in the file, and that is just what is missing.

```
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): simple.msl<eol>
Opening intmod for $SYS...

simple.msl 1

ERROR: file simple.msl page 1 line 5 at ##
write(logFile,"Hello, world." & eol;##
Expected )
Error Response: <eol>

Objmod for SIMPLE not generated
Intmod for SIMPLE not stored

compile (? for help): <eol>
*
```

Example 2.6-2. Compilation Dialogue for a Source File Containing Errors

When a compiler error occurs, you must alter the source file so that it conforms to rules for MAINSAIL code, then retry the compilation (actually, the compiler sometimes allows you to fix up your module while you are compiling it, then continue; but this is a little complicated until you have had some practice with the compiler. If you are really interested, you can look up the details in the "MAINSAIL Compiler User's Guide").

2.7. Exercises

Exercise 2-1.

Is "ABCDEFG" a legal module name? A legal identifier? How about "1X"? "A_B_C"? "abc"?

Exercise 2-2.

Change the name of SIMPLE to be "byeBye", and have it print out "Bye, folks!" instead of "Hello, world.".

Exercise 2-3.

Replace "PROCEDURE" in the file of Figure 2.3-1 with "PORCEDURE". See what error message results when you try to compile the altered file.

3. Data Types, Variables, Constants, and Expressions: Strings and Integers

This chapter introduces constants and variables, which are used to represent and record values in a program. Values of the two data types "string" and "integer" are used as examples. It is shown how variables and constants may be used to construct more complex expressions by means of operators. The declaration of symbolic constants is also covered.

3.1. Strings

A character is represented in a computer as an integral number. The numbers that represent valid characters are constrained by the operating system to fall within a certain range; the most common range is 0 through 255 (which is the range of values assumed by MAINSAIL). The character represented by each number may be discovered by transmitting the number to a terminal or a printer; when the terminal or printer receives the number, it writes the corresponding character. Some characters do not cause a terminal or printer to write anything; such characters are called "non-printing". The range of valid character numbers and the characters these numbers represent are referred to as the operating system's "character set". MAINSAIL runs only on operating systems on which the character set includes representations for the upper- and lowercase letters, digits, spaces, tabs, ends of lines, and certain common punctuation characters. For an exact list of the guaranteed characters, consult the "MAINSAIL Language Manual".

A MAINSAIL string is a series of up to 32,766 characters. The characters may be any character in the valid range, printing or non-printing (there is no rule, as in C and some other programming languages, that a string is terminated by the character represented by the number 0). A string may contain no characters at all, in which case it is called the "null string".

3.2. String Constants and String Constant Macros

The program WRITER of Example 3.2-1 writes five strings to logFile. The first four strings are all enclosed in pairs of double quotes. The fifth string is represented by the identifier "eol", which is predefined by MAINSAIL to be the string that represents an end-of-line. When eol is written to a terminal, the terminal ends the current line and starts a new line.

When WRITER is executed, the five calls to the procedure "write" are executed in sequence. Create a file containing what is shown in Example 3.2-1 and compile it, then execute it (by typing its name to the MAINEX asterisk prompt) to verify that the result looks as in Example 3.2-2.

```
BEGIN "writer"

INITIAL PROCEDURE;
BEGIN
write(logFile,"This ");
write(logFile,"is");
write(logFile," a ");
write(logFile,"sentence.");
write(logFile,eol);
END;

END "writer"
```

Example 3.2-1. A Module That Writes Strings to logFile

```
*writer<eol>
This is a sentence.
*
```

Example 3.2-2. Executing WRITER

Each of the five strings of Example 3.2-1 is a "string constant". A string constant may be represented in a source file by the characters in the string enclosed in double quotes, or by an identifier, as in the case of eol. If a string constant contains double quotes, the quotes inside the string must be written twice. Example 3.2-3 shows some valid string constants. "tab" is predefined by MAINSAIL to represent the character usually associated with the "TAB" key on a terminal; "eop" is the character used to separate pages in a text file (it is this sort of page that the MAINSAIL compiler counts when it prints page numbers during a compilation).

Strings may be joined ("concatenated") by placing an ampersand ("&") between them. The text of the first string followed by the text of the second string is the result of the concatenation; for example:

```
"Hello, " & "there"    =    "Hello, there"
"xxx" & "."            =    "xxx."
"" & "ABC"             =    "ABC"
"Def" & ""             =    "Def"
```

```
"""Good morning,"" said Eva."  | If written to logFile,
"abc"                          | the first string would
"This is a sentence."          | appear as:
eol                            | "Good morning," said Eva.
tab                            |
eop                            |
""                             | This is the null string.
```

Example 3.2-3. Some Valid String Constants

(concatenating the null string with another string doesn't add any characters to the result). The modules of both Example 3.2-4 and Example 3.2-5 write the same thing to logFile as the module of Example 3.2-1. Two strings concatenated with the ampersand operator constitute a "string expression"; a string expression is itself a string, and any string counts as a string expression. If both of the strings in a concatenation are constants, the expression is a "string constant expression".

```
BEGIN "write2"

INITIAL PROCEDURE;
BEGIN
write(logFile,"This " & "is" & " a " & "sentence." & eol);
END;

END "write2"
```

Example 3.2-4. The Use of "&" to Concatenate Strings

The user may define identifiers that represent string constants, just as eol, eop, and tab are predefined by MAINSAIL. Such identifiers are called "macros". The user creates a string constant macro by means of the keyword "DEFINE". A sample use of "DEFINE" is shown in the module of Example 3.2-6, which does the same thing as the module WRITER of Example 3.2-1.

"DEFINE" may appear immediately after the "BEGIN" of the initial procedure (it may appear other places as well, but that is a topic for later). In the macro definition of Example 3.2-6, "DEFINE" is followed by the identifier to be defined, an equals sign ("="), a string constant or

```
BEGIN "write3"

INITIAL PROCEDURE;
BEGIN
write(logFile,"This is a sentence." & eol);
END;

END "write3"
```

Example 3.2-5. A Simplified Version of WRITER

```
BEGIN "write4"

INITIAL PROCEDURE;
BEGIN
DEFINE stringToWrite = "This is a sentence." & eol;

write(logFile,stringToWrite);
END;

END "write4"
```

Example 3.2-6. WRITER with String Macros

string constant expression, and a semicolon. Other forms of macro definition will be discussed as they are encountered.

3.3. String Variables

Create, compile, and execute the module READER shown in Example 3.3-1. The execution should look something like Example 3.3-2.

The line reading "STRING s;" in the module READER of Example 3.3-1 is called a "variable declaration"; specifically, it is a "string variable declaration", since the variable is a string. Unlike a constant, a variable's value is determined at execution time; a string constant's value is known when the MAINSAIL compiler is invoked, so its value is said to be known at

```
BEGIN "reader"                                   | Initial "BEGIN"
                                                 |
INITIAL PROCEDURE;                               |
BEGIN                                            |
STRING s;                                        | String variable
                                                 | declaration
write(logFile,                                   |
    "Type something, end with <eol>: ");         |
read(cmdFile,s);                                 | cmdFile is
                                                 | terminal input
write(logFile,                                   |
    "The string you typed was """                | Note doubled
        & s & """." & eol);                      | double quotes
END;                                             |
                                                 |
END "reader"                                     | Final "END"
```

Example 3.3-1. A Module Containing a String Variable

```
*reader<eol>
Type something, end with <eol>: haute cuisine<eol>
The string you typed was "haute cuisine".
*
```

Example 3.3-2. Execution of the Module READER

"compiletime". Like the identifier "eol", the identifier "s" represents a string; unlike eol, however, s may take on a different value from execution to execution.

The value of s is determined in the line reading "read(cmdFile,s);". "read" is a procedure, like "write". Like "write", "read" is followed by a pair of parentheses enclosing a series of things separated by commas. The first thing in the series tells where to read from (the "source"); the remaining things are variable identifiers (the "destinations"). The values of the variables in the list are set according to what is read from the source. The source in this case is cmdFile, which normally corresponds to your terminal keyboard, just as logFile corresponds to your terminal screen. "read(cmdFile,s);" assigns to s the string that is the next line read from cmdFile (the <eol> you type to terminate the line is discarded by this form of "read"). When you execute READER, READER replies with whatever string you type.

"STRING" is a keyword indicating the "data type" of the identifier that follows it. MAINSAIL permits operations on a number of different data types; it is not convenient to represent all data as text strings.

3.4. Declarations

The lines reading:

```
                         STRING s;
```

in Example 3.3-1 and:

```
     DEFINE stringToWrite = "This is a sentence." & eol;
```

in Example 3.2-6 are examples of "declarations". Every identifier (with an exception noted in Chapter 11) in a MAINSAIL module must be declared before it is used. Each declaration has a "scope", or part of the source text over which the identifier declared may be used; in the case of a procedure, a declaration occurring immediately after the initial "BEGIN" has a scope extending up to the final "END" of the procedure. Scopes are discussed in detail in Section 6.4.

If you remove the line reading "STRING s;" from Example 3.3-1, and try to compile the resulting file (assume it is called "reader.msl"), you will get a compiler error message saying that the identifier s has not been declared. The dialogue looks as shown in Example 3.4-1. Note that an error message may be issued for each occurrence of the undeclared identifier.

3.5. Integer Constants, Variables, and Expressions

MAINSAIL has an "integer" data type, which may be used to represent whole numbers in the range -32,767 to +32,767 (on some machines the range may actually be larger; -32,767 to +32,767 is the "guaranteed range"). Like strings, integers may be constants or variables. An integer constant is represented by a series of digits (the characters in the set {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}) optionally preceded by a minus sign ("-"). Commas are not permitted in integer constants (so "32767" is legal, but "32,767" is not). Some valid integer constants are shown in Example 3.5-1.

Integer expressions may include (among others) the standard arithmetic operators "+", "-", "*" (multiplication), and "DIV" (integer division, which rounds downwards (for positive operands); most computer character sets do not have the standard division symbol (the one that looks like a colon superimposed on a minus sign)). Parentheses may be used in integer expressions to force the operands to be evaluated in a particular order (they may also be used in expressions of other types, including string expressions). "*" and "DIV" subexpressions are usually evaluated first within an expression, left to right; then "+" and "-" subexpressions, left to right. Use

```
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): reader.msl<eol>
Opening intmod for $SYS...

reader.msl 1

ERROR: file reader.msl page 1 line 8 at ##
read(cmdFile,s##);
Undeclared or ambiguous variable S
Error Response: <eol>


ERROR: file reader.msl page 1 line 12 at ##
        & s## & """." & eol);
Undeclared or ambiguous variable S
Error Response: <eol>

Objmod for READER not generated
Intmod for READER not stored

compile (? for help): <eol>
*
```

Example 3.4-1. Error Message Issued for Undeclared Identifier

```
22
-179
0
32767
-10000
```

Example 3.5-1. Some Valid Integer Constants parentheses if you have any doubt about the order of evaluation of an expression, or if you feel that the expression is difficult to read without parentheses.

Examples of integer expressions appear in Example 3.5-2.

```
Suppose a, b, and c are integer macro constants, and
xy and z are integer variables.  Then all of the
following are integer expressions:

1                     |
(1)                   | Same as 1
1 + xy                |
1 + a + z             | Sum of the three integers
z * 2 + a DIV c       | Same as (z * 2) + (a DIV c)
z * ((2 + a) DIV c)   | Not the same as previous expression

Note that xy is a single integer variable, not a product.
```

Example 3.5-2. Some Valid Integer Expressions

"write" may include integer expressions in the list of things to be written, and "read" may include integer variables in the list of of things to be read. "read" may also use a string (instead of cmdFile) as its source, in which case the characters read are removed from the front of the string. Enter, compile, and execute the program shown in Example 3.5-3.

Note that the first "read(cmdFile,s)" sets the value of s; the following "read(s,i)" changes the value of s by removing the part of s that represents an integer. The changed value of s is not used before the value of s is set again by another "read(cmdFile,s)" and changed again by another "read(s,i)".

3.6. Strong Typing

MAINSAIL does not permit an integer operation to be performed on strings or vice versa. The operator "&" is strictly a string operator; both of its operands must be strings, and its result is a string. The operators "+", "-", "*", and "DIV" may all be integer operators (they may operate on other data types as well, as discussed in Chapter 8); when their operands are both integers, their result type is an integer. Example 3.6-1 shows some expressions that are legal and some that are illegal.

```
BEGIN "iVars"

INITIAL PROCEDURE;
BEGIN
INTEGER i,j;
STRING s;

write(logFile,"Type the first number: ");
read(cmdFile,s); read(s,i);
write(logFile,"Type the second number: ");
read(cmdFile,s); read(s,j);
write(logFile,"The sum of the two numbers is ",i + j,
    eol & "The quotient of the two numbers is ",i DIV j,
    eol);
END;

END "iVars"
```

Example 3.5-3. A Program Using Integer Variables

The rules that prevent the mixing of different data types in expressions are referred to as "strong typing". This is an attribute of MAINSAIL, although some other languages (like C and many dialects of BASIC) are not so rigorous about mixing data types.

```
Assume a, b, and c are integer variables or constants, and
that s, t, and u are string variables or constants.  Then
the following are legal integer expressions:

    1
    1 + (a * c)
    (-4) * (-5)

The following are legal string expressions:

    "Time"
    ("Time")
    "Time " & (s & u)
    "Time " & s & u

The following are illegal because the types of the
operands and/or operators don't match:

    "Five " + 4
    "5" * b
    22 & 6
    ("S" & t) & c
```

Example 3.6-1. Legal and Illegal Expressions

3.7. Exercises

Exercise 3-1.

Replace "xxx" in Figure 3.7-1 with a string constant expression so that the module writes the same thing to logFile as the module WRITE4 of Example 3.2-6. Do not use string quotes in the text that replaces "xxx".

```
BEGIN "writeX"

INITIAL PROCEDURE;
BEGIN
DEFINE str1 = "This is a sentence.";
DEFINE str2 = eol;

write(logFile,xxx);
END;

END "writeX"
```

Figure 3.7-1. Replace "xxx" to Finish the Module

Exercise 3-2.

Write a program that reads in three integers, then writes out the first one minus two, the second times the third, and the third minus the first.

Exercise 3-3.

Which of the following expressions in Figure 3.7-2 are legal, and which illegal? What are the types of the legal expressions?

```
(((5)))
* 62
"Time " & " is " & " money."
eol + 4
tab & eop & eop
```

Figure 3.7-2. Which Are Legal Expressions?

4. Iterative, Assignment, Begin, and If Statements, and the Boolean Data Type

This chapter introduces statements, which are the smallest units of action in a MAINSAIL program. A third data type, boolean, is also introduced. The boolean data type is used in a variety of statements to direct the flow of control (sequence of actions) within a program.

4.1. Statements

The actions of a program are prescribed by "statements". In each example module so far, the text of the initial procedure following the declarations up to the terminating "END" of the procedure has consisted entirely of statements. The statements are separated from each other by semicolons.

The only statements encountered so far have been calls to the procedures "read" and "write"; these are instances of a kind of statement called the "Procedure Statement". MAINSAIL provides a variety of statement forms, of which the Procedure Statement is only one.

4.2. Iterative Statements with FOR-Clauses

An Iterative Statement allows another statement to be executed repeatedly until some condition is satisfied. One form of Iterative Statement is used in the program SQUARE of Example 4.2-1; enter, compile, and execute this program. The execution may look something like Example 4.2-2.

```
*square<eol>
How many squares do you want? 5<eol>
The square of 1 is 1.
The square of 2 is 4.
The square of 3 is 9.
The square of 4 is 16.
The square of 5 is 25.
*
```

Example 4.2-2. Execution of the Module SQUARE

```
BEGIN "square"

INITIAL PROCEDURE;
BEGIN
INTEGER i,j;
STRING s;

write(logFile,"How many squares do you want? ");
read(cmdFile,s); read(s,i);
FOR j := 1 UPTO i DO
    write(logFile,"The square of ",j," is ",j * j,
        "." & eol);
END;

END "square"
```

Example 4.2-1. A Program Using an Iterative Statement with a FOR-Clause

The Iterative Statement with a FOR-clause of Example 4.2-1 causes the call to "write" to be executed i times, where i is the number typed at execution time. On each execution of the loop body, the "iterative variable" j is incremented by one; it starts at one, and the Iterative Statement terminates when j becomes as large as i. The general format of an Iterative Statement with a FOR-clause is:

```
FOR <integer iterative variable> := <lower limit> UPTO
  <upper limit> DO <statement to be repeated>
```

or, if the integer variable is to be decremented by one instead of incremented by one on each execution:

```
FOR <integer iterative variable> := <upper limit> DOWNTO
  <lower limit> DO <statement to be repeated>
```

The statement to be repeated (also called the "iterated statement") is executed the number of times given by:

```
(<upper limit> - <lower limit>) + 1
```

unless the upper limit is less than the lower limit, in which case the iterated statement is skipped entirely.

MAINSAIL's FOR-clause does not provide increments and decrements other than one.

4.3. The Assignment Statement

Until now the only means encountered to assign a value to a variable is to call the system procedure "read" or to use the variable as an iteration variable in a FOR-clause. The value of a variable may be set directly to the value of an expression by means of the Assignment Statement. The form of an Assignment Statement is:

```
<variable> := <expression>
```

The variable's value is changed to be the value of the expression. The variable and the expression must be of the same data type; e.g., a string expression may be assigned only to a string variable, an integer expression only to an integer variable. Example 4.3-1 shows the use of the Assignment Statement. Enter, compile, and execute this module, which computes triangular numbers (the Nth triangular number is the sum of the integers 1 through N).

```
BEGIN "triang"

INITIAL PROCEDURE;
BEGIN
INTEGER i,count,sum;
STRING s;

write(logFile,"Which triangular number do you want? ");
read(cmdFile,s); read(s,count);

sum := 0;
FOR i := 1 UPTO count DO sum := sum + i;

write(logFile,"The ",count,"th triangular number is ",
    sum,eol);
END;

END "triang"
```

Example 4.3-1. A Module Using the Assignment Statement

Example 4.3-2 shows a module that asks for a number N, then prints out two-to-the-Nth-power x's. If you execute this module, be careful not to give too large a value for N; MAINSAIL may not be able to create a string longer than 32,766 characters. You will get an error message from the MAINSAIL runtime system if it attempts to create a string that is too long for it to handle.

```
BEGIN "bigStr"

INITIAL PROCEDURE;
BEGIN
INTEGER i,count;
STRING s;

write(logFile,"N = "); read(cmdFile,s); read(s,count);
s := "x";
FOR i := 1 UPTO count DO s := s & s;
write(logFile,s,eol);
END;

END "bigStr"
```

Example 4.3-2. The Assignment Statement with Strings

4.4. The Begin Statement

The Begin Statement allows several statements to be grouped together and treated as a single statement. This is useful in places where the rules of MAINSAIL syntax require a single statement, but you wish to perform a series of actions. The Begin Statement consists of the keyword "BEGIN" followed by a series of statements separated by semicolons followed by the keyword "END".

For example, the keyword "DO" in an Iterative Statement must be followed by a single statement (the iterated statement). The program of Example 4.4-1 uses Begin Statements as iterated statements to cause a series of statements to be executed repeatedly. Before you enter, compile, and execute this program, take a look at it. What does it do?

A string constant may follow the initial "BEGIN" of a Begin Statement, in which case the same string constant must also follow the final "END" of the Begin Statement. The MAINSAIL compiler issues a warning message if the string following an "END" differs from the string following the corresponding "BEGIN" (the distinction between upper and lower case is ignored in this check). This helps to catch mismatched "BEGIN"-"END" pairs, and helps to make a program more legible. Naming a Begin Statement does not have any effect on the actions it performs; it is mainly a convenience for the human reader, and is entirely optional.

```
BEGIN "delta"

INITIAL PROCEDURE;
BEGIN
INTEGER i,j,count;
STRING s;

write(logFile,"Height: "); read(cmdFile,s); read(s,count);
FOR i := 1 UPTO count DO
    BEGIN
    FOR j := 1 UPTO i DO write(logFile,"**");
    write(logFile,eol);
    END;
FOR i := count - 1 DOWNTO 1 DO
    BEGIN
    FOR j := 1 UPTO i DO write(logFile,"**");
    write(logFile,eol);
    END;
END;

END "delta"
```

Example 4.4-1. The Use of the Begin Statement

String constants may also follow the initial "BEGIN" and final "END" of a procedure, in which case they must match (except for case); see Example 4.6-1. Certain other statement forms may be named; the rules may be found in the "MAINSAIL Language Manual".

Unlike some other languages derived from ALGOL, MAINSAIL does not allow declarations in a Begin Statement.

4.5. The Boolean Data Type

The MAINSAIL data type boolean is designated by the keyword "BOOLEAN". It has only two different values, true and false. The constants of the data type boolean are the keywords "TRUE" and "FALSE".

A variety of operators produce boolean values. One of the most common is the equals sign ("="). The format is:

```
<expression one> = <expression two>
```

Expression one must be of the same data type as expression two. If expression one has the same value as expression two, the result is "TRUE"; otherwise, the result is "FALSE". Other comparison operators are "NEQ" (not equal), "<" (less than), ">" (greater than), "LEQ" (less than or equal), and "GEQ" (greater than or equal).

The operators "AND" and "OR" may each take two boolean operands (or sometimes operands of other data types, as described in Section 8.5), and produce a boolean result. The result of an "AND" operation is true if and only if both of its operands are true; the result of an "OR" operation is true if either one of its operands is true. "NOT" takes a single boolean operand; its result is false if its operand is true, and true if its operand is false.

Boolean expressions are frequently used in the If Statement and in certain forms of the Iterative Statement. Examples of the boolean data type appear throughout the remainder of this tutorial.

4.6. The If Statement

The simpler form of an If Statement is:

```
IF <expression> THEN <statement>
```

The expression is frequently of the data type boolean; other data types are possible, as explained in Section 8.2. For the moment only If Statements governed by boolean expressions will be considered. If the boolean expression is true, the statement is executed; if it is false, the statement is skipped. Example 4.6-1 shows the use of this form of If Statement. The execution of the module shown looks like Example 4.6-2. What happens if you answer something other than "yes" or "no"?

```
*iffy<eol>
Say hello (yes or no)? yes<eol>
Hello.
*iffy<eol>
Say hello (yes or no)? no<eol>
*
```

Example 4.6-2. Execution of IFFY

The more general form of an If Statement is:

```
BEGIN "iffy"

INITIAL PROCEDURE;
BEGIN "If Statement example"
STRING s;

write(logFile,"Say hello (yes or no)? "); read(cmdFile,s);
IF s = "yes" OR s = "YES" THEN
    write(logFile,"Hello." & eol);
END "If Statement example";

END "iffy"
```

Example 4.6-1. The Use of the If Statement

```
IF <expression> THEN <statement one>
  ELSE <statement two>
```

If the expression is true, statement one is executed and statement two skipped; if it is false, statement two is executed and statement one skipped.

Example 4.6-3 asks for the values of two boolean variables, then computes the values of some logical expressions containing the variables. Note the positioning of the parentheses in the statements that assign values to the variables a and b. In this example, the parentheses are redundant (they are present to make the expressions easier to understand). The precedence of the operators ":=", "=", and "OR" is described in the "MAINSAIL Language Manual", and also discussed in Sections 8.5 and 10.3.

4.7. Contracted and Abbreviated Forms

If Statements may be nested; i.e., an If Statement may follow the "THEN" or "ELSE" of another If Statement. An "ELSE" is matched with the innermost unmatched "IF"; for example, a statement of the form:

```
IF a THEN IF b THEN c ELSE d
```

is equivalent to:

```
IF a THEN
  BEGIN IF b THEN c ELSE d END
BEGIN "logic"                        |

INITIAL PROCEDURE;                   |
BEGIN                                |
BOOLEAN a,b,bo;                      |
STRING s;                            |

write(logFile,"A (T or F): ");       |
read(cmdFile,s);                     |
a := (s = "T" OR s = "t");           | A becomes TRUE if "T" or
write(logFile,"B (T or F): ");       | "t" was typed;
read(cmdFile,s);                     |
b := (s = "T" OR s = "t");           | B likewise.

bo := a OR b;                        | Another example of
write(logFile,"A OR B is ");         | boolean assignment
IF bo THEN                           |
    write(logFile,"TRUE")            |
ELSE write(logFile,"FALSE");         |
write(logFile,eol);                  |

bo := NOT bo;                        | This could also be
                                     | written "IF bo THEN
                                     |     bo := FALSE
                                     | ELSE bo := TRUE",
                                     | but the form shown is
                                     | simpler
write(logFile,"A NOR B is ");        |
IF bo THEN                           | This line could be
                                     | written (redundantly) as
                                     | "IF bo = TRUE THEN"
    write(logFile,"TRUE")            |
ELSE write(logFile,"FALSE");         |
write(logFile,eol);                  |
```

Example 4.6-3. Boolean Variables and the If Statement (continued)

```
write(logFile,"NOT A is ");       |
IF NOT a THEN                     |
    write(logFile,"TRUE")         |
ELSE write(logFile,"FALSE");      |
write(logFile,eol);               |
END;                              |

END "logic"                       |
```

Example 4.6-3. Boolean Variables and the If Statement (end)

rather than to:

```
IF a THEN
  BEGIN IF b THEN c END
ELSE d
```

If Statements following an "ELSE" may be used to process a series of choices. Since the sequence "ELSE IF" is common in MAINSAIL programs, MAINSAIL provides the abbreviation "EF". Other abbreviations are "THENB" for "THEN BEGIN", "EL" for "ELSE", "EB" for "ELSE BEGIN", and "DOB" for "DO BEGIN". The abbreviations are completely equivalent to their longer forms. The use of some abbreviations is shown in Example 4.7-1. Note the use of the integer macro constants "red", "yellow", and "blue".

4.8. Comments

It is frequently important that programs be readable to people other than the original programmer. Aside from using names for constants, variables, and procedures that are suggestive of the functions they perform, the most important means of making program text readable is the use of comments.

MAINSAIL comments begin with the character "#" and are terminated by the end of a line. A comment may appear anywhere except within string quotes, where the character "#" is treated as a part of the string. The MAINSAIL compiler ignores the text in a comment; comments are intended for the human reader only.

Examples of MAINSAIL text throughout the remainder of this document are liberally commented. You are encouraged to follow suit in your own programs. It is an art to write comments that are both concise and informative.

```
BEGIN "colors"

INITIAL PROCEDURE;
BEGIN
DEFINE red = 1;
DEFINE yellow = 2;
DEFINE blue = 4;

INTEGER color1,color2,totalColor;
STRING s;

write(logFile,"First paint color: "); read(cmdFile,s);
IF s = "red" THEN color1 := red
EF s = "yellow" THEN color1 := yellow
EF s = "blue" THEN color1 := blue
EB  write(logFile,s,": unknown color, red assumed" & eol);
    color1 := red END;

write(logFile,"Second paint color: "); read(cmdFile,s);
IF s = "red" THEN color2 := red
EF s = "yellow" THEN color2 := yellow
EF s = "blue" THEN color2 := blue
EB  write(logFile,s,": unknown color, red assumed" & eol);
    color2 := red END;

totalColor := color1 + color2;

write(logFile,"By mixing the two you get ");
IF totalColor = red + red THEN write(logFile,"red")
EF totalColor = red + yellow THEN write(logFile,"orange")
EF totalColor = yellow + yellow THEN
    write(logFile,"yellow")
EF totalColor = red + blue THEN write(logFile,"purple")
EF totalColor = yellow + blue THEN write(logFile,"green")
EF totalColor = blue + blue THEN write(logFile,"blue");
write(logFile,eol);

END;

END "colors"
```

Example 4.7-1. Nested If Statements and Abbreviated Forms

4.9. Other Forms of the Iterative Statement

The only form of the Iterative Statement encountered so far is the form with a FOR-clause. The FOR-clause is actually one of three types of optional clauses that may accompany an Iterative Statement. The other two clauses are the WHILE-clause and the UNTIL-clause; each uses an expression, usually of type boolean (rather than an integer counter like the FOR-clause), to govern execution of the iterated statement.

The three clauses may all be present in the same Iterative Statement, or all three may be absent. The form of an Iterative Statement with all three clauses is:

```
FOR <integer variable> := <integer expression one>
  UPTO/DOWNTO <integer expression two>
WHILE <expression three>
DO <statement>
UNTIL <expression four>
```

The order is important; first the FOR-clause, then the WHILE-clause, then the keyword "DO", the iterated statement, and finally the UNTIL-clause.

The simplest form of the Iterative Statement (with no clauses) is:

```
DO <statement>
```

This form of the Iterative Statement repeatedly executes the iterated statement until something in the statement causes it to terminate, usually a Done Statement, Return Statement, or an exception (all of which are described later).

The WHILE-clause tests a condition (usually a boolean expression; other types of expressions are possible and are explained in Section 8.2) before each execution of the iterated statement. If the condition is true, the statement is executed; otherwise, the Iterated Statement terminates.

The UNTIL-clause tests a condition after each execution of the iterated statement; if the condition is true, the Iterated Statement terminates. If it is false, the FOR-clause and WHILE-clause tests, if they are present, are performed to determine whether another repetition of the Iterated Statement is to be made.

An Iterative Statement containing only an UNTIL-clause will always execute its iterated statement at least once, since the test does not occur until after the execution of the statement; an Iterative Statement containing only a WHILE-clause may skip its iterated statement entirely if the condition of the WHILE-clause is false when the Iterative Statement is reached.

The use of WHILE- and UNTIL-clauses is shown in Examples 4.9-1 and 4.9-2.

```
BEGIN "sqrts"

# This program finds the integer square root of a number,
# i.e., the largest integer the square of which is less
# than or equal to the given number.  The given number
# is assumed to be nonnegative.

INITIAL PROCEDURE;
BEGIN
INTEGER num,i;
STRING s;

write(logFile,"Number: "); read(cmdFile,s); read(s,num);
i := 0;
WHILE i * i < num DO i := i + 1;

# Now either i * i = num or i is too big by one, and
# i * i > num.
IF i * i > num THEN i := i - 1;
write(logFile,"The integer square root of ",num," is ",
    i,eol);
END;

END "sqrts"
```

Example 4.9-1. Finding an Integer Square Root

```
BEGIN "loops"

# This program adds a series of integers entered from
# cmdFile.

INITIAL PROCEDURE;
BEGIN
INTEGER i,sum;
STRING s,ss;

sum := 0;
DOB write(logFile,"Next number (<eol> to stop): ");
    read(cmdFile,s); ss := s;
    IF s NEQ "" THENB
        read(s,i); sum := sum + i END END UNTIL ss = "";
write(logFile,"Sum = ",sum,eol);
END;

END "loops"
```

Example 4.9-2. Adding Up Some Numbers

4.10. Exercises

Exercise 4-1.

Write a program which asks for the height of a triangle, like Example 4.4-1, but which prints the triangle oriented like that depicted in Example 4.10-1, which shows what your triangle should look like when 5 is given for the height.

```
    *
   ***
  *****
 *******
*********
```

Example 4.10-1. A Sample Triangle

Exercise 4-2.

Write a program that repeatedly asks a question until the user answers with either "yes" or "no". If some other answer (e.g., "maybe") is given, the program should inform the user that only "yes" and "no" are acceptable answers, then ask the question again. See Example 4.6-1.

5. Introduction to Procedures; the Procedure and Return Statements

This chapter introduces procedures and the Return Statement, which terminates the execution of a procedure. Procedures provide a convenient means of "encapsulating" a series of actions so that a frequently used sequence of statements does not need to be written out in full each time it is used. Macros provide a slightly different, and less frequently used, means of achieving the same end; they are described in Chapter 13. Procedures may also be used to calculate a value, like the "functions" of Pascal or C.

5.1. Some Sample Procedures

Some of the example programs encountered so far contain repeated sections of similar code. Using procedures, a piece of code can be written only once, then invoked as needed from different places in a program. A detailed explanation of the program examples in this section is deferred until the next section, but the overall purpose of a procedure should be apparent when the programs with procedures are compared with the versions without procedures.

The programs of Examples 5.1-1 and 5.1-2 produce the same output. Compare the two and see if you can get a feeling for what is going on.

```
BEGIN "noProc"

# Write "Hello!" ten times.

INITIAL PROCEDURE;
BEGIN
INTEGER i;

FOR i := 1 UPTO 10 DO
    write(logFile,"Hello!" & eol);
END;

END "noProc"
```

Example 5.1-1. A Program with Only an Initial Procedure

```
BEGIN "procs"

# Write "Hello!" ten times.

PROCEDURE writeHello; # Procedure header for "writeHello"
BEGIN                              # These three lines are
write(logFile,"Hello!" & eol);     # the procedure body for
END;                               # "writeHello".



INITIAL PROCEDURE;
BEGIN
INTEGER i;

FOR i := 1 UPTO 10 DO
    writeHello; # This is a call to "writeHello".
END;

END "procs"
```

Example 5.1-2. A Program with Two Procedures

Some of the example programs encountered so far could benefit from the introduction of procedures. For example, Example 4.4-1 can be rewritten as shown in Example 5.1-3. The procedure header of "writeAsterisks", unlike that of "writeHello" of Example 5.1-2, has a parenthesized "parameter list".

Example 4.6-3 has been rewritten as Example 5.1-4. The original program has been altered to be more sophisticated when prompting for an input. Note how much simpler and easier to read the initial procedure has become in both Examples 5.1-3 and 5.1-4; the newly added procedures are also compact and easy to follow. Procedures enhance the readability of a program by breaking down complicated logic into easily understood pieces.

Enter, compile, and execute the programs of Examples 5.1-3 and 5.1-4.

5.2. Procedure Declarations, Calls, Parameters, and Arguments

The form of the header of a procedure declaration is:

```
BEGIN "delta2"

PROCEDURE writeAsterisks (INTEGER howMany);
# Writes (howMany * 2) asterisks, then an end-of-line.
BEGIN
INTEGER i;

FOR i := 1 UPTO howMany DO write(logFile,"**");
write(logFile,eol);
END;



INITIAL PROCEDURE;
BEGIN
INTEGER i,count;
STRING s;

write(logFile,"Height: "); read(cmdFile,s); read(s,count);
FOR i := 1 UPTO count DO writeAsterisks(i);
FOR i := count - 1 DOWNTO 1 DO writeAsterisks(i);
END;

END "delta2"
```

Example 5.1-3. The Use of a Procedure

```
    <qualifiers, if any> <data type, if any> PROCEDURE
      <procedure name> <parameter declaration list, if any>
```

All of the procedures of Examples 5.1-3 and 5.1-4 (except for the initial procedures) have a parameter declaration list, which is the parenthesized list follow the procedure name; none of them has a data type (procedures declared with a data type are explained in Section 5.4). One procedure qualifier, "INITIAL", is already familiar; others are described in Sections 7.2 and 7.5.

A procedure call may be used as a statement; such a statement is called a Procedure Statement. Calls to "read" and "write" are examples of Procedure Statements. In Example 5.1-4, the statements in the initial procedure that begin with "getValue" and "writeBoolean" are also Procedure Statements; each of them calls the procedure named. When a procedure is called, the statements in it are executed, and then control resumes immediately after the point at which the procedure was called.

```
BEGIN "logic2"

PROCEDURE getValue (STRING name;
                    PRODUCES BOOLEAN bo);
# Note that the "bo" in this procedure is completely
# distinct from the variable "bo" in the initial
# procedure.
BEGIN
BOOLEAN goodAnswerGiven;
STRING s;

# Insist on seeing an answer of either "T" or "F".
DOB write(logFile,name," (T or F): "); read(cmdFile,s);
    goodAnswerGiven := TRUE; # Assume for the moment
    IF s = "T" OR s = "t" THEN bo := TRUE
    EF s = "F" OR s = "f" THEN bo := FALSE
    EB  goodAnswerGiven := FALSE;
        write(logFile,"""",s,""" is not a valid answer."
            & eol) END END UNTIL goodAnswerGiven;
END;



PROCEDURE writeBoolean (BOOLEAN value);
BEGIN
IF value THEN write(logFile,"TRUE" & eol)
EL write(logFile,"FALSE" & eol);
END;



INITIAL PROCEDURE;
BEGIN
BOOLEAN a,b,bo;

getValue("A",a); getValue("B",b);

bo := a OR b;
write(logFile,"A OR B is "); writeBoolean(bo);

bo := NOT bo;
write(logFile,"A NOR B is "); writeBoolean(bo);
```

Example 5.1-4. More Procedures (continued)

```
write(logFile,"NOT A is "); writeBoolean(NOT a);
END;

END "logic2"
```

Example 5.1-4. More Procedures (end)

Parameters provide a way for procedures to consume and/or produce information. A parameter list consists of a series of variable declarations enclosed in parentheses. Each declaration is optionally preceded by one or more keywords called "parameter qualifiers", which describe how the parameter is used.

An argument is to a procedure call as a parameter is to a procedure declaration. Arguments are listed in parentheses following the name of the procedure called. Except in the case of parameters qualified with the keywords "OPTIONAL" or "REPEATABLE" (described in Section 6.2), there must be exactly one argument in the procedure call for each parameter in the procedure declaration. The data type of each argument must be the same as the data type of the parameter at the corresponding position.

Every procedure parameter is classified as either "uses", "modifies", or "produces", depending on whether it was declared with the "USES", "MODIFIES", or "PRODUCES" parameter qualifier (an example of the "PRODUCES" qualifier is shown in the declaration of the procedure "getValue" in Example 5.1-4). Parameters declared with no qualifier are considered to be uses parameters; the "USES" keyword is therefore always redundant, and is used only to emphasize to a human reader that a parameter is not a produces or modifies. Uses and modifies parameters constitute "input" parameters, and correspond to Ada's "in" and "in out" parameters, respectively. Modifies and produces parameters are "output" parameters, corresponding to Ada's "in out" and "out" parameters. Input parameters are set to the value of their corresponding arguments when the procedure starts execution, and arguments corresponding to output parameters have their values set to those of the parameters when the procedure finishes.

5.3. How a Procedure Call Works

The substitution of an argument for a parameter during the execution of a program is called "passing the argument (as or for the parameter)". Within a procedure, parameters are treated like variables local to the procedure. Uses and modifies arguments are evaluated, and then their values are assigned to the procedure's parameters. When the procedure finishes execution, the modifies and produces parameter values are copied back to the corresponding arguments (which must be variables rather than non-variable expressions).

Parameters are passed only by the above method, which is called "copy-restore" (meaning that input argument values are copied into their corresponding parameters, and output parameter values are restored into their corresponding arguments). Some other program languages provide different parameter passing methods ("by reference" or "by name"), but MAINSAIL does not support these.

Variables declared within a procedure are called "local variables" because they cannot be used outside of the procedure (the MAINSAIL compiler gives an error message if you attempt to do so). Different procedures may use different local variables with the same name, as is the case with "bo" in Example 5.1-2. Data storage for local variables (and parameters) of a procedure is by default "automatic" (re-allocated on each procedure call), as opposed to the "static" allocation of many dialects of FORTRAN.

For example, consider the procedure declaration and call of Example 5.3-1. If variables are imagined as boxes in which information is stored, then the situation before the call to "proc" might look as in Example 5.3-2.

```
procedure declaration:

    PROCEDURE proc (USES INTEGER i1;
                    MODIFIES INTEGER i2;
                    PRODUCES INTEGER i3);
    BEGIN
    INTEGER i;
    i := (i1 * 2) - 1;
    i3 := i1 + i2;
    i2 := i + 1;
    i1 := i2;
    END;

call:

    proc(arg1A + arg1B,arg2,arg3);

arg1A, arg1B, arg2, and arg3 are all local
integer variables.
```

Example 5.3-1. A Procedure Declaration and Call

When the procedure call occurs, the parameters i1, i2, and i3 are allocated. The value "arg1A + arg1B" is calculated and stored into i1, and the value of arg2 is copied into i2. The value of i3 is not set; i3 is said to be "uninitialized". See Example 5.3-3.

```
Assume the variables arg1A, arg1B, arg2, and
arg3 have the values 6, 11, -4, and 9,
respectively.

+---------------+
|       6       | arg1A
+---------------+
|      11       | arg1B
+---------------+
|      -4       | arg2
+---------------+
|       9       | arg3
+---------------+
```

Example 5.3-2. Before the Procedure Call

```
+---------------+
|  17 (6 + 11)  | i1
+---------------+
|      -4       | i2
+---------------+
|   <unknown>   | i3
+---------------+
|       6       | arg1A
+---------------+
|      11       | arg1B
+---------------+
|      -4       | arg2
+---------------+
|       9       | arg3
+---------------+
```

Example 5.3-3. At the Time of the Procedure Call

Control passes to the first statement of the procedure "proc". The local variable "i" of proc is allocated; it is also uninitialized. See Example 5.3-4. The variables arg1A, arg1B, arg2, and arg3 are not shown. They are local to the (now inactive) procedure that called proc, and are therefore "invisible" while proc is executing.

```
+---------------+
|   <unknown>   | i
+---------------+
|      17       | i1
+---------------+
|      -4       | i2
+---------------+
|   <unknown>   | i3
+---------------+
```

Example 5.3-4. Upon Entry to "proc"

The arithmetic performed in proc sets the values of i, i1, i2, and i3 as shown in Example 5.3-5 by the time the end of the procedure is reached. As control leaves proc, the values of the local variables (in this case, the value of i) are discarded.

```
+---------------+
|      33       | i
+---------------+
|      34       | i1
+---------------+
|      34       | i2
+---------------+
|      13       | i3
+---------------+
```

Example 5.3-5. At the End of "proc"

Control returns to the point at which proc was called. The values of i2 and i3 are copied to arg2 and arg3, respectively. i1, i2, and i3 are then discarded. The final result looks like Example 5.3-6.

Note that the final statement of proc, "i1 := i2", has no effect, since the value of i1 is discarded when the procedure returns.

Care must be taken not to use uninitialized local variables or produces parameters. For example, if the body of proc had consisted entirely of the statement:

```
+---------------+
|       6       | arg1A
+---------------+
|      11       | arg1B
+---------------+
|      34       | arg2
+---------------+
|      13       | arg3
+---------------+
```

Example 5.3-6. After Return from "proc"

```
i2 := i3
```

then both i2 and i3 would have had unknown values upon exit from proc, and arg2 and arg3 would have been assigned unknown values. It is important to initialize any variable or parameter of which the value is to be used subsequently.

5.4. Typed Procedures and the Return Statement

A typed procedure is declared with a data type name immediately preceding the keyword "PROCEDURE". A typed procedure returns a value (by means of a Return Statement) that may be used in an expression by its caller. The same effect could be achieved by using passing a variable as a produces parameter to an untyped procedure, then using the variable in an expression, but a typed procedure is often more convenient syntactically. For example, the programs of Example 5.4-1 and Example 5.4-2 perform the same actions.

Like a call to an untyped procedure, a call to a typed procedure may be used as a Procedure Statement, in which case the value returned by the typed procedure is discarded. Typed procedures that perform some action but do not always return a value of interest to the program, like the system procedure "open" (see Chapter 9), are often used in Procedure Statements.

Every typed procedure must return a value of its declared type. It does this by means of a typed Return Statement, the form of which is:

```
RETURN(<expression>)
```

The expression is of the procedure's declared data type. When a Return Statement is executed, the procedure is immediately terminated, and the procedure returns the value of the expression.

```
BEGIN "cmpStr"

PROCEDURE isSame (STRING str1,str2; PRODUCES BOOLEAN bo);
BEGIN
bo := (str1 = str2);
END;



INITIAL PROCEDURE;
BEGIN
BOOLEAN same;
STRING s1,s2;

write(logFile,"First string: "); read(cmdFile,s1);
write(logFile,"Second string: "); read(cmdFile,s2);
isSame(s1,s2,same);
IF same THEN write(logFile,"Same." & eol)
EL write(logFile,"Different." & eol);
END;

END "cmpStr"
```

Example 5.4-1. An Untyped Procedure and a Variable

The MAINSAIL runtime system issues an error message if a typed procedure reaches its final "END" without executing a Return Statement.

A typed procedure call may be used as an expression; the value of the expression is the procedure's returned value, as shown in Example 5.4-2.

5.5. Order of Evaluation

MAINSAIL does not specify the order in which procedure arguments are evaluated. For example, consider the program fragment of Example 5.5-1. The call to p2 has two arguments, which may be evaluated in either order. Even though the value of n is known to be 12 immediately before the call to p2, you cannot tell from looking at the program whether the call to p2 will be equivalent to "p2(6,23)" (if the first argument is evaluated first) or "p2(6,12)" (if the second argument is evaluated first).

```
BEGIN "cmpStr"

STRING PROCEDURE getString (STRING promptString);
BEGIN
STRING s;

write(logFile,promptString); read(cmdFile,s);
RETURN(s);
END;



BOOLEAN PROCEDURE isSame (STRING str1,str2);
BEGIN
RETURN(str1 = str2);
END;



INITIAL PROCEDURE;
BEGIN
STRING s1,s2;

s1 := getString("First string: ");
s2 := getString("Second string: ");

IF isSame(s1,s2) THEN write(logFile,"Same." & eol)
EL write(logFile,"Different." & eol);
END;

END "cmpStr"
```

Example 5.4-2. Typed Procedures

The MAINSAIL compiler may choose one order of evaluation for such a call in one version of MAINSAIL, and a different order in another version. The MAINSAIL compiler does not issue an error message for procedure calls in which the order of evaluation is ambiguous, so it is your responsibility to avoid writing code of this type.

```
INTEGER PROCEDURE p1 (MODIFIES INTEGER i);
BEGIN
i := 23;
RETURN(6);
END;



PROCEDURE p2 (INTEGER i1,i2);
...

...
INTEGER n;
n := 12;
p2(p1(n),n); # This is ambiguous: p1 (which changes the
             # value of n) may be called before the second
             # argument is evaluated, so that the new
             # value of n is passed for the second
             # argument.  Alternatively, the second
             # argument may be evaluated first, in which
             # case the original value of n is used.
...
```

Example 5.5-1. Order of Evaluation Ambiguities

5.6. Exercises

Exercise 5-1.

Rewrite Example 4.7-1 to use procedures to avoid repeating a series of similar statements.

Exercise 5-2.

Does the code fragment from Example 5.4-2 shown in the upper part of Figure 5.6-1 perform the same actions as the code fragment in the lower part? Explain.

```
Does this code:

    STRING s1,s2;

    s1 := getString("First string: ");
    s2 := getString("Second string: ");

    IF isSame(s1,s2) THEN ...

do the same thing as this code:

    IF isSame(getString("First string: "),
            getString("Second string: ")) THEN ...
```

Figure 5.6-1. Sample Code Fragments

Exercise 5-3.

Using typed procedures, write a program that prompts for a nonnegative integer, then reports three facts about it:

- Whether it is odd or even.
- Whether it is prime.

- Whether its square root is an integer.

6. More on Procedures, Characters, and Strings

This chapter describes a variety of facilities for manipulating strings and characters in MAINSAIL. It describes the "OPTIONAL" and "REPEATABLE" parameter qualifiers, which provide syntactic flexibility for procedure calls. Section 6.4 gives important information on the scope of declarations.

6.1. Characters

Unlike some programming languages, MAINSAIL does not have a separate data type for representing characters. Characters are represented by their integer character codes. Character constants are formed by surrounding the selected character in single quotes (the "'" character); each character constant is really an integer constant. See Example 6.1-1.

```
The following are valid character constants:

    'A'
    'a'
    ' '
    '''

Since character constants belong to the data type
integer, the following are valid integer expressions:

    '0' + 4              (equal to '4', since the digits
                          are guaranteed to be contiguous)
    'Z' - 'A'            (not necessarily equal to 25,
                          since the letters are not
                          guaranteed to be contiguous)

If the character set is ASCII, the following
boolean expressions are true:

    'A' = 65
    'B' = 'A' + 1
    ' ' < '!'
```

Example 6.1-1. Character Constants

6.2. Repeatable and Optional Parameters

The parameter qualifiers "OPTIONAL" and "REPEATABLE" are used for syntactic convenience. They allow an argument in a procedure call to be omitted or to be repeated.

A procedure declaration with some optional parameters is shown in Example 6.2-1. Optional parameters follow all non-optional parameters in a parameter list. When no argument is specified corresponding to an optional parameter, the value 0 is passed if the data type is integer, the null string if the data type is string, and false if the data type is boolean. These values are called the "Zeros" of their respective data types (further described in Section 8.2).

A modifies or produces parameter may be declared optional, in which case the resulting value is discarded if no corresponding argument appears in the procedure call.

```
If the header of p looks like:

    PROCEDURE p (INTEGER i1;
                 OPTIONAL INTEGER i2;
                 MODIFIES OPTIONAL STRING s;
                 OPTIONAL BOOLEAN bo);

then the call:

    p(2)

is equivalent to:

    s := "";
    p(2,0,s,FALSE)

where s is a string variable the value of which is ignored
after the call to p.  Also equivalent are:

    s := "";
    p(2,0,s)

and:

    p(2,0)
```

Example 6.2-1. Optional Parameters

The last one or more parameters in a parameter list may be qualified with "REPEATABLE". The repeatable parameters are called the "repeatable group". More than one group of arguments may be passed for such a group of parameters. The procedure is reinvoked for each group of arguments passed for a repeatable group of parameters; see Example 6.2-2.

```
If p has the header:

    PROCEDURE p (INTEGER i1;
                 REPEATABLE INTEGER i2);

then the repeatable group of parameters is the single
parameter i2, and the call:

    p(1,4,6,-8)

is equivalent to:

    p(1,4);
    p(1,6);
    p(1,-8)

If p has the header:

    PROCEDURE p (INTEGER i1,i2;
                 REPEATABLE STRING s;
                 REPEATABLE INTEGER i3);

then the repeatable group of parameters is s and i3, and
the call:

    p(1,2,"three",3,"five",5,"eight",8)

is equivalent to:

    p(1,2,"three",3);
    p(1,2,"five",5);
    p(1,2,"eight",8)
```

Example 6.2-2. Repeatable Parameters

A parameter may be both optional and repeatable. The output from the execution of the program of Example 6.2-3 looks like Example 6.2-4.

```
BEGIN "repTst"

PROCEDURE p (OPTIONAL REPEATABLE INTEGER i);
BEGIN
write(logFile,"I is ",i,eol);
END;



INITIAL PROCEDURE;
BEGIN
p;       # Allowed because parameter is optional;
         # equivalent to "p(0)"
p(1);
p(5,6); # Allowed because parameter is repeatable
END;

END "repTst"
```

Example 6.2-3. A Repeatable Optional Parameter

```
*reptst<eol>
I is 0
I is 1
I is 5
I is 6
*
```

Example 6.2-4. Output from REPTST

A repeatable parameter may also be a modifies or produces parameter. Examples will be encountered later.

6.3. System Procedures

A number of identifiers, including the procedures "read" and "write" and the files "logFile" and "cmdFile", are "predeclared" by MAINSAIL; i.e., they may be used in a module that does not itself declare them. A "system procedure" is any predeclared procedure.

Unlike some languages for which partial or non-standard implementations exist, every implementation of MAINSAIL is guaranteed to have every system procedure listed in the "MAINSAIL Language Manual". You may use them all without fear of compromising the portability of your programs.

This section introduces some of the more common system procedures used for string processing. MAINSAIL is richly endowed with facilities for character and character string manipulation. Table 6.3-1 lists the headers of some of these system procedures. The headers shown are not necessarily the "real" headers, since some of the procedures are generic (meaning they have several forms); at this point, however, just treat the procedures as if they were declared as in Table 6.3-1. All system procedures are described more formally in the "MAINSAIL Language Manual".

6.3.1. cRead

The procedure cRead removes the first character from the string s and returns the character code of that character. For example, after the statements:

```
s := "ABC";
ch := cRead(s)
```

are executed, s has the value "BC", and ch the value 'A'.

If the string is the null string, it is unchanged, and the character code returned is -1.

6.3.2. cvcs

The procedure cvcs returns a string consisting of the character that is its argument. For example, "cvcs('X')" is equal to the string "X".

6.3.3. cvl and cvu

These procedures return the lowercase and uppercase versions of their arguments, respectively. The case of a string is changed by changing the case of every letter in the string; non-letter characters are not affected. For example, "cvl("ABC")" is equal to "abc"; "cvu("Hello,

```
INTEGER PROCEDURE cRead (MODIFIES STRING s);

STRING PROCEDURE cvcs (INTEGER characterCode);

STRING PROCEDURE cvl (STRING s);

INTEGER PROCEDURE cvl (INTEGER characterCode);

STRING PROCEDURE cvu (STRING s);

INTEGER PROCEDURE cvu (INTEGER characterCode);

STRING PROCEDURE cWrite (MODIFIES STRING s;
                         REPEATABLE INTEGER char);

INTEGER PROCEDURE first (STRING s);

INTEGER PROCEDURE last (STRING s);

INTEGER PROCEDURE length (STRING s);

INTEGER PROCEDURE rcRead (MODIFIES STRING s);

PROCEDURE rcWrite (MODIFIES STRING s;
                   REPEATABLE INTEGER char);

BOOLEAN PROCEDURE isUpperCase (INTEGER characterCode);

BOOLEAN PROCEDURE isLowerCase (INTEGER characterCode);

BOOLEAN PROCEDURE isAlpha (INTEGER characterCode);
```

Table 6.3-1. Some System Procedures for String and Character Manipulation

there.")" is equal to "HELLO, THERE.". Integer arguments are treated in the obvious way; e.g., "cvu('a')" is 'A'. These procedures are examples of generics; the compiler distinguishes between the forms to use based on whether the argument is a string or an integer.

6.3.4. cWrite

cWrite appends a character (or more than one, since its second argument is repeatable) to a string. After executing:

```
s := "ABC";
cWrite(s,'x','y')
```

s has the value "ABCxy".

6.3.5. first

The procedure "first" returns the character code of the first character of its string argument. Unlike cRead, it does not remove the character from the string. For example, after the statements:

```
s := "ABC";
ch := first(s)
```

are executed, s is unchanged, and ch has the value 'A'.

If the string is the null string, first returns -1.

6.3.6. last

last returns the character code of the last character of a string, without removing it from the string. The expression "last("ABC")" has the value 'C'.

If the string is the null string, last returns -1.

6.3.7. length

length returns the number of characters in a string. The expression "length("ABC")" has the value 3.

6.3.8. rcRead

rcRead returns the character code of the last character in a string. Unlike the procedure "last", it removes the character from the string. The "r" in "rcRead" stands for "reverse"; i.e., rcRead is the reverse of cRead, since it operates on the opposite end of the string.

If the string is the null string, it is unchanged, and the character code returned is -1.

6.3.9. rcWrite

rcWrite prepends a character to the beginning of a string (being the "reverse" of cWrite, which appends a character to the end).

6.3.10. isUpperCase, isLowerCase, and isAlpha

isUpperCase and isLowerCase return true if their arguments are the character codes for an upper- or lowercase letter, respectively, and false otherwise (they always return false for non-letters). isAlpha returns true if and only if its argument is a letter, i.e., if isUpperCase or isLowerCase is true of its argument.

6.3.11. String Processing Example

The program of Example 6.3.11-1 produces a substring of a given string, given the first and last character positions in the string to include in the substring (this is not really necessary, since MAINSAIL has a built-in substring mechanism, which is described in Section 10.5). Enter, compile, and execute the module of Example 6.3.11-1.

6.4. Scopes

So far the only declarations and definitions encountered have been "local", i.e., confined to the scope of a single procedure. When the MAINSAIL compiler reaches the end of a procedure, it forgets that it has ever seen the locally defined identifiers.

An "outer" declaration may be used to create an identifier that is visible to all subsequently declared procedures in the same module. Every procedure declaration is an outer declaration in the sense that every following procedure may call it (preceding procedures may also call it under some circumstances; see Section 7.2). Outer macros may be used by several procedures in a module, and outer variables may be used instead of parameters to pass information among procedures. Outer variables do not lose their values when a procedure exits; they persist as long as the containing module does.

Outer declarations appear between the initial "BEGIN" and final "END" of a module, outside of a procedure. Their format is exactly the same as that of local declarations. Outer macro defintions, variable declarations, and procedure declarations may appear in any order, as long as each identifier is declared before it is used.

```
BEGIN "subStr"

PROCEDURE convertPos (MODIFIES INTEGER i;
                      STRING s);
# I is a string position.  If negative, convert to
# equivalent positive position; otherwise, don't change
# it.  See comment at procedure subString.
BEGIN
IF i < 0 THEN i := length(s) + i + 1;
# If less than 1, it's before the beginning of the string:
IF i < 1 THEN i := 1;
END;



STRING PROCEDURE subString (STRING s;
                            INTEGER startPos,stopPos);
# String positions are numbered starting at 1.  If
# startPos or stopPos is negative, it specifies a position
# from the end of the string, which is position -1.
# Position 0 for either start or stop causes the null
# string to be returned.
# Positions beyond the end (or beginning) of the string
# are converted to the end (or beginning) of the string.
# If stopPos is less than startPos, the null string is
# returned.
BEGIN
INTEGER i;

IF startPos = 0 OR stopPos = 0 THEN RETURN("");
# Convert positions to equivalent positive positions
convertPos(startPos,s); convertPos(stopPos,s);
# Now remove extra trailing characters, if any
WHILE length(s) > stopPos DO
    rcRead(s); # Remove and discard last character of s
# Now remove initial characters, if any
FOR i := 1 UPTO startPos - 1 DO
    cRead(s); # Remove and discard first character of s
RETURN(s);
END;
```

Example 6.3.11-1. Getting a Substring (continued)

```
INITIAL PROCEDURE;
BEGIN
STRING s,t;
INTEGER start,stop;

DOB write(logFile,"String of which to take substring" &
           " (<eol> to quit): ");
    read(cmdFile,s);
    IF s NEQ "" THENB
        write(logFile,"Start position: ");
        read(cmdFile,t); read(t,start);
        write(logFile,"Stop position: ");
        read(cmdFile,t); read(t,stop);
        write(logFile,"Substring is """,
            subString(s,start,stop),"""" & eol);
        END END UNTIL s = "";
END;

END "subStr"
```

Example 6.3.11-1. Getting a Substring (end)

In MAINSAIL, unlike Pascal and some other languages, procedures may not contain other ("nested") procedures.

Example 6.4-1 shows the use of some outer variables in a simple Reverse Polish Notation calculator that performs only addition. Each line typed by the user may consist of unsigned integers, plus signs, the command "S" (show the contents of the stack), and the command "Q" (quit), separated from one another by spaces and tab characters. When it sees an integer, it pushes it onto its "stack" of integers. When it sees a plus sign, it removes the top two integers from the stack, adds them, and pushes the result back onto the stack, unless there are fewer than two integers, in which case it gives an error message. When it sees an "S", it prints the contents of the stack, bottom to top. When it sees a "Q", it stops. At the end of each command line, it prints the value at the top of the stack, if any. Example 6.4-2 shows a sample execution.

6.5. When to Use Outer Variables

stack, stackDepth, and timeToQuit are the outer variables in Example 6.4-1. They could all have been declared in the initial procedure instead, but then they would have had to be passed as parameters to all the procedures that manipulate them. In the case of stack and stackDepth,

```
BEGIN "rpn"

BOOLEAN timeToQuit; # An outer variable; true if "Q"
                    # command seen

STRING stack;       # The stack of numbers.

INTEGER stackDepth; # How many numbers are on the stack
                    # now.

STRING PROCEDURE getCommand (MODIFIES STRING s);
# Return the next thing on the command line.  If it is
# "Q" or the end of the line, return the null string.
# If it is illegal, write an error message and return
# the null string.
# Remove the string read from s.
BEGIN
INTEGER ch;
STRING t,u;

WHILE first(s) = ' ' OR first(s) = first(tab) DO cRead(s);
    # Remove initial blanks and tabs, if any
t := "";
WHILE s NEQ "" AND
    (first(s) NEQ ' ' AND first(s) NEQ first(tab)) DO
    cWrite(t,cRead(s)); # Add the next character to the
                        # result string

IF t = "Q" THENB timeToQuit := TRUE; RETURN("") END;
IF t = "+" OR t = "S" OR t = "" THEN RETURN(t);
```

Example 6.4-1. Use of Outer Variables (continued)

```
# If it isn't "Q", "S", "+", or nothing, then it must be
# an integer or illegal.
u := t;
WHILE u DOB
    ch := cRead(u);
    # Take advantage of the assumption that the digit
    # characters are contiguous (see the "MAINSAIL
    # Language Manual").
    IF ch < '0' OR ch > '9' THENB
        write(logFile,"Illegal command ",t,eol);
        RETURN("") END END;
RETURN(t); # It was an integer
END;



PROCEDURE push (STRING s);
# Add the integer in s to the top of the stack.  The
# integers are separated by the space character.
BEGIN
cWrite(stack,' '); stack := stack & s;
stackDepth := stackDepth + 1;
END;



STRING PROCEDURE stackTop (OPTIONAL BOOLEAN doNotPop);
# Unless doNotPop is true, remove the top item from the
# stack.
BEGIN
INTEGER ch;
STRING s;
```

Example 6.4-1. Use of Outer Variables (continued)

```
IF stackDepth < 1 THENB
    write(logFile,"Stack empty."); RETURN("") END;
s := "";
DOB ch := rcRead(stack);
    IF ch NEQ ' ' THEN rcWrite(s,ch);
    END UNTIL ch = ' ';
stackDepth := stackDepth - 1;
IF doNotPop THEN push(s); # Put it back onto the stack
RETURN(s);
END;



STRING PROCEDURE stringAdd (STRING s,t);
# The strings s and t represent integers.  Return the
# string that represents their sum.
BEGIN
INTEGER i,j;
STRING u;
# Take the easy way out by using "read" and "write".
# Note that if i, j, or their sum is larger than 32,767,
# this procedure may not work.
read(s,i); read(t,j); u := ""; write(u,i + j);
RETURN(u);
END;
```

Example 6.4-1. Use of Outer Variables (continued)

```
PROCEDURE processCommand (STRING s);
# If s = "+", pop the top two items from the stack, add
# them, then push the result onto the stack.  If s is
# an integer, push it onto the stack.
BEGIN
STRING t,u;
IF s = "S" THEN write(logFile,"Stack:",stack,eol)
EF s = "+" THENB
    IF stackDepth < 2 THEN
        write(logFile,"Cannot add; stack has only ",
            stackDepth," items." & eol)
    EB  t := stackTop; u := stackTop;
        push(stringAdd(t,u)) END END
EL  push(s);
END;



INITIAL PROCEDURE;
BEGIN
STRING s,t;

timeToQuit := FALSE; # Haven't seen "Q" yet
stack := ""; stackDepth := 0; # No numbers on stack yet

DOB write(logFile,"CALC: "); read(cmdFile,s);
    s := cvu(s);  # Convert to upper case so as not to
                  # have to distinguish "Q"/"q", "S"/"s"
    DOB t := getCommand(s); # getCommand returns the null
                            # string at end of line or if
                            # command "Q" seen
        IF t NEQ "" THEN processCommand(t);
        END UNTIL t = "";
    write(logFile,stackTop(TRUE),eol);
    END UNTIL timeToQuit;
END;

END "rpn"
```

Example 6.4-1. Use of Outer Variables (end)

this would be inconvenient, since nearly all the procedures manipulate them, and the same two parameters would have to be passed over and over again.

```
*rpn<eol>
CALC: s<eol>
Stack:
Stack empty.
CALC: 1 6<eol>
6
CALC: s<eol>
Stack: 1 6
6
CALC: +<eol>
7
CALC: 8 13 + 22 + + s<eol>
Stack: 50
50
CALC: 16 s<eol>
Stack: 50 16
16
CALC: + wfkpe<eol>
Illegal command WFKPE
66
CALC: +<eol>
Cannot add; stack has only 1 items.
66
CALC: q<eol>
66
*
```

Example 6.4-2. Execution of RPN

Some philosophers of computer programming consider that is better, when possible, to use local variables and pass parameters instead of sharing data among procedures using outer variables. They think this makes it more obvious what data are consumed and produced by each procedure, since all of the relevant input values and output variables appear in the procedure call itself. Others feel that long lists of parameters passed over and over to different procedures clutter the code and make programs difficult to read, and try to pass as much information as possible in outer variables. The examples in this tutorial try to steer a middle course; data shared among many procedures are represented in outer variables, whereas information used primarily in a small number of procedures is passed around through parameters.

Under these principles, timeToQuit in Example 6.4-1 might well have been declared in the initial procedure and set by getCommand, since getCommand and the initial procedure are the only two procedures in which the variable timeToQuit appears. getCommand's procedure header would have been changed to:

```
    STRING PROCEDURE getCommand (MODIFIES STRING s;
                                 PRODUCES BOOLEAN timeToQuit);
```

and its call in the initial procedure to:

```
                        getCommand(s,timeToQuit)
```

However, timeToQuit was made an outer variable with the idea that the program might someday be expanded to have more conditions that required termination, and so procedures other than getCommand might set timeToQuit. timeToQuit can also be viewed as a property of the program as a whole, and therefore worthy of appearing in the declarations at the beginning of the module, where a human reader trying to understand the program often starts.

6.6. Exercises

Exercise 6-1.

Rewrite the procedure stringAdd in Example 6.4-1 to handle numbers larger than MAINSAIL's maximum integer by doing the arithmetic on strings yourself instead of calling "read" and "write". You need handle only nonnegative numbers.

Exercise 6-2.

Write a more sophisticated Reverse Polish Notation calculator than that of Example 6.4-1. Support negative and nonnegative integers of any length, subtraction, multiplication, and division as well as addition. Also provide a way to clear the stack (i.e., remove all the numbers from the stack).

7. Even More on Procedures; the Done and Continue Statements

This chapter describes the Done and Continue Statements, which are flow-of-control constructs used within an iterated statement. It describes recursive procedures (procedures that call themselves, directly or indirectly), own (static) variables, and the generic procedure construct, which allows one identifier to represent more than one procedure.

7.1. The Done, Continue, and Untyped Return Statements

7.1.1. The Done Statement

The Done Statement may appear in the iterated statement part of an Iterative Statement. When it is executed, the innermost Iterative Statement is terminated (a named Iterative Statement may be terminated from an Iterative Statement nested within it; see the procedure "search" in Example 16.4-1). Example 7.1.1-1 shows the use of a Done Statement. It consists of just the keyword "DONE".

7.1.2. The Continue Statement

Like the Done Statement, the Continue Statement appears in the iterated statement part of an Iterative Statement. Instead of terminating the iteration, however, a Continue Statement causes control to skip to the end of the iterated statement. If there is a FOR-clause, WHILE-clause, and/or UNTIL-clause, the appropriate increments, decrements, and/or tests are performed, and the iteration is restarted if the tests pass. Like the Done Statement, the Continue Statement may operate on a named Iterative Statement. See Example 7.1.2-1.

7.1.3. The Untyped Return Statement

An untyped procedure may be terminated before it reaches the final "END" by means of an untyped Return Statement, the form of which is:

```
RETURN
```

with no following parentheses. Examples will appear later.

```
BEGIN "bkwrds"

STRING PROCEDURE backwards (STRING s);
# Reverse the characters of s.
BEGIN
STRING t;

t := "";
WHILE s NEQ "" DO cWrite(t,rcRead(s));
RETURN(t);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

DOB write(logFile,"String to reverse (<eol> to quit): ");
    read(cmdFile,s); IF s = "" THEN DONE;
    write(logFile,backwards(s),eol) END;
END;

END "bkwrds"
```

Example 7.1.1-1. The Done Statement

7.2. Recursion and Forward Procedures

It is sometimes useful to have a procedure call itself, or to call other procedures that call it. This process is known as "recursion". Every MAINSAIL procedure may be recursive, i.e., may call itself or call procedures that call it.

7.2.1. A Procedure That Calls Itself

On a recursive procedure invocation, new copies of the procedure's parameters and local variables are made, as described in Section 5.3. Consider the procedure of Example 7.2.1-1, which performs integer multiplication by repeated addition (not a very fast method). In brief,

```
BEGIN "sums"

# Sum the integers input from cmdFile, one on each line.
# Require that each integer be in a valid format.

BOOLEAN PROCEDURE validInteger (STRING s);
BEGIN
INTEGER ch;

IF first(s) = '-' THEN cRead(s); # May be initial "-"
WHILE s DOB
    ch := cRead(s);
    IF ch < '0' OR ch > '9' THEN RETURN(FALSE) END;
RETURN(TRUE); # If made it to here, must be OK
END;



INITIAL PROCEDURE;
BEGIN
INTEGER i,sum;
STRING s;

sum := 0;
DOB write(logFile,"Next integer (<eol> to stop): ");
    read(cmdFile,s); IF s = "" THEN DONE;
    IF NOT validInteger(s) THENB
        write(logFile,s,": invalid integer" & eol);
        CONTINUE END;
    read(s,i); sum := sum + i;
    write(logFile,"Sum: ",sum,eol) END;
END;

END "sums"
```

Example 7.1.2-1. Use of the Continue Statement

the call "mul(5,2)" returns "mul(5,1) + 5". In turn, "mul(5,1)" returns "mul(5,0) + 5"; "mul(5,0)" returns 0. Therefore, the call "mul(5,2)" returns (0 + 5) + 5, or 10, as desired.

Consider that the original call "mul(5,2)" is "invocation #1" of the procedure, as shown in Example 7.2.1-2.

```
INTEGER PROCEDURE mul (INTEGER i,j);
# Calculate the product of i and j by repeated addition.
# j must be nonnegative.
BEGIN
IF j = 0 THEN RETURN(0); # Zero times anything is zero
# otherwise j > 0:
RETURN(mul(i,j - 1) + i); # (i * (j - 1)) + i = j * i
```

Example 7.2.1-1. A Recursive Procedure to Perform Multiplication

```
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #1
|     j = 2     |  /
+---------------+
```

Example 7.2.1-2. Invocation #1 of "mul" on Entry to the Procedure

Invocation #1 finds that j is not zero, so it executes the second statement of the procedure, which requires it to evaluate "mul(i,j - 1)", i.e., "mul(5,1)". Invocation #1 now calls "mul" with the arguments 5 and 1, resulting in invocation #2. See Example 7.2.1-3

```
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #2
|     j = 1     |  /
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #1 (inactive)
|     j = 2     |  /
+---------------+
```

Example 7.2.1-3. Invocation #2 of "mul" on Entry to the Procedure

Invocation #2 again finds that j is not zero, so it too executes the second statement, which requires it to evaluate "mul(5,0)", creating invocation #3. The situation then looks like Example 7.2.1-4.

```
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #3
|     j = 0     |  /
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #2 (inactive)
|     j = 1     |  /
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #1 (inactive)
|     j = 2     |  /
+---------------+
```

Example 7.2.1-4. Invocation #3 of "mul" on Entry to the Procedure

Invocation #3 finds that j is zero, so it returns zero. The Return Statement immediately terminates invocation #3 so that invocation #2 becomes active again. See Example 7.2.1-5.

```
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #2
|     j = 1     |  /
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #1 (inactive)
|     j = 2     |  /
+---------------+
```

Example 7.2.1-5. Invocation #2 of "mul" upon Return from #3

Invocation #2, which was in the middle of evaluating the expression in the Return Statement, receives the value 0 returned by invocation #3, and so calculates "0 + i", i.e., "0 + 5". Invocation #2 therefore terminates, returning the value 5. Consequently, invocation #1 becomes active again. See Example 7.2.1-6.

```
+---------------+
|     i = 5     |  \
+---------------+   > Invocation #1
|     j = 2     |  /
+---------------+
```

Example 7.2.1-6. Invocation #1 of "mul" upon Return from #2

Invocation #1, which was also evaluating the expression in the middle of the Return Statement, receives the value 5 returned by invocation #2. Therefore, it calculates and returns "5 + 5", i.e., 10, as expected.

7.2.2. Mutual Recursion and the "FORWARD" Qualifier

Recursion may occur when two procedures call each other ("mutual recursion"). MAINSAIL requires every procedure's declaration to appear in the source file prior to any calls to the procedure. If procedure a calls procedure b, and b calls a, this poses a problem, since b must be declared before a, but a must be declared before b. The "forward declaration" provides a solution to this problem. An example is shown in Example 7.2.2-2, where the procedures expression and value are given forward declarations.

The program CALC of Example 7.2.2-2 is an interactive calculator, like that of Example 6.4-1. However, CALC accepts a more sophisticated syntax; a summary of its grammar is shown in Example 7.2.2-1. The technique used in Example 7.2.2-2 to parse the grammar is called "recursive descent", since it uses recursion in "descending" (breaking down) sentences of the grammar.

Each line of CALC input must be a "command" in accordance with the rules of Example 7.2.2-1. A sample run is shown in Example 7.2.2-3.

The forward declaration of a procedure appears before the first call to a procedure. It is composed of the keyword "FORWARD" followed by the header of the procedure. Unlike Pascal, MAINSAIL requires that the full header of the procedure be given again when the full procedure declaration (including the procedure body) appears. The full declaration is said to "give the forward procedure a body".

```
"=>" means "is defined as".  Single quote marks are
placed around commands typed literally.  Comments appear
in parentheses.  Square brackets are used for grouping,
"!" appears between alternatives, and "*" indicates
zero or more repetitions of the preceding item.

command => 'S' (show all accumulators)
        => 'Q' (quit)
        => '' (empty command line)
        => value (print the value)

value   => assignment
        => expression

assignment => register '=' value
        (do the assignment and return the value assigned)

expression => term [['+' term] ! ['-' term]]*

term    => factor [['*' factor] ! ['/' factor]]*

factor  => unsignedInteger
        => register
        => '(' expression ')'

An unsignedInteger is just an unsigned integer.

register => 'A' (there are three registers)
         => 'B'
         => 'C'
```

Example 7.2.2-1. The Grammar of CALC Commands

```
BEGIN "calc"

INTEGER a,b,c;  # Values in the accumulators
```

Example 7.2.2-2. Mutual Recursion (continued)

```
STRING PROCEDURE getToken (MODIFIES STRING s);
# Remove the next thing from s.
BEGIN
INTEGER ch;
STRING t;

WHILE first(s) = ' ' OR first(s) = first(tab) DO cRead(s);
IF NOT s THEN RETURN(""); # End of string
ch := cRead(s);
# Everything but integers is one character:
IF ch < '0' OR ch > '9' THEN RETURN(cvcs(ch));
# Now it must be an integer:
t := cvcs(ch);
WHILE first(s) GEQ '0' AND first(s) LEQ '9' DO
    cWrite(t,cRead(s));
RETURN(t);
END;



PROCEDURE ungetToken (MODIFIES STRING s;
                      REPEATABLE STRING token);
# Put token back at the front of s.
BEGIN
s := token & " " & s;
END;



BOOLEAN PROCEDURE validInteger (STRING s);
# Returns true if s is an unsigned integer.
BEGIN
INTEGER ch;

IF NOT s THEN RETURN(FALSE);
WHILE s DOB
    ch := cRead(s);
    IF ch < '0' OR ch > '9' THEN RETURN(FALSE) END;
RETURN(TRUE); # If made it to here, must be OK
END;
```

Example 7.2.2-2. Mutual Recursion (continued)

```
FORWARD INTEGER PROCEDURE expression (MODIFIES STRING s);

INTEGER PROCEDURE factor (MODIFIES STRING s);
BEGIN
INTEGER i;
STRING t;

t := getToken(s);
IF t = "A" THEN RETURN(a)
EF t = "B" THEN RETURN(b)
EF t = "C" THEN RETURN(c)
EF t = "(" THENB
    i := expression(s); t := getToken(s);
    IF t NEQ ")" THEN
        write(logFile,"Factor: missing ')'" & eol);
    RETURN(i) END
EF validInteger(t) THENB read(t,i); RETURN(i) END
EB  write(logFile,"Factor: illegal factor ",t,eol);
    RETURN(0) END; # Treat garbage as a zero
END;



INTEGER PROCEDURE term (MODIFIES STRING s);
BEGIN
INTEGER product,i;
STRING t;

product := factor(s);
DOB t := getToken(s);
    IF t = "*" THEN product := product * factor(s)
    EF t = "/" THENB
        i := factor(s);
        IF i NEQ 0 THEN product := product DIV i
        EB  write(logFile,"Term: division by zero" & eol);
            product := 0 END END # Give zero result
    EB ungetToken(s,t); DONE END END;
RETURN(product);
END;
```

Example 7.2.2-2. Mutual Recursion (continued)

```
INTEGER PROCEDURE expression (MODIFIES STRING s);
BEGIN
INTEGER sum;
STRING t;

sum := term(s);
DOB t := getToken(s);
    IF t = "+" THEN sum := sum + term(s)
    EF t = "-" THEN sum := sum - term(s)
    EB ungetToken(s,t); DONE END END;
RETURN(sum);
END;


FORWARD INTEGER PROCEDURE value (MODIFIES STRING s);

INTEGER PROCEDURE assignment (MODIFIES STRING s);
BEGIN
STRING regName;
INTEGER val;

regName := getToken(s); getToken(s); # Discard "="
val := value(s);
IF regName = "A" THEN a := val
EF regName = "B" THEN b := val
EF regName = "C" THEN c := val
EL write(logFile,"Assignment: illegal register name ",
        regName,eol);
RETURN(val);
END;



INTEGER PROCEDURE value (MODIFIES STRING s);
BEGIN
BOOLEAN isAssignment;
STRING t,u;
```

Example 7.2.2-2. Mutual Recursion (continued)

```
t := getToken(s); u := getToken(s);
isAssignment := (u = "=");
ungetToken(s,u,t);
IF isAssignment THEN RETURN(assignment(s))
EL RETURN(expression(s));
END;



BOOLEAN PROCEDURE doCommand (STRING s);
# Return false if the command was "Q", true otherwise.
BEGIN
STRING t;

t := getToken(s);
IF t = "Q" THEN RETURN(FALSE);
IF t = "S" THEN
    write(logFile,"A: ",a,"  B: ",b,"  C: ",c,eol)
EF t THENB
    ungetToken(s,t); write(logFile,value(s),eol) END;
IF t := getToken(s) THEN # Something was left over
    write(logFile,"Illegal token ",t,eol);
RETURN(TRUE);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

# Set up the accumulators:
a := 0; b := 0; c := 0;

DOB write(logFile,"CALC command ('Q' to quit): ");
    read(cmdFile,s) END UNTIL NOT doCommand(cvu(s));
END;

END "calc"
```

Example 7.2.2-2. Mutual Recursion (end)

```
*calc<eol>
CALC command ('Q' to quit): 3<eol>
3
CALC command ('Q' to quit): s<eol>
A: 0  B: 0  C: 0
CALC command ('Q' to quit): a = b = 3<eol>
3
CALC command ('Q' to quit): s<eol>
A: 3  B: 3  C: 0
CALC command ('Q' to quit): c = b + 9 / (a - 1)<eol>
7
CALC command ('Q' to quit): 2+2<eol>
4
CALC command ('Q' to quit): z<eol>
Factor: illegal factor Z
0
CALC command ('Q' to quit): s<eol>
A: 3  B: 3  C: 7
CALC command ('Q' to quit): q<eol>
*
```

Example 7.2.2-3. Sample Execution of CALC

7.3. Source Libraries

A procedure declared with the "FORWARD" qualifier need not be given a body if it is never called. This fact is used with a form of the "FORWARD" qualifier that indicates the source file name in which the procedure may be found to produce "source libraries" or "compiletime libraries", files that contain procedures to be compiled into a module only if they are used by the module.

Source libraries are explained in detail in Section 6.1 of part II of the "MAINSAIL Tutorial".

7.4. Own Variables

Although most variables declared within a procedure are allocated dynamically each time the procedure is invoked, there is a type of variable called the "(local) own variable" that is not lost from invocation to invocation of the procedure. The own variable persists like an outer variable; the difference is that an own variable may be used only within the procedure where it is declared. Example 7.4-1 shows how an own variable, depth, is used to prevent the depth of a recursive procedure from exceeding a certain value. Another own variable, messageGiven, ensures that the error message for a deep recursion is printed only once. The procedure "ack" implements Ackermann's function.

The value of an own variable is initially the Zero for its data type. For an integer, this means the value 0; for a boolean, the value false. See Section 8.2.

```
BEGIN "acker"

DEFINE maxDepth = 100;

INTEGER PROCEDURE ack (INTEGER m,n);
# Return zero if we've gone too deep.
BEGIN
INTEGER retVal;
OWN BOOLEAN messageGiven;
OWN INTEGER depth;

depth := depth + 1;
IF messageGiven THEN retVal := 0
EF depth GEQ maxDepth THENB
    write(logFile,"Ack: depth GEQ ",maxDepth,eol);
    messageGiven := TRUE; retVal := 0 END
EF m = 0 THEN retVal := n + 1
EF n = 0 THEN retVal := ack(m - 1,1)
EL retVal := ack(m - 1,ack(m,n - 1));
depth := depth - 1; RETURN(retVal);
END;



INITIAL PROCEDURE;
BEGIN
INTEGER m,n,result;
STRING s;

write(logFile,"m: "); read(cmdFile,s); read(s,m);
write(logFile,"n: "); read(cmdFile,s); read(s,n);
result := ack(m,n);
write(logFile,"ack(",m,",",n,") = ",result,eol);
END;

END "acker"
```

Example 7.4-1. An Own Variable Used to Keep Track of Recursion Depth

7.5. Generic Procedures

You may have been wondering how the procedures "read" and "write" manage to take arguments of a variety of data types. The answer is that they are declared as "generic" procedures.

A generic procedure declaration does not look like an ordinary procedure declaration; rather, it has the form:

```
    GENERIC PROCEDURE <generic name> <list of instance names>
```

The generic name is the name to be used when the procedure is called. A call using the generic name is called a "generic call". Each generic call actually selects a call to one or more of the instance procedures in the generic procedure declaration, based on the data types of the arguments.

For example, we have seen that the procedure "write" may write either a string or an integer to a text file. The generic procedure declaration of "write" looks (in part) like

```
                GENERIC PROCEDURE write "itWrite,stWrite";
```

(there are actually many more instance procedures for "write" than just itWrite and stWrite). The list of instance procedure names is just a string constant.

The instance procedures itWrite and stWrite are declared as in Example 7.5-1. "POINTER(textFile)" is the data type designator for a file; this form is explained more fully in Chapters 9 and 11.

```
PROCEDURE itWrite
    (POINTER(textFile) f; REPEATABLE INTEGER v);

PROCEDURE stWrite
    (POINTER(textFile) f; REPEATABLE STRING v);
```

Example 7.5-1. Instance Procedure Headers for "write"

Because the last parameter to the instance procedures for "write" is repeatable, "write" accepts as its arguments first a file, then a series of expressions that may be either integers or strings. For each expression that is an integer, "write" calls itWrite; for each string, it calls stWrite. See Example 7.5-2.

```
The generic call:

    write(logFile,"The square of ",i," is ",i * i,eol)

results in the instance procedure calls (assume i is
an integer):

    stWrite(logFile,"The square of ");
    itWrite(logFile,i);
    stWrite(logFile," is ");
    itWrite(logFile,i * i);
    stWrite(logFile,eol)
```

Example 7.5-2. Selection of Instance Procedures for "write"

You should never call an instance procedure of a generic system procedure directly; you should always use the generic name. XIDAK reserves the right to change the instance names of generic system procedures without notice.

The exact algorithm used in the selection of generic procedures is fairly complicated to account for some special cases, and may be found in the "MAINSAIL Language Manual".

A sample (and rather artificial) use of generic procedures is shown in Example 7.5-3.

```
BEGIN "quotes"

PROCEDURE quoteNum (INTEGER i);
BEGIN
write(logFile,"""",i,"""",eol);
END;



PROCEDURE quoteString (STRING s);
BEGIN
write(logFile,"""" & s & """",eol);
END;



GENERIC PROCEDURE quote "quoteNum,quoteString";



INITIAL PROCEDURE;
BEGIN
INTEGER i;
STRING s;

DOB write(logFile,"A string (<eol> to stop): ");
    read(cmdFile,s); IF s = "" THEN DONE;
    quote(s);
    write(logFile,"A number: "); read(cmdFile,s);
    read(s,i); quote(i) END;
END;

END "quotes"
```

Example 7.5-3. User-Defined Generic Procedures

7.6. Exercises

Exercise 7-1.

Write a program that reads lines from cmdFile and translates them into Pig Latin, terminating when a blank line is read. Pig Latin words are the same as their English equivalents, except that words beginning with a vowel have "yay" appended to them, and words beginning with one or more consonants have the consonants stripped from the front and appended to the end, followed by "ay". For example, "oink" translates to "oinkyay", "mud" to "udmay", "sty" to "ystay". If an English word starts with an uppercase letter, its Pig Latin translation should also start with an uppercase letter (you may assume that words are otherwise all lowercase). Be sure to do something sensible with punctuation (don't throw it away).

Exercise 7-2.

Modify the calculator program of Example 7.2.2-2 to operate on hexadecimal numbers of arbitrary precision, both positive and negative, rather than decimal numbers limited to the maximum size of a MAINSAIL integer.

8. More Data Types, Variables, Constants, and Expressions: Long Integer, Real, Long Real, Bits, and Long Bits

This chapter describes all of the "high-level" MAINSAIL data types (except pointer) that have not yet been encountered, as well as such important concepts as the Zero of a data type and conversion procedures. There are also tables listing all the unary and binary operators provided by MAINSAIL.

8.1. More Data Types

The data types seen up to this point are boolean, integer, and string. The data types introduced in this chapter are long integer, real, long real, bits, and long bits. The data type pointer, which is used to manipulate array and record data structures, is described in Chapter 11. The discussion of the last two data types, address and charadr, is deferred until Chapter 18.

The procedures "read" and "write" may be used with any of the data types described in this chapter. They treat the other numeric and bits data types in a fashion analogous to the way they treat integers. Complete descriptions of "read" and "write" may be found in the "MAINSAIL Language Manual".

8.1.1. Long Integers

The data type long integer is designated by "LONG INTEGER". Long integer constants have the same form as numeric integer constants except that they are followed by the letter "L" (or lowercase "l"). The guaranteed range of long integers is -2,147,483,647 to +2,147,483,647, as opposed to the -32,767 to +32,767 range of the integer data type.

The operations "+", "-", "*", and "DIV" all operate on long integers as well as integers. Long integers and integers may not be mixed in the same arithmetic expression, however; e.g., "2L + 2" is illegal.

Like integers, long integers may be used in FOR-clauses. The iteration variable, lower limit, and upper limit must all be of the same data type, i.e., all integers or all long integers.

The module TETRA of Example 8.1.1-1 prompts for the value of N, then uses long integers to calculate the Nth tetrahedral number (the sum of the first N triangular numbers). Note that when "read" reads a long data type from a file or a string, it does not expect a trailing "L". For example, execution of TETRA looks as in Example 8.1.1-2.

```
BEGIN "tetra"

LONG INTEGER PROCEDURE triang (LONG INTEGER ii);
BEGIN
LONG INTEGER jj,sum;

sum := 0L;
FOR jj := 1L UPTO ii DO sum := sum + jj;
RETURN(sum);
END;



INITIAL PROCEDURE;
BEGIN
LONG INTEGER ii,jj,sum;
STRING s;

write(logFile,"Calculate what tetrahedral number? ");
read(cmdFile,s); read(s,ii); sum := 0L;
FOR jj := 1L UPTO ii DO sum := sum + triang(jj);
write(logFile,"The ",ii,"th tetrahedral number is ",sum,
    eol);
END;

END "tetra"
```

Example 8.1.1-1. Long Integers to Calculate Tetrahedral Numbers

8.1.2. Reals and Long Reals

Reals and long reals represent floating point numbers. They use standard decimal notation, with a period (".") separating the integer part from the fractional part (the presence of the period is what distinguishes a real or long real constant from an integer or long integer constant). Long reals are distinguished from reals by the addition of an "L" at the end of the constant. Adding an "E" followed by an integer N to a floating point number multiplies by ten-to-the-Nth power, e.g., "1.0E-6L" is the long real fraction one one-millionth.

"+", "-", and "*" are implemented for reals and long reals as for integers and long integers. The real and long real division symbol is "/", not "DIV".

```
*tetra<eol>
Calculate what tetrahedral number? 3<eol>
The 3th tetrahedral number is 10
*tetra<eol>
Calculate what tetrahedral number? 100<eol>
The 100th tetrahedral number is 171700
*tetra<eol>
Calculate what tetrahedral number? 200<eol>
The 200th tetrahedral number is 1353400
*
```

Example 8.1.1-2. The Execution of TETRA

A complete set of trigonometric functions is provided for real numbers. See Example 8.1.2-1. A more complete specification of the real and long real data types may be found in the "MAINSAIL Language Manual".

8.1.3. Bits and Long Bits

The data types bits and long bits are used to represent "bit vectors" or small sets of "flags" or "bits" that may be either "set" or "clear". Each flag resembles a boolean variable in that it may have only two values; the "set" or "1" value corresponds to the boolean "TRUE" and the "clear" or "0" value to "FALSE". A bits or long bits value may also be thought of as a series of binary digits (an unsigned binary number). A bits value may have up to 16 binary digits; a long bits, up to 32 (these are the portably guaranteed ranges; some implementations may provide more binary digits).

A bits constant is written as a single quote mark ("'") followed by a radix letter, "B", "O", or "H", followed by a binary, octal, or hexadecimal number. The octal forms represent three binary digits with each octal digit; the hexadecimal forms, four binary digits with each hexadecimal digit. The octal digits "0" through "7" represent the binary sequences "000" through "111"; the hexadecimal digits "0" through "9", "A" through "F" represent the binary sequences "0000" through "1111". Leading zero digits may be omitted. The radix letter may be omitted for octal values. See Example 8.1.3-1.

Long bits constants have the same format as bits constants except that they are followed by the letter "L".

The bit positions in a bits or long bits may be numbered according to the powers of two they represent if the bits or long bits is considered to be an unsigned integer; see Figure 8.1.3-2.

```
BEGIN "sines"

# The name of the MAINSAIL trigonometric sine procedure
# is "sin".

INITIAL PROCEDURE;
BEGIN
REAL r;
LONG REAL rr;
STRING s,t;

DOB write(logFile,
        "Floating point number (<eol> to stop): ");
    read(cmdFile,s); IF s = "" THEN DONE;
    t := s; read(s,r);
    write(logFile,"Real sin(",r,") = ",sin(r),eol);
    read(t,rr);
    write(logFile,"Long real sin(",rr,") = ",sin(rr),eol);
    END;
END;

END "sines"
```

Example 8.1.2-1. Calculation of Sines

The forms on each line represent the same value.

Binary	Octal	Hexadecimal
'B0	'O0 ('0)	'H0
'B110110	'O66 ('66)	'H36
'B1111111111111111	'O177777 ('177777)	'HFFFF
'B0001101000101011	'O015053 ('15053)	'H1A2B

Example 8.1.3-1. Bits Constants

Descriptions of bits and long bits operations often refer to "corresponding bits" in two bits or long bits values; corresponding bits are the bit positions with the same number in both values.

```
<- LEFT                                  RIGHT ->
(most significant in        (least significant in
 an unsigned integer)        an unsigned integer)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

For example, in the bits value '027, the bits
numbered 0, 1, 2, and 4 are set.  These bits may
also be referred to as the '1, '2, '4, and '20
(or 'H10) bits, respectively.
```

Figure 8.1.3-2. Bit Numbers in a Bits

Examples of the use of the bits and long bits data types are deferred until Section 8.5, where the bits and long bits operators are introduced.

8.2. The Zero Value of a Data Type

Every data type has a special value that is called its "Zero" value. The Zero value is used in various ways. Table 8.2-1 lists the Zeros for all the MAINSAIL data types.

When a module starts execution, every own or outer variable initially has the Zero value of its data type. Thus, for example, the line in the initial procedure of Example 7.2.2-2 that sets the variables a, b, and c to the value 0 is unnecessary, since they already have that value at that point. However, such an initialization is good form, since it makes it clear to the human reader what is going on.

Local variables (other than omitted optional uses and modifies parameters) are not initialized to Zero. It is important to initialize local variables before they are used.

An expression of any data type listed in Table 8.2-1 may be used as the governing expression of an If Statement, WHILE-clause, or UNTIL-clause. The expression is treated like a false boolean expression if it has the Zero value for its data type, and like a true boolean expression if it has any other value. See Example 8.2-2.

boolean:	FALSE
integer:	0
long integer:	0L
real:	0.0
long real:	0.0L
bits:	'0
long bits:	'0L
string:	""
pointer:	NULLPOINTER
address:	NULLADDRESS
charadr:	NULLCHARADR

Although "array" is not really considered a separate data type, it has a Zero of its own:

array:	NULLARRAY

Table 8.2-1. The Zeros of the MAINSAIL Data Types

```
Assume s and t are string variables.  Then
this statement:

    WHILE s NEQ "" DO cWrite(t,cRead(s))

is equivalent to this statement:

    WHILE s DO cWrite(t,cRead(s))

Both do the same thing as:

    t := t & s; s := ""
```

Example 8.2-2. Using a String as the Controlling Expression of a WHILE-Clause

8.3. Conversion Procedures

MAINSAIL is a strongly typed language that does not automatically convert expressions of one data type into expressions of another. It is therefore necessary to have a way of explicitly converting between data types. MAINSAIL provides a set of system procedures called "conversion procedures" to perform this task.

The names of the conversion procedures are shown in Table 8.3-1. Each of the conversion procedures takes a single argument, the expression to be converted, and returns a value, which is the value of the argument "translated" into the desired data type. The exact effect of the translations is described in the "MAINSAIL Language Manual", but the translations usually do what you would expect; for example, "cvi(1.0)", "cvi(1L)", and "cvi('1)" are all equal to the integer 1.

Procedure	Converts to
cvi	integer
cvli	long integer
cvr	real
cvlr	long real
cvb	bits
cvlb	long bits
cvs	string
cva	address
cvc	charadr
cvp	pointer

Table 8.3-1. Names of Conversion Procedures

The conversion procedures are generic, and provide most of the possible conversions from one MAINSAIL data type to a different data type. There are some exceptions; for example, it is not possible to convert directly from a bits to a real or vice versa. See the "MAINSAIL Language Manual" for a table of allowed conversions.

8.4. Reading Numeric Values from cmdFile

Until now, reading an integer i from cmdFile has been accomplished by the sequence "read(cmdFile,s); read(s,i)", where s is a string variable. Alternatively, "cvi(s)" may be used to obtain the integer value of s. Like "read", cvi scans its string argument looking for a numeric value, returning 0 if it finds nothing that looks like a number.

An integer could be read directly from cmdFile with the statement "read(cmdFile,i)", but this has the undesirable property that the portion of the input line following the integer (including the end-of-line character) remains unread from cmdFile. The next string read from cmdFile therefore consists of the (possibly null) remainder of the line from which the integer was read. Therefore, the most usual sequence to read an integer that appears by itself on a line of cmdFile is "read(cmdFile,s); i := cvi(s)".

What does the program of Example 8.4-1 do? Try it out. Does it do what you would expect?

```
BEGIN "wrong"

INITIAL PROCEDURE;
BEGIN
INTEGER i;
STRING s;

write(logFile,"An integer: "); read(cmdFile,i);
write(logFile,"Your integer was ",i,eol,
    "A string: "); read(cmdFile,s);
write(logFile,"Your string was """,s,"""" & eol);
END;

END "wrong"
```

Example 8.4-1. The Wrong Way to Read an Integer from cmdFile

8.5. Tables of Operations

The operators encountered so far include arithmetic operations for integer and real data types, the concatenation operator for strings, comparison operators, and the boolean operators "AND", "OR", and "NOT". In every example of a binary operator encountered so far, the data type of the first operand has been the same as that of the second and (except for the comparison operators) the same as the data type of the result. In fact, the operators "AND", "OR", and "NOT" may take operands of any data type; they treat any non-Zero value as if it were boolean true and any Zero value as false. Other operators require or permit their operands to be of different types. Every MAINSAIL operator is listed in this section, except for the assignment operator, described in Section 10.3.

The tables of operations in this section are adapted from those in the "MAINSAIL Language Manual", which are in a slightly different format. The entry "arith." as a data type means integer, long integer, real, or long real.

MAINSAIL's two unary operators are shown in Table 8.5-1. Each unary operator precedes its operand in source text.

Operator	Operand Data Type	Result Data Type	Description of Result
NOT	any	boolean	true if the operand's value is not the Zero of its data type; false if its value is Zero
-	arith.	same as operand type	arithmetic negation of operand (operand is subtracted from the Zero of the appropriate data type)

Table 8.5-1. Unary Operators

Table 8.5-2 explains all of the binary operators provided by MAINSAIL except the assignment operator. Each binary operator appears in source text between its two operands. An operand may be any expression of the appropriate data type.

Example 8.5-3 shows how the bits operation "IOR" may be considered as the boolean "OR" of corresponding bits if each bit is viewed as a boolean value. Also, "MSK" is to "AND" as "IOR" is to "OR", and "XOR" is to "NEQ" as "IOR" is to "OR". There is no primitive MAINSAIL boolean operation directly corresponding to the bits operation "CLR".

Example 8.5-3 also illustrates the use of control bits in the use of the optional parameter to "cvs" in order to print a bits value in binary, octal, or hexadecimal. See the "MAINSAIL Language Manual" for a full description of cvs.

Example 8.5-4 shows a sample execution of the module of Example 8.5-3. Note how cvb operates on a string containing a single quote character. Note also that the default radix for a "write" to a text file of a bits or long bits value is octal.

Operator	Data Type of First Operand (e1)	Data Type of Second Operand (e2)	Data Type of Result	Description
=	any	same as e1	boolean	true if e1 has the same value as e2
NEQ	any	same as e1	boolean	true if e1 does not have the same value as e2
<	arith., string, address, charadr	same as e1	boolean	for arithmetic data types, true if e1's value is less than e2's. For address and charadr, true if e1 represents a lower address than e2. Strings are compared character by corresponding character until a difference is found or one string runs out; true if the character code of the first non-matching character of e1 is less than the corresponding character of e2, or if e1 is shorter than e2 and the strings match up to the end of e1
LEQ	arith., string, address, charadr	same as e1	boolean	true if e1 < e2 or e1 = e2. Stands for "Less or EQual"

Table 8.5-2. Binary Operators (continued)

>	arith., string, address, charadr	same as e1	boolean	true if e2 < e1
GEQ	arith., string, address, charadr	same as e1	boolean	true if e1 > e2 or e1 = e2. Stands for "Greater or EQual"
OR	any	any	boolean	true if either e1 is not the Zero of its data type or e2 is not the Zero of its type or neither one is Zero
AND	any	any	boolean	false if either e1 or e2 is the Zero of its data type; true if both are non-Zero
TST	bits, long bits	same as e1	boolean	true if any set bit in e1 corresponds to a set bit in e2. Stands for "TeST"
NTST	bits, long bits	same as e1	boolean	true if NOT e1 TST e2
TSTA	bits, long bits	same as e1	boolean	true if the bit corresponding to every set bit in e2 is set in e1
NTSTA	bits, long bits	same as e1	boolean	true if NOT e1 TSTA e2

Table 8.5-2. Binary Operators (continued)

IOR, !	bits, long bits	same as e1	same as e1	each bit in result is set if and only if corresponding bit is set in e1 or e2 or both. Stands for "Inclusive OR". "!" has higher precedence than "IOR", but they are otherwise the same
XOR	bits, long bits	same as e1	same as e1	each bit in result is set if and only if corresponding bit is set in e1 or e2, but not both. Stands for "eXclusive OR"
MSK	bits, long bits	same as e1	same as e1	each bit in result is set only if corresponding bit is set in both e1 and e2. Stands for "MaSK"
CLR	bits, long bits	same as e1	same as e1	each bit in result is set only if corresponding bit in e1 is set and corresponding bit in e2 is clear. Stands for "CLeaR"
SHL	bits, long bits	integer	same as e1	e1 shifted left e2 positions; zeros brought in from right. Not defined if e2 GEQ <number of bits in e1's data type>

Table 8.5-2. Binary Operators (continued)

SHR	bits, long bits	integer	same as e1	e1 shifted right e2 positions; zeros brought in from left. Not defined if e2 GEQ <number of bits in e1's data type>
MIN	arith., string, address, charadr	same as e1	same as e1	the value of e1, if e1 < e2; otherwise the value of e2. Stands for "MINimum"
MAX	arith., string, address, charadr	same as e1	same as e1	the value of e1, if e1 > e2; otherwise the value of e2. Stands for "MAXimum"
+	arith.	same as e1	same as e1	sum of e1 and e2
-	arith.	same as e1	same as e1	difference of e1 and e2
*	arith.	same as e1	same as e1	product of e1 and e2
DIV	integer, long integer	same as e1	same as e1	e1 divided by e2. Integer quotient: remainder is discarded. Not defined if e1 is negative or e2 is not positive
MOD	integer, long integer	same as e1	same as e1	remainder of e1 divided by e2. Not defined if e1 is negative or e2 is not positive

Table 8.5-2. Binary Operators (continued)

/	real, long real	same as e1	same as e1	quotient of e1 and e2
^	arith.	integer, real	same as e1	e1 raised to the power e2. If e2 is real, then e1 must be real or long real
&	string	string	string	string concatenation

Table 8.5-2. Binary Operators (end)

```
BEGIN "bAndB"

BITS PROCEDURE bitByBitIor (BITS b1,b2);
BEGIN
BOOLEAN bit1,bit2;
INTEGER i;
BITS result;

result := 'HFFFF; # First assume all bits set
FOR i := 0 UPTO 15 DOB
    bit1 := b1 TST ('1 SHL i); # Boolean for bit number i
    bit2 := b2 TST ('1 SHL i);
    IF NOT (bit1 OR bit2) THEN # Result bit isn't set
        result := result CLR ('1 SHL i) END;
RETURN(result);
END;



PROCEDURE writeBits (BITS b);
BEGIN
write(logFile,b," (oct) ",cvs(b,hex)," (hex) ",
    cvs(b,binary)," (bin)" & eol);
END;
```

Example 8.5-3. Correspondence Between Bits and Boolean Operations (continued)

```
INITIAL PROCEDURE;
BEGIN
BITS b1,b2;
STRING s;

DOB write(logFile,"Bits value one (<eol> to stop): ");
    read(cmdFile,s);
    IF NOT s THEN DONE; # Equivalent to: IF s = ""...
    b1 := cvb(s);
    write(logFile,"Bits value two: ");
    read(cmdFile,s); b2 := cvb(s);

    # Truncate to sixteen bits, since some machines may
    # allow more bits:
    b1 := b1 MSK 'HFFFF; b2 := b2 MSK 'HFFFF;

    write(logFile,"IOR calculated bit by bit: ");
    writeBits(bitByBitIor(b1,b2));
    write(logFile,"IOR calculated by MAINSAIL: ");
    writeBits(b1 IOR b2) END;
END;

END "bAndB"
```

Example 8.5-3. Correspondence Between Bits and Boolean Operations (end)

```
Bits value one (<eol> to stop): 31<eol>
Bits value two: 13<eol>
IOR calculated bit by bit: 33 (oct) 1B (hex) 11011 (bin)
IOR calculated by MAINSAIL: 33 (oct) 1B (hex) 11011 (bin)
Bits value one (<eol> to stop): 'H01<eol>
Bits value two: 'H34<eol>
IOR calculated bit by bit: 65 (oct) 35 (hex) 110101 (bin)
IOR calculated by MAINSAIL: 65 (oct) 35 (hex) 110101 (bin)
Bits value one (<eol> to stop): 'B11000<eol>
Bits value two: 'B00010<eol>
IOR calculated bit by bit: 32 (oct) 1A (hex) 11010 (bin)
IOR calculated by MAINSAIL: 32 (oct) 1A (hex) 11010 (bin)
Bits value one (<eol> to stop): <eol>
```

Example 8.5-4. Execution of BANDB

8.6. Exercises

Exercise 8-1.

There is no single primitive boolean operation corresponding to the bits operation "CLR" as "OR" corresponds to "IOR", "AND" to "MSK", and "NEQ" to "XOR". Construct a boolean expression describing the boolean operation performed on corresponding bits in a "CLR" operation; i.e., fill in the following table:

The bit at position number n of bits values b1 and b2 is set in the result of:	if and only if:
`b1 IOR b2`	`(b1 SHR n TST '1) OR (b2 SHR n TST '1)`
`b1 MSK b2`	`(b1 SHR n TST '1) AND (b2 SHR n TST '1)`
`b1 XOR b2`	`(b1 SHR n TST '1) NEQ (b2 SHR n TST '1)`
`b1 CLR b2`	`<construct this expression>`

Exercise 8-2.

Write a program that performs unsigned 16-bit addition using the bits data type. You may not convert the bits to an integer and then perform the addition; you must do it bit by bit. Be sure to give some indication if the operation overflows 16 bits.

9. Sequential Input and Output

This chapter describes file input and output: opening, closing, reading, and writing files. Only sequential input and output are discussed; random access to files is deferred until Section 10.11.

9.1. Introduction to MAINSAIL I/O

Unlike many programming languages, MAINSAIL defines input and output in a complete and portable manner. Most file input and output are performed by the system procedures "read" and "write"; however, special forms of I/O exist for some purposes. For example, Section 1.15 of part II of the "MAINSAIL Tutorial" describes input and output of large amounts of data from a single file with a single procedure call; the "MAINSAIL Structure Blaster User's Guide" describes the Structure Blaster, which is used to store and retrieve MAINSAIL data structures in an efficient manner.

9.2. Text Files and Data Files

A file is a collection of data organized serially; it has a well-defined beginning position, though not all files have a well-defined ending position. Every operating system on which MAINSAIL runs provides a system-dependent "file system"; in addition, MAINSAIL supports some files that are independent of the operating system by means of special modules called "device modules" (these special files are documented in the "MAINSAIL Utilities User's Guide"). On some operating systems, MAINSAIL may support additional system-dependent device modules; consult the appropriate operating-system-specific user's guide for details.

Every MAINSAIL file is either a "text file" or a "data file". The units of data in a text file are "character units" (each of which is large enough to hold exactly one character); the units in a data file are usually measured in "storage units" (each MAINSAIL data type occupies one or more storage units); the units for data files opened for a type of I/O called "PDF I/O" ("Portable Data Format"), however, are measured in character units like text files (see Section 10.12). Text files are interpreted by reading the data into string and character (integer) variables; the data in a data file are usually read into variables of the numeric or bits data types, although it is also possible to interpret them as text. Some files must be opened as text files, and others as data; some may treated as either text or data.

The data type specifier for text files is "POINTER(textFile)"; that for data files is "POINTER(dataFile)". The type specifier "POINTER(file)" is used in the declarations of some system procedure parameters to represent either a text or data file; however, the user should not declare a variable to be a "POINTER(file)" (for reasons that will become clear later).

"POINTER(textFile)", "POINTER(dataFile)", and "POINTER(file)" are actually instances of the pointer data type, described in Chapter 11.

9.3. Opening, Closing, Reading, and Writing Files

Before the data in a file may be accessed, the file must be "open". A file may be opened by calling the system procedure "open". "open" is a generic procedure that may operate on either text or data files.

When a program finishes accessing the data in a file, it must close it by calling the system procedure "close". If a program fails to close a file, the file is closed when MAINSAIL exits (provided MAINSAIL exits normally). However, since closing a file may free up memory or other resources associated with an open file, it is a good idea to close files as soon as you are done with them.

When "open" is called, a program must specify whether it intends to use the file for input or for output (or both; see Section 10.11). "write" may be called only for files open for output, "read" only for files open for input.

The predeclared files "logFile" and "cmdFile" are text files that are opened before a user program gains control. logFile is opened for output, cmdFile for input.

The declarations of "open" and "close" are shown in Figure 9.3-1. The file "f" passed to "open" is the variable declared by the user (or predeclared by MAINSAIL) that is used in subsequent calls to "read", "write", and "close". The fileName parameter is the name of the file. The openBits parameter specifies the type of access desired for the file; some of the applicable predefined bits constants are "input", "create", "output", and "errorOK". The fileSize parameter is rarely specified and can be ignored.

A program that opens a text file and copies the contents to logFile is shown in Example 9.3-2.

Note the use of the "!" in specifying both the "input" and "errorOK" bits to "open". The openBits parameter could have been written "input IOR errorOK"; however, when several named bits are specified in a bits parameter, it is customary to use "!" rather than "IOR". The presence of the errorOK bit indicates that if "open" is unable to process the named file, it should return false. If errorOK is not specified, "open" writes a message to logFile indicating that the named file cannot be opened, and prompts repeatedly for a new file name until a file is named that can be opened; if errorOK is not specified, "open" always returns true.

When the end of a text file is reached, "read" fails, since there are no more strings to read from the file. "$gotValue(f)" returns false if the last attempted read from f failed, true otherwise. $gotValue is the usual way to test for end-of-file. The $gotValue test is not precise; see Section 10.13.

```
# open is not really declared twice; it is
# actually a generic procedure.

BOOLEAN PROCEDURE open
    (PRODUCES POINTER(textFile) f;
     STRING                     fileName;
     BITS                       openBits;
     OPTIONAL LONG INTEGER      fileSize);

BOOLEAN PROCEDURE open
    (PRODUCES POINTER(dataFile) f;
     STRING                     fileName;
     BITS                       openBits;
     OPTIONAL LONG INTEGER      fileSize);

PROCEDURE close (MODIFIES POINTER(file) f;
                 OPTIONAL BITS closeBits)
```

Figure 9.3-1. The Declarations of "open" and "close"

Example 9.3-3 shows a program that reads a data file, processes it, and writes the result to another data file. The input data file contains a series of sets of long real numbers to be added. Each set is preceded by an integer specifying the number of long reals. The program terminates when it finds a set with zero long reals. Since "read" returns a Zero of the appropriate data type whenever the end of file has been reached, this ensures that the program will terminate if it accidentally reads beyond the end of the input file.

Example 9.3-3 illustrates the use of the open bits "create" and "prompt". "prompt" specifies that the fileName parameter is actually a prompt to be written to logFile; the real file name is then read from cmdFile. The "create" bit must be specified when opening a new file; the "output" bit must be specified whenever the "create" bit is specified.

Example 9.3-4 shows what an input file to the program of Example 9.3-3 might look like, and the resulting output file. The data files are shown diagrammatically because they would presumably be illegible if interpreted as text files. The amount of storage occupied by each data item is not proportional to the space allotted to it in the diagram; all integers take up the number of storage units required to represent an integer, and all long reals the number of storage units required to represent a long real. These numbers are dependent on the system on which MAINSAIL is running; see Section 10.11.

```
BEGIN "copFil"

INITIAL PROCEDURE;
BEGIN
STRING s;
POINTER(textFile) inputFile;

write(logFile,"Input file name: "); read(cmdFile,s);
IF NOT open(inputFile,s,input!errorOK) THENB
    # Open for input access; return false if can't open
    write(logFile,"Unable to open file ",s,eol);
    RETURN END; # "RETURN" terminates execution of the
                # initial procedure, and consequently of
                # the program
DOB read(inputFile,s);
    IF NOT $gotValue(s) THEN DONE;
        # Have we reached the end of inputFile?
    write(logFile,s,eol) END;
close(inputFile);
END;

END "copFil"
```

Example 9.3-2. Copying a Text File to logFile

```
BEGIN "nums"

INITIAL PROCEDURE;
BEGIN
INTEGER i,count;
LONG REAL r,sum;
POINTER(dataFile) inFile,outFile;

open(inFile,"Input file: ",prompt!input);
open(outFile,"Output file: ",create!prompt!output);
    # Create a new file for output access
DOB read(inFile,count); IF NOT count THEN DONE;
    sum := 0.0L;
    FOR i := 1 UPTO count DOB
        read(inFile,r); sum := sum + r END;
    write(outFile,sum) END;
close(inFile); close(outFile);
write(logFile,"File processed." & eol);
END;

END "nums"
```

Example 9.3-3. The Use of Data Files

```
+-- start of input file
|
v
+---+------+------+-------+---+------+-------+---+
| 3 | 6.0L | 4.2L | -1.8L | 2 | 1.1L | 9.04L | 0 |
+---+------+------+-------+---+------+-------+---+


+-- start of output file
|
v
+------+--------+
| 8.4L | 10.14L |
+------+--------+
```

Example 9.3-4. Sample Input and Output Files for NUMS

9.4. File Names, Logical Names, and I/O Redirection

The MAINSAIL syntax for operating-system-dependent file names is described in the operating-system-specific MAINSAIL user's guide for your system if it differs from the operating system's standard syntax (it usually does not differ). MAINSAIL also makes some guarantees about system-specific file names; these guarantees are described in the "MAINSAIL Language Manual".

9.4.1. Logical Names

In general it is a bad idea to hardwire any file name into your programs. A program designed to be portable should either prompt the user for the names of any files it needs or use a "logical name", i.e., a file name that the program does not expect to be the real file name. Logical file names often (but not always) have a format that would be illegal or unusual for an operating-system-dependent file name.

Real file names may be substituted for logical names in one of three ways:

1. When a program fails to open a file using its logical name, an error message is (by default) written to logFile. The user may then enter the real file name from cmdFile.

2. Before the program is executed, another program in the same MAINSAIL session may establish a correspondence between the logical name and the real file name by calling the procedure "enterLogicalName". When the file is opened, the correspondence is found and the real file name automatically substituted for the logical name.

3. The MAINEX "ENTER" subcommand may be given before the program is executed. The "ENTER" subcommand causes MAINEX to call enterLogicalName with the appropriate arguments.

The declarations of enterLogicalName and its companion procedure, lookupLogicalName, appear in Figure 9.4.1-1. The logicalName parameter is a logical name; enterLogicalName associates it with the real file name trueName, and lookupLogicalName returns the associated real file name, or the null string if there is no associated real file name.

Example 9.4.1-2 is a fragment of a program PROG that uses the logical names "[input file]" and "[output file]". The ".name" suffix on a file variable provides a string that is the name actually used when the file is opened (i.e., the name after logical name substitutions have been performed).

```
STRING
PROCEDURE lookUpLogicalName (STRING logicalName);

PROCEDURE enterLogicalName (STRING logicalName,trueName);
```

Figure 9.4.1-1. Declarations of enterLogicalName and lookupLogicalName

```
BEGIN "prog"

POINTER(textFile) inputFile,outputFile;

PROCEDURE processData;
# Process the input file and write to the output file
        .
        .
        .

INITIAL PROCEDURE;
BEGIN
open(inputFile,"[input file]",input);
write(logFile,"Opened ",inputFile.name,eol);
open(outputFile,"[output file]",create!output);
write(logFile,"Opened ",outputFile.name,eol);
processData;
close(inputFile); close(outputFile);
END;

END "prog"
```

Example 9.4.1-2. Program Fragment That Uses Logical Names

Assuming the operating system does not provide any files actually named "[input file]" and "[output file]", execution of PROG might begin as shown in Example 9.4.1-3. The user substitutes the file name "in.dat" for "[input file]" and "out.dat" for "[output file]".

Instead of allowing a runtime error to occur and specifying the file names to a the "New file name:" prompt, the user may first execute the module SETUP shown in Example 9.4.1-4 to set up the desired logical name correspondences. The substitution set up by enterLogicalName

```
*prog<eol>

Operating system error: file not found

ERROR: Cannot open (<eol> to enter new file name)
    [input file]
Error response: <eol>
New file name: in.dat<eol>
Opened in.dat

Operating system error: illegal file name

ERROR: Cannot open (<eol> to enter new file name)
    [output file]
Error response: <eol>
New file name: out.dat<eol>
Opened out.dat
          .
          .
          .
```

Example 9.4.1-3. Execution of PROG

endures until MAINSAIL returns to the operating system, whereas file names given in response to a "New file name:" prompt are not subsequently substituted for the logical names that produced the original error. For example, if PROG were run again after the execution of Example 9.4.1-3, MAINSAIL would once again be unable to open "[input file]" and "[output file]" and would therefore issue error messages and prompt for new file names. If PROG were run again after the execution of Example 9.4.1-5, however, MAINSAIL would again automatically substitute the file names established by SETUP, and no error message would be issued.

MAINEX provides a facility that performs the function of the module SETUP of Example 9.4.1-4. The "ENTER" subcommand (like other MAINEX subcommands) may be specified in response the ">" subcommand prompt if a line typed to the "*" prompt ends in a comma. The "ENTER" subcommand sets up a logical name correspondence by calling enterLogicalName; the PROG logical names are set up in this way in Example 9.4.1-6.

"SEARCHPATH" is a MAINEX subcommand that allows file name substitutions to be based on a pattern. If many similar file name substitutions are to be made, "SEARCHPATH" is better than "ENTER".

```
BEGIN "setUp"

INITIAL PROCEDURE;
BEGIN
STRING s;

write(logFile,"Correspondence for ""[input file]"": ");
read(cmdFile,s); enterLogicalName("[input file]",s);
write(logFile,"Correspondence for ""[output file]"": ");
read(cmdFile,s); enterLogicalName("[output file]",s);
END;

END "setUp"
```

Example 9.4.1-4. A Module to Set Up Logical Name Correspondences

```
*setup<eol>
Correspondence for "[input file]": in.dat<eol>
Correspondence for "[output file]": out.dat<eol>
*prog<eol>
Opened in.dat
Opened out.dat
          .
          .
          .
```

Example 9.4.1-5. Execution of PROG with Automatic Logical Name Substitution

MAINEX subcommands are described in the "MAINSAIL Utilities User's Guide".

Note that a logical name need not have an illegal or unusual file name format. For example, the MAINSAIL text editor, MAINEDIT, attempts to open a file named "eparms" from which it reads directions for setting up the text editing environment. Often such a file exists, but sometimes you may, for example, want to use somebody else's text editing environment. In such a case, you would do as shown in Example 9.4.1-7 (EDIT is the module that invokes MAINEDIT).

```
*prog,<eol>                          |
 ----------                          |
>enter [input file] in.dat<eol>      |
 ------------------------------      |
>enter [output file] out.dat<eol>    |
 --------------------------------    |
><eol>                               | A blank line to the
 -----                               | ">" prompt terminates
Opened in.dat                        | subcommand mode.
Opened out.dat                       |
         .                           |
         .                           |
         .                           |
```

Example 9.4.1-6. Use of the MAINEX "ENTER" Subcommand

```
*edit,<eol>
>enter eparms someone-elses-eparms<eol>
><eol>
         .
         .
         .
```

Example 9.4.1-7. A Logical Name Substitution for MAINEDIT's "eparms" File

The "ENTER" and "SEARCHPATH" subcommands are frequently used in a MAINSAIL bootstrap to set up logical names used by all the programs to be executed by that bootstrap; see Chapter 20.

9.4.2. Redirection of cmdFile and logFile

The files cmdFile and logFile are normally associated with terminal input and output (or whatever the operating system provides as the main input and output channels), respectively. MAINEX provides subcommands to specify that cmdFile input is to be read from or logFile output written to some other file. As soon as subcommand mode is exited, the "CMDFILE" and "LOGFILE" subcommands take effect. For example, assume a file "cmd" contains what is shown in Example 9.4.2-1. The modules ACKER and CALC are those of Examples 7.4-1 and 7.2.2-2.

```
acker                   |
3                       |
3                       |
calc                    |
1 + 4                   | The blank line at the end
a = b + 2               | causes MAINSAIL to return
s                       | to the operating system
q                       | when the file's end is
                        | reached.
```

Example 9.4.2-1. A Sample Command File

The session of Example 9.4.2-2 shows the redirection of cmdFile and logFile. The resulting logFile "log" appears in Example 9.4.2-3. Note that the commands (including end-of-line characters) of "cmd" are not echoed into "log" (you may cause the contents of cmdFile to be echoed to logFile; see the description of the "ECHOCMDFILE" and "ECHOIFREDIRECTED" MAINEX subcommands in the "MAINSAIL Utilities User's Guide").

```
*,<eol>
>cmdfile cmd<eol>
>logfile log<eol>
><eol>
(the operating system command
 processor prints its own prompt here)
```

Example 9.4.2-2. Use of the "CMDFILE" and "LOGFILE" Subcommands

```
*m: n: ack(3,3) = 61
*CALC command ('Q' to quit): 5
CALC command ('Q' to quit): 2
CALC command ('Q' to quit): A: 2  B: 0  C: 0
CALC command ('Q' to quit): *
```

Example 9.4.2-3. A Redirected logFile

9.5. The File "TTY" and the System Procedures ttyRead, ttyWrite, and ttycWrite

The files cmdFile and logFile are originally associated with your terminal by being opened with the file name "TTY" (or "tty"). MAINSAIL recognizes this as a special file name. A text file with the string "TTY" provided as the fileName parameter to "open" is associated with the terminal keyboard if opened for input or with the terminal screen if opened for output (or whatever the operating system provides in the way of primary input and output files).

For example, the MAINSAIL utility COPIER, which copies text files, may be made to display a file on the user's terminal if "TTY" is specified for the output file name. See Example 9.5-1 and the description of COPIER in the "MAINSAIL Utilities User's Guide".

```
*copier<eol>
Text File Copier
Input file (just <eol> to stop): cmd<eol>
Output file: tty<eol>
acker
3
3
calc
1 + 4
a = b + 2
s
q

Input file (just <eol> to stop): <eol>
*
```

Example 9.5-1. Use of COPIER and the File "TTY"

The system procedures ttyRead, ttycWrite, and ttyWrite provide direct access to the file "TTY", regardless of how cmdFile and logFile may have been redirected. Their declarations are shown in Figure 9.5-2. ttyRead returns the next string typed from the terminal keyboard; ttycWrite writes a single character to the terminal screen, and ttyWrite writes any of the values accepted by "write" when it writes to a file or string.

Because ttyRead, ttycWrite, and ttyWrite cannot be redirected from the terminal within MAINSAIL, it is usually preferable to use input from cmdFile and output to logFile. Example 16.4-1 contains a use of ttyRead and ttyWrite, since it is not expected that the interactive

```
STRING PROCEDURE ttyRead;

PROCEDURE ttycWrite (REPEATABLE INTEGER char);

GENERIC PROCEDURE ttyWrite ...
```

Figure 9.5-2. Declarations of ttyRead, ttycWrite, and ttyWrite

display-oriented text editor program of that example would be invoked from a redirected cmdFile.

9.6. alterOK

MAINSAIL guarantees not to replace an existing file without warning. Therefore, when a file is opened with the "create" open bit and a file with the specified name already exists, MAINSAIL ordinarily writes a message to logFile asking whether you really want to replace the existing file, and awaits your reply from cmdFile (this dialogue does not take place on operating systems that provide "version numbers" for files so that the creation of a new file does not actually overwrite an old file of the same name). The open bit alterOK tells MAINSAIL to suppress this dialogue and overwrite the named file without warning. alterOK is often used when a temporary output file or log file is created. The program fragment of Example 9.6-1 uses the alterOK bit in this way.

```
POINTER(textFile) inFile,outFile;

open(inFile,"Input file: ",input!prompt);
open(outFile,inFile.name & ".log",create!output!alterOK);
```

Example 9.6-1. Use of the alterOK Open Bit

9.7. Exercises

Exercise 9-1.

Add two commands to the program CALC of Example 7.2.2-2:

- The "F" command, which specifies the name of a file from which subsequent commands are to be read. Commands read are to be echoed to logFile. When the end of the specified file is reached, commands are again read from cmdFile (unless "Q" was encountered in the "F" file). The format is "F <file name>". Do not permit an "F" command to appear in an "F" file.

- The "T" command, which specifies the name of a file to which subsequent output is to be written. Output to the "T" file should also be written to logFile, and input (whether from cmdFile or from an "F" file) should be echoed to the "T" file. The format is "T <file name>". If no file name is given (i.e., the "T" is alone on a line), the "T" file, if any, should be closed and subsequent output written only to logFile.

Exercise 9-2.

Write a calculator program that reads a data file composed of pairs of reals and bits. The first value of the pair is a real; the second value, the bits, indicates what to do with the real. Table 9.7-1 shows how the bits value is to be interpreted. The real accumulator initially contains 0.0.

Every bits command must have the '4 bit set. When a bits value without this bit set is read, the program should print the current value of the accumulator, then stop.

The rightmost two bits of the bits value tell what to do with the real value just read:

'0	add the real value to the accumulator
'1	subtract the real value from the accumulator
'2	multiply the accumulator by the real value
'3	divide the accumulator by the real value

If the '10 bit is set, write the resulting accumulator value to logFile. If the '20 bit is set (or both the '10 and '20 bits are set), write the current accumulator value and the operation to be performed as well as the resulting accumulator value.

Table 9.7-1. Interpretation of Data File Commands

10. More on Expressions and Strings; the Expression, Case, and Empty Statements; Random Access to Files

The expressions, statements, and system procedures introduced in this chapter are less "fundamental" than those introduced in previous chapters. However, many of them provide syntactic convenience that greatly contributes to the readability of MAINSAIL programs.

Random access to files is not a syntactic convenience, but rather an essential input/output technique.

10.1. Procedure "BEGIN" and "END"

If a procedure consists of a single statement and has no local declarations, the "BEGIN" and "END" bracketing the body of the procedure may be omitted. See Example 10.2-1.

10.2. The If Expression

The keyword "IF" may begin an expression as well as a statement. The form of an If Expression is:

```
IF <expression one> THEN <expression two> ELSE
  <expression three>
```

If expression one has a non-Zero value for its data type, the value of the If Expression is the value of expression two; otherwise, it is the value of expression three. Expressions two and three must have the same data type, which need not be the same as the data type of expression one.

If Expressions may be nested. The abbreviations "EF" and "EL" may be used in If Expressions, just as in If Statements.

The procedure "ack" of Example 7.4-1 can be written to use an If Expression inside its Return Statement, provided the code to check recursion depth is removed. See Example 10.2-1; compare the two versions of ack.

```
INTEGER PROCEDURE ack (INTEGER m,n);
RETURN(
    IF NOT m THEN n + 1
    EF NOT n THEN ack(m - 1,1)
    EL ack(m - 1,ack(m,n - 1)));
```

Example 10.2-1. An If Expression inside a Return Statement

10.3. The Assignment Expression

An Assignment Expression has the same form as an Assignment Statement. The value of the expression is the same as the value assigned to the variable on the left side of the assignment operator and is of the same data type as that variable.

Assignment Expressions permit the use of "chain" or "multiple" assignments; see Example 10.3-1.

```
If a, b, and c are all integer variables, then:

    a := b := c := 0

sets all three variables to have the value 0.
The statement could also be written as:

    a := (b := (c := 0))
```

Example 10.3-1. Chain Assignments Using the Assignment Expression

The precedence of the assignment operator in an Assignment Expression is not quite the same as in an Assignment Statement. In the Assignment Statement, the assignment operator has a lower precedence than any other operator; in an Assignment Expression, it has a higher precedence than the comparison operators, "NOT", "AND", and "OR". See Example 10.3-2. If you find the precedence rules too confusing to remember, you may use redundant parentheses to specify the order of evaluation of expressions.

```
    IF v := e1 OR e2 THEN ...

is equivalent to:

    IF (v := e1) OR e2 THEN ...

NOT equivalent to:

    IF v := (e1 OR e2) THEN ...

But the statement:

    v := e1 OR e2;

is equivalent to:

    v := (e1 OR e2);
```

Example 10.3-2. Precedence of the Assignment Operator in Expressions and Statements

10.4. Short-Circuit Evaluation

The operators "AND" and "OR" evaluate their second operands only if the first operand is non-Zero or Zero, respectively. That is, the "AND" or "OR" expression is evaluated only far enough to determine its value. If the first operand to "AND" is false, the entire expression is necessarily false, so the second operand need not be evaluated; likewise, if the first operand to "OR" is true, the entire expression is true, regardless of the value of the second operand. This property of "AND" and "OR" is referred to as "short-circuit evaluation". See Example 10.4-1.

The short-circuit evaluation of "AND" is particularly useful in connection with pointers; see Example 11.3-1.

10.5. Substrings

A substring is a string expression which, as the name implies, is calculated as a sequence of characters contained within another string. A substring is specified by giving the string of which the substring is to be taken, the starting position, and the ending position in the form:

```
<string> [ <start position> TO <end position> ]
```

```
Assume a procedure "lookUp" should be called for a
string s only if s is not the null string.  If s is
"", or if lookUp returns a Zero value, then action
A should be taken; otherwise, some other action B
should be taken.  The code to do this looks like:

     IF s AND lookUp(s) THENB
         ... code for action B... END
     EB  ... code for action A... END

Note that lookUp is not called if s is "".
```

Example 10.4-1. The Use of Short-Circuit Evaluation

or the form:

```
    <string> [ <start position> FOR <number of characters> ]
```

The string may be any string expression. The positions are integers; the first position in a string is number one. The first form of substring specifies the end position absolutely; the second specifies it relative to the start position. Example 10.5-1 shows an example of the second form. Other examples of both forms appear in the "MAINSAIL Language Manual" and Example 16.4-1.

```
One way to see if a string begins with a certain
sequence of characters is to use a substring.  The
following code checks to see whether a string s
begins with "new file" (ignoring case):

IF cvu(s)[1 FOR length("NEW FILE")] = "NEW FILE"...
```

Example 10.5-1. A Common Use of Substrings

"INF" is a special integer expression that may appear only within substring brackets. It represents the length of the string of which the substring is taken. For example, "s[1 TO INF - 2]" represents the string that is s with its last two characters removed.

10.6. String Comparison

Strings may be (and frequently are) compared with the standard comparison operators "=", "NEQ", ">", "<", "GEQ", and "LEQ". However, there are some circumstances in which it is more efficient to call the system procedure "compare" or the system procedure "equ".

"compare(r,s)", where r and s are strings, returns -1 if r < s, 0 if r = s, and 1 if r > s. If a program needs to do something different in each of the three cases, compare may be called just once, whereas at least two of the standard comparison operators would have to be called to achieve the same end. See Example 10.6-1.

```
Assume r and s are strings and i an integer variable.
Then the code:

    IF (i := compare(r,s)) < 0 THENB
        ... code for r < s... END
    EF i = 0 THENB
        ... code for r = s... END
    EB  ... code for r > s... END

achieves the same thing as:

    IF r < s THENB
        ... code for r < s... END
    EF r = s THENB
        ... code for r = s... END
    EB  ... code for r > s... END

However, the first example calls compare only once,
whereas the second example uses "<" once and "=" once.
Since "compare", "<", and "=" all result in approximately
the same execution overhead, the first example is more
efficient.  If you feel that the second example is more
legible and that execution efficiency is not important,
then use the second form; otherwise, use the first form.
```

Example 10.6-1. The Procedure "compare" and the Comparison Operators

"compare(r,s,upperCase)" performs a caseless comparison by first converting r and s to upper case. It is more efficient than doing the case conversion explicitly, i.e., than "compare(cvu(r),cvu(s))".

The procedure equ checks for equality only. "equ(r,s)" returns true if "compare(r,s)" returns 0; "equ(r,s,upperCase)" returns true if "compare(r,s,upperCase)" returns 0. equ is not more efficient than "=" unless the upperCase bit is specified. Most programmers prefer "=" to equ if the upperCase option is not specified, since they feel the former is easier to understand.

Using equ, the code in Example 10.5-1 could be written more efficiently as:

```
IF equ(s[1 FOR length("NEW FILE")],"NEW FILE",upperCase)...
```

10.7. The Procedure "scan"

The procedure "scan" is used to remove a prefix of a string based on the characters in the string. The form of "scan" in which the scanCtrl parameter is a string is the only form described here. For some purposes, the forms in which the scanCtrl is an integer or a bits are more efficient; these forms are described in the "MAINSAIL Language Manual". "scan" may be used to read characters from a file as well as string; details may be found in the "MAINSAIL Language Manual".

The declaration of the form of "scan" discussed here is shown in Figure 10.7-1.

```
STRING PROCEDURE scan (MODIFIES STRING source;
                       STRING scanCtrl;
                       OPTIONAL BITS ctrlBits;
                       PRODUCES OPTIONAL INTEGER brkChr)

"scan" is actually a generic procedure.  Consult the
"MAINSAIL Language Manual" for details.
```

Figure 10.7-1. Declaration of the Procedure "scan"

If neither optional parameter is specified, "scan" breaks the source string into two parts. It looks for the first character in source that is the same as one of the characters in scanCtrl; this character is called the "break character". The portion of source up to the break character is returned by "scan"; the remainder of the string is left in the changed value of source. For example, if the string variable s has the value:

```
"Hello, there"
```

then "scan(s,",")" returns "Hello" and changes the value of s to:

```
", there"
```

The break character is returned in the parameter brkChr. In this case, the break character is ",". If no character is found that matches a character in scanCtrl, "scan" returns the entire source string, sets source to "", and returns -1 for brkChr.

The parameter ctrlBits may be used to specify the options shown in Figure 10.7-2. Other options exist but are more rarely used; see the "MAINSAIL Language Manual".

Control Bit Name	Meaning
proceed	Instead of scanning up to a break character that is in scanCtrl, "scan" scans for the first character NOT in scanCtrl.
discard	Remove the break character from source before returning.
append	Remove the break character from source and append it to the returned string.
omit	Discard characters scanned; i.e., the result string is always "". This is more efficient if the result string is not to be used.

Figure 10.7-2. Named Control Bits for "scan"

Using "scan", the procedure "getToken" of Example 7.2.2-2 could be rewritten as shown in Example 10.7-3. Note that the second call to "scan" removes no characters from s and returns the null string if s does not begin with a digit.

10.8. The Expression Statement

The Expression Statement, like the Assignment Statement, is a form that may be used either as an expression or a statement. The form of an Expression Statement is:

```
<variable> <dotted operator> <expression>
```

```
STRING PROCEDURE getToken (MODIFIES STRING s);
# Remove the next thing from s.
BEGIN
STRING t;

scan(s," " & tab,proceed!omit); # Remove leading blanks
                                # and tabs
# Everything but integers is one character:
RETURN(
    IF NOT s THEN "" # End of string
    EF t := scan(s,"0123456789",proceed) THEN t
    EL cvcs(cRead(s)));
END;
```

Example 10.7-3. The Use of "scan"

The variable and the expression must be of the same data type. The dotted operator is any of the operators in Table 10.8-1, preceded by a dot (".").

```
Operators on the same line have the same precedence.
The first line has the lowest (least binding) precedence,
the last line the highest:

MIN   MAX
+    - (binary)   IOR   XOR   MSK   CLR
*    /   &   DIV   MOD   SHL   SHR
!    ^
- (unary)
```

Table 10.8-1. Operators That May Be Dotted

An Expression Statement of the form "v .op e" is equivalent to the Assignment Statement (or Assignment Expression) "v := v op e". The Expression Statement is used primarily for syntactic convenience; however, it can also produce more efficient code if the variable v is not a simple variable (e.g., an array element or field variable; see Chapter 11). Like its equivalent Assignment Expression, the Expression Statement has the value of variable being assigned to when used as an expression.

As an example, "i .+ 1" is equivalent to "i := i + 1". Numerous examples appear throughout the remainder of this tutorial.

10.9. The Case Statement

The Case Statement is used to select one of series of statements based on the value of an integer expression. If there are many expressions, it is usually more efficient (and more readable) to use a Case Statement than an equivalent series of nested If Statements.

The Case Statement is introduced by:

```
CASE <integer expression> OFB
```

which is followed by case selectors, which are integer constant values or integer ranges in brackets, intermixed with statements. If the integer expression has the value in a given selector, the statement immediately following the selector is executed, and then the Case Statement is terminated (unlike the "switch" statement of C, in which the next statement is executed unless the selected statement is terminated by a "break"). The Case Statement is terminated by the keyword "END".

The If Statement outlined in Example 10.6-1 could be rewritten as a Case Statement as shown in Example 10.9-1.

```
CASE compare(r,s) OFB
    [-1] BEGIN
        ... code for r < s... END;
    [0] BEGIN
        ... code for r = s... END;
    [1] BEGIN
        ... code for r > s... END;
    END
```

Example 10.9-1. A Case Statement for String Comparison

A Case Selector consisting of an empty pair of brackets is called the "catch-all" or "default" selector. The statement following the catch-all selector is executed if the selecting expression does not match any of the other selectors. If there is no catch-all selector and the selecting statement does not match any of the selectors, a runtime error occurs.

The program of Example 10.9-2 counts the number of digits, white space characters, and other characters in an input file. Note that since eol, tab, and eop are all one-character strings, the first character of each is the same as its last character. For historical reasons, XIDAK programmers usually use "last(eol)" but "first(tab)" or "first(eop)" to represent the characters in these one-character strings.

```
BEGIN "countr"

INITIAL PROCEDURE;
BEGIN
INTEGER numDigits,numWhite,numOther;
POINTER(textFile) f;

numDigits := numWhite := numOther := 0;
open(f,"Input file: ",input!prompt);
DO  CASE cRead(f) OFB
        [-1] DONE; # -1 means end-of-file
        ['0' TO '9']
            numDigits .+ 1;
        [last(eol)] [' '] [first(tab)] [first(eop)]
            numWhite .+ 1;
        [ ] numOther .+ 1;
        END;
close(f);
write(logFile,"Digits: ",numDigits,eol &
    "White space characters: ",numWhite,eol &
    "Other characters: ",numOther,eol &
    "Total characters: ",
    numDigits + numWhite + numOther,eol);
END;

END "countr"
```

Example 10.9-2. Use of a Case Statement

10.10. The Empty Statement

The Empty Statement consists of nothing at all and performs no action. Empty Statements have been used in the programming examples in this tutorial wherever a semicolon precedes an "END". In the style used in this document, a semicolon precedes an "END" whenever the "END" is on the following line, and the semicolon is omitted whenever the "END" is on the same line. Since a semicolon always separates two statements, and no statement begins with the keyword "END", there must be an Empty Statement between the semicolon and the "END". Example 10.10-1 shows a program fragment that includes an Empty Statement.

```
IF j := process(i) THENB
    write(logFile,"process(",i,") = ",j,eol); i .+ 1;
    # The Empty Statement is here.  If the preceding
    # line did not end with a semicolon, there would
    # be no Empty Statement, but the meaning of the
    # program would be unchanged.
    END
```

Example 10.10-1. The Use of an Empty Statement

The presence or absence of an Empty Statement does not affect the meaning of a program. Empty Statements may appear in places other than before the keyword "END", but that is the most common place for them. A simple rule to remember is that a semicolon may always appear before the keyword "END", but is always optional.

10.11. Random File Access

Every datum written to or read from a file has a particular position in the file. File positions are designated by long integers; the first position in a file is 0L. In a text file, one character is stored at each position; in a data file, one "storage unit" is stored at each position (except for files opened for PDF I/O, explained below). The size of storage units varies from system to system, but every MAINSAIL data type occupies an integral number of storage units. The number of bits in a storage unit is available to a program as the integer constant "$bitsPerStorageUnit".

For every data type in MAINSAIL, a corresponding integer "type code" is predefined. The names of the type codes appear in Table 10.11-1. The number of storage units occupied by a value of a given data may be found by calling the procedure "size"; for example, the number of storage units occupied by an integer is given by "size(integerCode)".

Storage units and data type sizes are explained in detail in Chapter 18.

The position on each "read" from or "write" to a file is incremented by the number of characters or storage units read or written. When a file is opened for sequential access, this is the only control you have over file positions. When a file is opened for random access, however, you

```
booleanCode
integerCode
longIntegerCode
realCode
longRealCode
bitsCode
longBitsCode
stringCode
addressCode
charadrCode
pointerCode
```

Table 10.11-1. Names of the MAINSAIL Type Codes

may explicitly record or change the current file position, so that the next "read" or "write" occurs at a specified place in the file.

The procedures "setPos", "getPos", and "relPos" are used to manipulate file positions. Their declarations are shown in Table 10.11-2. setPos sets the current position of the file f to be n. If, for some reason (e.g., the position specified is beyond the end of the file), it is not possible to set the position to be n, setPos returns false; otherwise it returns true. In addition, if it returns false and the bit errorOK is not set in the ctrlBits parameter, an error message is issued to logFile. getPos returns the current file position associated with f. "relPos(f,n)" is equivalent to "setPos(f,getPos(f) + cvli(n))".

```
BOOLEAN PROCEDURE setPos (POINTER(file) f;
                          OPTIONAL LONG INTEGER n;
                          OPTIONAL BITS ctrlBits);

LONG INTEGER PROCEDURE getPos (POINTER(file) f);

BOOLEAN PROCEDURE relPos (POINTER(file) f; INTEGER n;
                          OPTIONAL BITS ctrlBits);
```

Table 10.11-2. Declarations of setPos, relPos, and getPos

Example 10.11-4 shows a program that maintains a symbol table or primitive database as a data file. Each record in the database has a string name and consists of a string that represents the data in the record (also called the "value" of the record). The records may be created or examined by the user with commands read from cmdFile.

In order to speed lookup of the records within the file, each record name has associated with it an integer "hash code". Records with similar hash codes are stored on the same list (or in the same "hash bucket"); the whole data structure is called a "hash table". The purpose of a hash table is to shorten searches for a given item through a data structure representing a set of items. Instead of maintaining a single list through which to search, N (the "number of hash buckets") lists are maintained, so that (if items are distributed evenly among the buckets) searches take about 1/N as long as through a single list. The hash bucket in which to store or search for a given item is computed from some characteristic of the item (the item's "key") by a function called a "hash function" (the procedure "hash" in Example 10.11-4). The best hash functions produce values distributed evenly among the hash buckets when given a typical mixture of keys.

The format of the file is shown in Figure 10.11-3.

The system procedure "confirm" writes its argument to logFile as a prompt and accepts a "yes" or "no" answer from cmdFile. If the user types something other than "yes" or "no" (or an abbreviation thereof), "confirm" reprompts until an acceptable answer is given. The procedure "errMsg" writes an error message to logFile, then prompts with the standard "Error response:" prompt. It is used by most MAINSAIL utilities to indicate an error condition. More detailed descriptions of these procedures may be found in the "MAINSAIL Language Manual".

```
The first thing in the file is an integer, which is the
number of hash buckets in the file.

The next thing in the file is the null record used to
terminate hash lists.  It consists of a single long
integer, 0L.  An algorithm that traverses a hash list
may therefore terminate by checking whether the nextRec
field (which is the first field) of the record is 0L; if
so, the current record is the null record and contains no
data.

The next thing in the file is the current end-of-file
position.  New records are created at this position, i.e.,
they are added at the end of the file.

The next thing in the file is a series of N long integers,
where N is the number of hash buckets.  Each long integer
is the file position of the first record in the
corresponding bucket.  If a given hash bucket is empty,
the long integer is the position of the null record.

The next thing in the file is the data records themselves.

The information in each record is stored as the
following sequence of values:

+---------+---------+------------+---------+------------+
| nextRec | nameLen | name chars | dataLen | data chars |
+---------+---------+------------+---------+------------+

nextRec is a long integer, representing the file position
of the next record in this hash bucket.  It points at the
null record if this is the last record in the list.

nameLen is the number of characters in the record name;
it is an integer.

name chars are the characters in the name of the record,
stored as individual integers.

dataLen and data chars are the length and characters of
the data string, respectively.
```

Figure 10.11-3. The Format of Hash Lists and Records Used by SYMTAB

```
BEGIN "symTab"

# Maintains a symbol table or primitive database in the
# form of a random-access data file.
POINTER(dataFile) f;     # The database file
INTEGER numBuckets;      # How many buckets in the file

DEFINE numBucketsPos = 0L;
DEFINE nullRecordPos =
    numBucketsPos + cvli(size(integerCode));
DEFINE eofPosPos =
    nullRecordPos + cvli(size(longIntegerCode));
DEFINE firstBucketPos =
    eofPosPos + cvli(size(longIntegerCode));

BOOLEAN PROCEDURE createNewDataBase (STRING name);
# Return true if successful creation.
BEGIN
INTEGER i;
LONG INTEGER eofPos;
STRING s;
```

Example 10.11-4. The Use of a Random-Access Data File (continued)

```
IF NOT confirm("Create new database file " & name) THEN
    RETURN(FALSE);
IF NOT open(f,name,create!input!output!random!errorOK)
    THENB
    errMsg("Couldn't create",name); RETURN(FALSE) END;
setPos(f,nullRecordPos); write(f,0L); # Create null rec.
write(logFile,"Number of hash buckets to use in file ",
    name," (<eol> for 131): "); # 131 is a good number
read(cmdFile,s);
numBuckets := IF NOT s THEN 131 EL cvi(s);
IF numBuckets < 1 OR numBuckets > 1000 THENB
    # Sensible numBuckets?
    errMsg("Bad number of buckets " & s,eol &
        "Should be 1 - 1000"); RETURN(FALSE) END;
setPos(f,numBucketsPos); write(f,numBuckets);
# Now initialize all the buckets to be empty:
setPos(f,firstBucketPos);
FOR i := 1 UPTO numBuckets DO write(f,nullRecordPos);
eofPos := getPos(f); setPos(f,eofPosPos);
write(f,eofPos); # eofPos is current end-of-file position
RETURN(TRUE);
END;



INTEGER PROCEDURE hash (STRING s);
# Returns a value in the range 0 to numBuckets - 1
BEGIN
INTEGER h,i,j;
i := (h := length(s)) MIN 4; j := 1;
WHILE (i .- 1) GEQ 0 DO h .+ cRead(s) * (j .+ 2);
RETURN(h MOD numBuckets) END;



STRING PROCEDURE getString (INTEGER numChars);
# Read the next numChars integers from the file into a
# string.
BEGIN
INTEGER ch;
STRING s;
```

Example 10.11-4. The Use of a Random-Access Data File (continued)

```
s := "";
WHILE (numChars .- 1) GEQ 0 DOB
    read(f,ch); cWrite(s,ch) END;
RETURN(s);
END;



LONG INTEGER PROCEDURE bucketPos (INTEGER hashCode);
# Return the position of the start of the hash list with
# hash code hashCode.
RETURN(firstBucketPos +
    cvli(hashCode * size(longIntegerCode)));



BOOLEAN PROCEDURE lookup
    (STRING recName; PRODUCES OPTIONAL STRING recVal);
# Return true if record recName is found, or if recName
# is "" (illegal record name)
BEGIN
INTEGER nameLen,valLen;
LONG INTEGER nextPos;

IF NOT recName THENB
    errMsg("Null record name"); recVal := "";
    RETURN(TRUE) END; # Act as if we found it

# Position to hash list for this record name:
setPos(f,bucketPos(hash(recName))); read(f,nextPos);
setPos(f,nextPos); # Pos of first record in list
DOB read(f,nextPos);
    IF NOT nextPos THEN RETURN(FALSE); # End of this list
    read(f,nameLen);
    IF getString(nameLen) NEQ recName THENB
        setPos(f,nextPos); CONTINUE END;
    read(f,valLen); recVal := getString(valLen);
    RETURN(TRUE) END;
END;
```

Example 10.11-4. The Use of a Random-Access Data File (continued)

```
PROCEDURE writeRecord (STRING recName,recVal);
# Write the new record at the current end-of-file
# position.
BEGIN
LONG INTEGER eofPos,listPos;

setPos(f,eofPosPos); read(f,eofPos);
# Insert the record at the head of the hash list:
setPos(f,bucketPos(hash(recName)));
# Overwrite the head of the list position:
read(f,listPos); relPos(f,- size(longIntegerCode));
write(f,eofPos); setPos(f,eofPos);
write(f,listPos,length(recName));
WHILE recName DO write(f,cRead(recName));
write(f,length(recVal));
WHILE recVal DO write(f,cRead(recVal));
eofPos := getPos(f); setPos(f,eofPosPos); write(f,eofPos);
END;



PROCEDURE createRecord (STRING s);
BEGIN
STRING recVal,t;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF lookup(s) THENB
    errMsg("Record already exists:",s); RETURN END;
# Now read the record value from cmdFile:
write(logFile,
    "Enter record value; end with blank line" & eol);
recVal := "";
DOB read(cmdFile,t); IF NOT t THEN DONE;
    write(recVal,t,eol); # Same as "recVal .& (t & eol)"
    END;
writeRecord(s,recVal); # Now write it into the file
END;
```

Example 10.11-4. The Use of a Random-Access Data File (continued)

```
PROCEDURE lookupRecord (STRING s);
BEGIN
STRING recVal;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF NOT lookup(s,recVal) THEN errMsg("No such record:",s)
EL write(logFile,recVal);
END;



PROCEDURE showRecords;
BEGIN
INTEGER i,nameLen;
LONG INTEGER nextPos;

FOR i := 0 UPTO numBuckets - 1 DOB
    setPos(f,bucketPos(i)); read(f,nextPos);
    setPos(f,nextPos); # Pos of first record in list
    DOB read(f,nextPos); IF NOT nextPos THEN DONE;
        read(f,nameLen);
        write(logFile,getString(nameLen),eol);
        setPos(f,nextPos) END END;
END;
```

Example 10.11-4. The Use of a Random-Access Data File (continued)

```
BOOLEAN PROCEDURE processCommand (STRING s);
BEGIN
# Return false if s is the quit command, true otherwise.
# The commands are "Q" (quit), "C" (create a new record),
# "S" (show names of all existing records), and "L" (look
# up an existing record).  "C" and "L" commands are
# followed by the record name.
s := cvu(s); # So we don't have to worry about case
CASE cRead(s) OFB
    [-1]  ; # Do nothing if blank line
    ['Q'] RETURN(FALSE);
    ['S'] showRecords;
    ['C'] createRecord(s);
    ['L'] lookupRecord(s);
    ['?'] ['H'] write(logFile,
        "Q              to quit" & eol &
        "S              show names of all records" & eol &
        "C recName      create record recName" & eol &
        "L recName      look up record recName" & eol &
        "? or H         get this message" & eol);
    [ ] write(logFile,
            "Invalid command (? for help)" & eol);
    END;
RETURN(TRUE);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

DOB write(logFile,"Database file name: ");
    read(cmdFile,s) END
    UNTIL open(f,s,random!input!output!errorOK) OR
        createNewDataBase(s);
    # Note use of short-circuit evaluation:
    # createNewDataBase is called only if open fails

setPos(f,numBucketsPos);
read(f,numBuckets); # Get the number of buckets
```

Example 10.11-4. The Use of a Random-Access Data File (continued)

```
DOB write(logFile,"Command: "); read(cmdFile,s) END
    UNTIL NOT processCommand(s);

close(f);
END;

END "symTab"
```

Example 10.11-4. The Use of a Random-Access Data File (end)

10.12. PDF I/O

By preceding a file name with "PDF" and the device module prefix character (defined as $devModBrk, '>' on most systems), or by including the $pdf bit in the call to open, a file can be opened for PDF, or "Portable Data Format", I/O. This format is used for interchange of data among different processors. When a file is open for PDF I/O, the file positions are in terms of character units instead of storage units. To allow a program to handle data files opened for either normal I/O or PDF I/O, the procedure $ioSize should be used instead of size to position within a data file.

$ioSize's procedure header looks like:

```
    INTEGER PROCEDURE $ioSize (POINTER(file) f; INTEGER typ);
```

(Actually, $ioSize is a macro, not a procedure, but it acts as if it were a procedure declared with the above header). $ioSize returns the number of storage or character units, as appropriate, occupied by the data type with the type code typ in the file f, based on whether or not f is open for PDF I/O.

Since the program of Example 10.11-4 uses size instead of $ioSize, it will not work if the "PDF" device prefix is given in the database file name. To allow for this possibility, all occurrences of "size" would have to be replaced in the module with appropriate calls to $ioSize; for example, the body of the procedure bucketPos would be changed to:

```
        RETURN(firstBucketPos +
            cvli(hashCode * $ioSize(f,longIntegerCode)));
```

10.13. Restrictions on Files

Some characteristics of files vary from system to system. For example, on some operating systems it is not possible to determine the exact end position of a file, so that the system procedure $gotValue should not be counted on to become false at exactly the last position to which a program has written a file. You should read the system-specific MAINSAIL user's guide for your system and the chapter on files in the "MAINSAIL Language Manual" for further information.

10.14. Exercises

Debugging tip for the following exercises: the MAINSAIL utilities TVIEW and DVIEW can be used to examine the contents of text and data files, respectively. The MAINEDIT back end DATMGR is also useful for examining a data file. This may help you figure out what's in your symbol table files.

Exercise 10-1.

Modify the program of Example 10.11-4 to provide a command to delete a record. Try to reuse the space occupied by the deleted record. You may have to change the format of the information stored in the file in order to do this in an efficient manner.

Exercise 10-2.

Write a program that performs the same functions as that of Example 10.11-4, but uses a random-access text file instead of a data file. If you use the null character code in the file (0 on an ASCII or EBCDIC system), you must open the file with the "keepNul" control bit; see the "MAINSAIL Language Manual".

11. Records and Pointers

This chapter discusses the pointer data type, which is used to access records. Records are the most common means of maintaining miscellaneous information in memory while a program is running; such data structures as lists, trees, and arbitrary graphs are commonly built from records.

The pointer data type is also used to access arrays (Chapter 12) and data sections (Chapter 15). Records, arrays, and data sections are all subject to garbage collection and may be moved about in memory during execution by MAINSAIL's memory management routines (but this is usually invisible to a program).

11.1. Records, Classes, and Pointers

Until now, all information used by a program during its execution has had to be maintained either in a file (which may continue to exist after the program completes) or in the outer or local variables declared in the program. Data in a file are clumsier to access than data in named variables (see Example 10.11-4). However, if a program is to deal with information of unknown quantity and structure, it is not possible to create a named variable for every possible datum that may be manipulated during the course of the program's execution.

Records provide a repository for data that can be accessed more easily than data in files, although not so easily as data in named variables. A program may allocate as many records as it needs during execution (within the constraints of the computer's memory size). Since records are stored in the program's memory, they disappear at the end of a MAINSAIL execution unless explicitly stored in a file (the Structure Blaster provides one convenient technique for doing this; see the "MAINSAIL Structure Blaster User's Guide").

The pointer data type is used to access records. A pointer is so named because it "points" to a record; Example 11.1-1 shows how a pointer p pointing to a record r is depicted graphically.

Every record may be thought of as containing zero or more variables, possibly of different data types. Every record has a "class", or shape, telling which variables are present in the record and in which order they occur. The variables are called "fields" of the record or of the record's class. Each field is referred to by a name. The names of classes and of the fields they contain are declared in "class declarations". For example, the class declared in Example 11.1-2 is named "c" and has two integer fields named "int1" and "int2" and a string field named "stringField". The semicolon preceding the closing parenthesis of a class declaration is optional (such an optional semicolon is allowed in procedure parameter lists as well, although no examples appear in this tutorial).

```
p ----+
      |
      +----> +---------------+
             |               |
             |       r       |
             |               |
             +---------------+
```

Example 11.1-1. A Pointer p Pointing to a Record r

```
CLASS c (
    INTEGER int1,int2;
    STRING stringField; # This semicolon is optional
);
```

Example 11.1-2. A Sample Class Declaration

A record of the class c of Example 11.1-2 would be depicted diagrammatically as shown in Example 11.1-3. Sometimes the data types or values of the fields are shown in the boxes in addition to or instead of the field names.

```
+-------------+
| int1        |
+-------------+
| int2        |
+-------------+
| stringField |
+-------------+
```

Example 11.1-3. A Record of Class c

Most pointers are "classified"; i.e., they are allowed to point to records of only one class. That class is specified when the pointer is declared. Example 11.1-4 shows the declaration of a pointer p declared to point to records of the class c of Example 11.1-2.

```
POINTER(c) p;
```

Example 11.1-4. A Pointer Declared to Be of Class c

At any given moment a pointer variable points either to a record (or a data section, as described in Chapter 15) or to nothing at all; in the latter case the pointer is said to be (or to have the value) "nullPointer". The keyword "NULLPOINTER" represents the value nullPointer; performing the assignment "p := NULLPOINTER" causes a pointer variable p to have the value nullPointer, i.e., to point to nothing. The value nullPointer is the pointer Zero.

Records may themselves contain fields that are pointers to other records. Example 11.1-5 shows some class and pointer declarations and a data structure built up with pointers and records of the declared classes. "o" is used to depict a pointer that is nullPointer. The structure of Example 11.1-5 contains records with pointers to each other and to themselves.

Note that there is no path in Example 11.1-5 from p or q to REC 6, i.e., there is no expression of the form "p.f1.f2.f3..." or "q.f1.f2.f3..." ("f" stands for "field") that points to REC 6. If there is no path to REC 6 from some named variable, REC 6 is said to be "inaccessible". Such a record is useless, because it can never be referred to from a program. The MAINSAIL garbage collector eventually tracks down inaccessible records and reuses the space they occupy, so REC 6 is doomed.

Example 11.1-5 illustrates the exception to the rule that every identifier in MAINSAIL must be declared before it is used. A class need not be declared before its name is used in other declarations. The class declaration must appear, however, before any pointers of the class are used to reference fields of records of the class. Example 11.1-5 also shows that a named variable (e.g., the pointer p) may have the same name as the field of a class; furthermore, the same field name may be used in different classes. The uses of such names are always distinguishable by context.

```
CLASS c1 (
    STRING s;
    POINTER(c2) p; # c2 need not have been declared at
                   # this point
);
```

Example 11.1-5. Some Declarations and a Data Structure (continued)

```
CLASS c2 (
    POINTER(c1) p1;
    POINTER(c2) p2;
);

POINTER(c2) p,q;
                                      +--------------+
                                      |              |
                         REC 2        |      REC 3   |
        REC 1   +--> +-------------+  +--> +------+  |
p --> +------+  |    | s = "Hello" |  |    | p1 o |  |
      | p1 >-+--+-+  +-------------+  |    +------+  |
      +------+    |  | p >---------+--+    | p2 >-+--+
      | p2 >-+--+ |  +-------------+  |    +------+
      +------+  | |                   |
                | +--------------+    +--------------+
 +--------------+                |                   |
 |                       REC 5   |           REC 6   |
 |      REC 4     +--> +------+  |         +------+  |
 +--> +------+    |    | p1 >-+--+         | p1 o |  |
 |    | p1 o |    |    +------+            +------+  |
 +-+  +------+    |    | p2 o |            | p2 >-+--+
   |  | p2 >-+----+    +------+            +------+
q -+  +------+
```

A field of a record is referred to by means of the pointer to the record followed by a period (".") followed by the field name. Therefore, the following statements are true of the above diagram:

p.p1.s = "Hello", since p points to REC 1, and p.p1 points to REC 2, and the "s" field of REC 2 has the value "Hello".

p.p2 = q, since both p.p2 and q point to REC 4; two pointers are equal if and only if they point to the same record.

Example 11.1-5. Some Declarations and a Data Structure (continued)

```
q.p2.p1 = p.p1, since q.p2 points to REC 5, so q.p2.p1
points to REC 2, which is also pointed to by p.p1.

p.p1.p = p.p1.p.p2, since p.p1.p points to REC 3, and
the p2 field of REC 3 points to REC 3.
```

Example 11.1-5. Some Declarations and a Data Structure (end)

11.2. Allocation of Records

To create (or "allocate") a record, the system procedure "new" must be called. "new" is a special procedure because it takes a parameter that is the name of a class (the ordinary user is not able to declare a procedure with such a parameter). The structure of Example 11.1-5 could have been built up by the statements of Example 11.2-1.

```
p := new(c2); # Allocate REC 1 and make p point to it
p.p1 := new(c1); # Allocate REC 2; p.p1 points to it
p.p2 := new(c2); # Allocate REC 4; p.p2 points to it
p.p1.s := "Hello"; # Field s of REC 2 gets value "Hello"
p.p1.p := new(c2); # Allocate REC 3; p.p1.p points to it
p.p2.p1 := NULLPOINTER; # Zero p1 field of REC 4
p.p2.p2 := new(c2); # Allocate REC 5; p.p2.p2 points to it
p.p1.p.p1 := NULLPOINTER; # Zero p1 field of REC 3
p.p1.p.p2 := p.p1.p; # p2 field of REC 3 points to REC 3
p.p2.p2.p1 := p.p1; # p1 field of REC 5 points to REC 2
p.p2.p2.p2 := NULLPOINTER; # Zero p2 field of REC 5
q := new(c2); # Allocate REC 6, make q point to it
q.p1 := NULLPOINTER; # p1 field of REC 6 points nowhere
q.p2 := p.p1.p; # p2 field of REC 6 points to REC 3
q := p.p2; # Change q to point to REC 4; REC 6 now becomes
           # inaccessible, since no pointer points to it
```

Example 11.2-1. Building the Structure of Example 11.1-5

When "new" allocates a record, all of the record's fields are originally Zero. Thus, the assignments with "NULLPOINTER" on the right-hand side in Example 11.2-1 are not really necessary, although they make clearer what is going on.

Attempting to reference a field using a pointer that is Zero results in an error message from the MAINSAIL runtime system.

11.3. Using Pointers to Maintain a List

The program of Example 11.3-1 maintains a list of records something like that of Example 10.11-4, although in Example 11.3-1 only one list is used; i.e., there are no hash buckets. The user is allowed to enter a series of records, each with a name and a value, and to look up records by name. Unlike the structure created in Example 10.11-4, the list of records disappears as soon as program execution completes, since the records are not preserved in a file.

The program of Example 11.3-1 provides one command not provided by Example 10.11-4: a command to delete a record. The space occupied by the record is reclaimed automatically by the MAINSAIL runtime system, since the deleted records become inaccessible.

Class declarations may appear only among the outer declarations of a module; classes local to a procedure are not allowed.

```
BEGIN "list"

CLASS rec (
    STRING name,value;
    POINTER(rec) next;
);

POINTER(rec) recList;

POINTER(rec) PROCEDURE lookup
    (STRING name;
     PRODUCES OPTIONAL POINTER(rec) previousRecord);
# If no record found, return NULLPOINTER; if the record
# found is not the first one on the list, set
# previousRecord to point to the previous one, else
# NULLPOINTER
BEGIN
POINTER(rec) p;

previousRecord := NULLPOINTER; p := recList;
# Note use of short-circuit evaluation: it would be an
# error to refer to "p.name" if p were NULLPOINTER
WHILE p AND (p.name NEQ name) DOB
    previousRecord := p; p := p.next END;
RETURN(p);
END;
```

Example 11.3-1. Use of Pointers to Maintain a List of Records (continued)

```
PROCEDURE showRecords;
BEGIN
POINTER(rec) p;

p := recList;
WHILE p DOB write(logFile,p.name,eol); p := p.next END;
END;



PROCEDURE createRecord (STRING s);
# Append the new record to the beginning of recList
BEGIN
STRING t;
POINTER(rec) p;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF lookup(s) THENB
    errMsg("Record already exists:",s); RETURN END;
# Allocate the record:
p := new(rec); p.name := s;
# Now read the record value from cmdFile:
write(logFile,
    "Enter record value; end with blank line" & eol);
DOB read(cmdFile,t); IF NOT t THEN DONE;
    write(p.value,t,eol) END;
# This is the standard way to add a new record to the
# head of a list:
p.next := recList; recList := p;
END;



PROCEDURE lookupRecord (STRING s);
BEGIN
POINTER(rec) p;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF p := lookup(s) THEN write(logFile,p.value)
EL errMsg("No such record:",s);
END;
```

Example 11.3-1. Use of Pointers to Maintain a List of Records (continued)

```
PROCEDURE deleteRecord (STRING s);
BEGIN
POINTER(rec) p,priorToP;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF p := lookup(s,priorToP) THENB # Unlink p from the list
    # If priorToP is NULLPOINTER, it means p is the first
    # record on recList
    IF priorToP THEN priorToP.next := p.next
    EL recList := recList.next END
EL errMsg("No such record:",s);
END;
```

Example 11.3-1. Use of Pointers to Maintain a List of Records (continued)

```
BOOLEAN PROCEDURE processCommand (STRING s);
# Return false if s is the quit command, true otherwise.
# The commands are "Q" (quit), "C" (create a new record),
# "S" (show names of all existing records), "L" (look
# up an existing record), and "D" (delete a record).
# "C", "D", and "L" commands are followed by the record
# name.
BEGIN
s := cvu(s); # So we don't have to worry about case
CASE cRead(s) OFB
    [-1]  ; # Do nothing if blank line
    ['Q'] RETURN(FALSE);
    ['S'] showRecords;
    ['C'] createRecord(s);
    ['L'] lookupRecord(s);
    ['D'] deleteRecord(s);
    ['?'] ['H'] write(logFile,
        "Q              to quit" & eol &
        "S              show names of all records" & eol &
        "C recName      create record recName" & eol &
        "D recName      delete record recName" & eol &
        "L recName      look up record recName" & eol &
        "? or H         get this message" & eol);
    [ ] write(logFile,
            "Invalid command (? for help)" & eol);
    END;
RETURN(TRUE);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

DOB write(logFile,"Command: "); read(cmdFile,s) END
    UNTIL NOT processCommand(s);
END;

END "list"
```

Example 11.3-1. Use of Pointers to Maintain a List of Records (end)

11.4. Assignment Compatibility, Prefix Classes, and Unclassified Pointers

The MAINSAIL compiler usually issues an error message if it finds a pointer of one class assigned to a pointer of another class or passed as a parameter of another class. However, two classes may share some common initial fields by means of a mechanism called a "prefix class". If one class is a prefix class of another, then the two classes are considered "assignment compatible"; i.e., pointers of one class may be assigned to or passed as pointers of the other. Examples of prefix class declarations are deferred until Section 15.3; however, it is worth noting here that the system procedure "close" (of which the parameter is declared as a "POINTER(file)") accepts parameters declared as either "POINTER(textFile)" or "POINTER(dataFile)" because the class "file" is a prefix class of both textFile and dataFile. More information on prefix classes may be found in the "MAINSAIL Language Manual" and in Section 15.3 of this tutorial.

An "unclassified" pointer is one declared without a parenthesized class name following the keyword "POINTER". An unclassified pointer is assignment compatible with any other pointer. An example appears in Section 15.7.

11.5. Using Pointers to Construct a Binary Tree

The program of Example 11.5-2 uses a data structure known as a binary tree to alphabetize a series of strings. The strings are read in, one per line, from a file; as each is read, it is added to the binary tree. The tree is maintained in such a way that if it is printed out in "infix order", the result is an alphabetized list.

Each node in the tree consists of three fields: two pointers and a string. The "left" pointer points to a subtree of strings alphabetically preceding the string field; the "right" field, to a subtree of strings following the string field. See Example 11.5-1. Note that all the words on the branch descending from the left of a given word alphabetically precede it; all those descending from the right of a word follow it.

```
       owl
      /   \
    cat   zebra
   /   \
 ape   fish
          \
          lemming
```

Example 11.5-1. A Binary Tree of Strings

To maintain the tree in this alphabetical order, each new word must be added according to the following algorithm:

1. Set the "current node" to be the root of the tree (e.g., the "owl" node in Example 11.5-1).

2. If the current node is nullPointer, add the new word at the current node and stop.

3. If the new word alphabetically precedes the word at the current node, set the current node to be the node down to the left; if it follows the current word, set the current node to be the node down to the right. If the new word is the same as the word at the current node, stop (i.e., don't put it in the tree twice); otherwise, go back to step 2.

Printing the tree in infix order means printing the strings of the left subtree of each node (in infix order), then the string at this node, then the strings of the right subtree (in infix order). This is a recursive algorithm, and is implemented in Example 11.5-2 as a recursive procedure. If you are unfamiliar with the notion of "infix order", you may be excused if you have to think about this for a while to verify that it works.

Note that the tree constructed by this algorithm does not necessarily have the same shape if the order of the input strings is altered; however, it always comes out in proper alphabetical order if printed in infix order.

```
BEGIN "binTre"

# Use a binary tree to alphabetize the lines in a text
# file.

CLASS bin (
    POINTER(bin) left,right;
    STRING here;
);

POINTER(bin) root;
```

Example 11.5-2. Use of Pointers to Construct a Binary Tree (continued)

```
PROCEDURE alphabetize (STRING s; MODIFIES POINTER(bin) p);
# If p is nullPointer, create a node for it containing s.
# If p is not nullPointer, add a node on the left subtree
# if s precedes p.here, on the right subtree if it
# follows.  If it's the same as a node already there,
# don't add it.
BEGIN
IF NOT p THENB # create the node
    p := new(bin); p.here := s; RETURN END;
CASE compare(s,p.here,upperCase) OFB
    [-1] alphabetize(s,p.left);
    [0] RETURN;
    [1] alphabetize(s,p.right);
    END;
END;



PROCEDURE infixPrint (POINTER(bin) p);
BEGIN
IF NOT p THEN RETURN;
infixPrint(p.left);
write(logFile,p.here,eol);
infixPrint(p.right);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;
POINTER(textFile) f;

open(f,"Input file: ",input!prompt);
DOB read(f,s); IF NOT $gotValue(f) THEN DONE;
    IF s THEN alphabetize(s,root) END;
close(f);
infixPrint(root);
END;

END "binTre"
```

Example 11.5-2. Use of Pointers to Construct a Binary Tree (end)

11.6. Exercises

Exercise 11-1.

Which records in Example 11.1-5 are accessible from p? Which from q? For each record accessible from p, construct an expression beginning with p that points to the record (e.g., "p.p1.p" points to REC 3); do the same for q.

Exercise 11-2.

Write a program that sorts strings using a linear list rather than a binary tree. Which algorithm would you expect to run faster?

12. Arrays; the System Procedure cmdMatch

This chapter describes the array, a random-access data structure. MAINSAIL arrays may have one, two, or three dimensions. Like records, arrays are subject to garbage collection.

12.1. Lists and Arrays

The linked list constructed of records used in Chapter 11 is used to store a series of pieces of information in memory. It is a satisfactory data structure when you want to find an item based on some characteristic of the item, as in the program of Example 11.3-1, where a search is based on the string "name" field of a record.

Items in a series of things are often accessed by number. To find the tenth item in a linked list, you must start a search at the first item in the list and progress through the items one by one until you reach the tenth. The MAINSAIL array data structure provides a more efficient (and syntactically more convenient) means of accessing an item by number.

An array consists of a number of components (called "elements" or "array elements"), all of the same data type. The elements may be of any MAINSAIL data type, but they may not themselves be arrays (the array is not considered to be a data type). Some sample array declarations appear in Example 12.1-1. The numbers in parentheses following the keyword "ARRAY" are called "bounds"; the values preceding the keyword "TO" are lower bounds, and those following are upper bounds.

```
INTEGER ARRAY(0 TO 9) digitChars;
    # One integer for every digit

STRING ARRAY(1 TO 100) strings; # 100 strings

CLASS xxx (
    POINTER(xxx) next;
    INTEGER value;
);

POINTER(xxx) ARRAY(1 TO 20,-10 TO 10) yyy;
    # A two-dimensional array of 20 x 21 = 420 pointers
```

Example 12.1-1. Sample Array Declarations

Unlike most programming languages, MAINSAIL does not allocate arrays when they are declared. Arrays are allocated by the system procedure "new" (a different form of "new" from that described in Chapter 11). To allocate an array by means of "new" is called "to new" the array (to allocate a record is also sometimes referred to as "to new" the record or the pointer variable that points to it). The arrays of Example 12.1-1 are newed in Example 12.1-2. The elements of an array are not accessible until after it has been newed; after the call to "new", all the elements of the allocated array have the Zero value for their data type.

```
new(digitChars);
new(strings);
new(yyy);
```

Example 12.1-2. Newing Some Arrays

An array element is accessed by placing an integer (or two or three integers if the array is two- or three-dimensional) in square brackets following the name of the array. The integer or integers in brackets are called "subscripts". The array name and its bracketed subscript or subscripts together constitute a "subscripted variable". Some sample uses of subscripted variables using the arrays of Example 12.1-1 appear in Example 12.1-3.

```
digitChars[6] := '6'; # Element 6 of the array acquires
                      # the value that is the character
                      # code for the digit "6"

INTEGER i; STRING s;
# Look for the first null string in the array (note the
# use of an Empty Statement as the iterated statement):
FOR i := 1 UPTO 100 WHILE strings[i] DO;
# i has the value 101 if no null string was found:
IF i > 100 THEN errMsg("No null string found")
EL strings[i] := s; # Replace the null string with s

INTEGER i,j;
# A subscripted variable that is a pointer may be used
# to access fields:
yyy[i,j] := yyy[j,yyy[i,j].value].next;
```

Example 12.1-3. Subscripted Variables

An array variable is actually a special sort of pointer variable. When the array is allocated, it points at the data structure allocated by "new". When one array variable is assigned to another, both array variables are then made to point to the same array; no copying of elements occurs. If a change is made to an element of the array using one array variable, the changed value is accessed when the element is examined using the other array variable; see Example 12.1-4. Similarly, when an array is passed as an array parameter, no copying of elements occurs; only the array pointer is actually passed.

An array variable with the array Zero value, nullArray (designated by the keyword "NULLARRAY"), points to no array.

As with records, the memory occupied by an array that has become inaccessible is recycled by the MAINSAIL runtime system.

```
INTEGER ARRAY(1 TO 3) numbers,counters;

        .
        .
        .

new(numbers);
# numbers now points to an allocated array.  "new" sets
# all the elements of the allocated array to have the
# Zero value for their data type.  counters points to
# no array; it is an outer variable and so initially
# has the value nullArray:
# numbers ----> +----------+      counters o
#               |        0 |
#               +----------+
#               |        0 |
#               +----------+
#               |        0 |
#               +----------+
```

Example 12.1-4. Array Assignment (continued)

```
numbers[2] := -17;
# numbers ----> +----------+       counters o
#               |        0 |
#               +----------+
#               |      -17 |
#               +----------+
#               |        0 |
#               +----------+

counters := numbers;
# counters -+
#           |
# numbers --+-> +----------+
#               |        0 |
#               +----------+
#               |      -17 |
#               +----------+
#               |        0 |
#               +----------+
#
# counters and numbers now point to the same array.

counters[3] := 6;
# counters -+
#           |
# numbers --+-> +----------+
#               |        0 |
#               +----------+
#               |      -17 |
#               +----------+
#               |        6 |
#               +----------+
#
# Note that "numbers[3]" now has the value 6 as well.
```

Example 12.1-4. Array Assignment (continued)

```
new(numbers); # Make numbers point at a new, separate
              # array; counters continues to point at
              # the original array:
# counters -> +----------+  numbers --> +----------+
#             |        0 |              |        0 |
#             +----------+              +----------+
#             |      -17 |              |        0 |
#             +----------+              +----------+
#             |        6 |              |        0 |
#             +----------+              +----------+
#
# "counters[3]" still has the value 6, but "numbers[3]"
# now has the value 0.
        .
        .
        .
```

Example 12.1-4. Array Assignment (end)

12.2. Sample Program with Arrays and Pointers

In Example 12.2-1, the program of Example 11.3-1 is adapted to use a hash table (see Section 10.11 for a discussion of hash tables). The hash table is maintained as an array of pointers. The items are records of the class "rec", and their keys are their "name" fields; the hash function is implemented by the procedure "hash".

```
BEGIN "pList"

DEFINE numBuckets = 131; # Number of hash buckets

CLASS rec (
    STRING name,value;
    POINTER(rec) next;
);

POINTER(rec) ARRAY(0 TO numBuckets - 1) hashTable;

INTEGER PROCEDURE hash (STRING s);
# Returns a value in the range 0 to numBuckets - 1
BEGIN
INTEGER h,i,j;
i := (h := length(s)) MIN 4; j := 1;
WHILE (i .- 1) GEQ 0 DO h .+ cRead(s) * (j .+ 2);
RETURN((h MAX 0) MOD numBuckets) END;



POINTER(rec) PROCEDURE lookup
    (STRING name;
     PRODUCES OPTIONAL POINTER(rec) previousRecord);
# If no record found, return NULLPOINTER; if the record
# found is not the first one on the list, set
# previousRecord to point to the previous one, else
# NULLPOINTER
BEGIN
POINTER(rec) p;
```

Example 12.2-1. Use of an Array as a Hash Table (continued)

```
previousRecord := NULLPOINTER; p := hashTable[hash(name)];
# Note use of short-circuit evaluation: it would be an
# error to refer to "p.name" if p were NULLPOINTER
WHILE p AND (p.name NEQ name) DOB
    previousRecord := p; p := p.next END;
RETURN(p);
END;



PROCEDURE showRecords;
BEGIN
INTEGER i;
POINTER(rec) p;

FOR i := 0 UPTO numBuckets - 1 DOB
    p := hashTable[i];
    WHILE p DOB
        write(logFile,p.name,eol); p := p.next END END;
END;



PROCEDURE createRecord (STRING s);
# Append the new record to the beginning of its hash list
BEGIN
INTEGER i;
STRING t;
POINTER(rec) p;
```

Example 12.2-1. Use of an Array as a Hash Table (continued)

```
scan(s," " & tab,proceed!omit); # Remove leading blanks
IF lookup(s) THENB
    errMsg("Record already exists:",s); RETURN END;
# Allocate the record:
p := new(rec); p.name := s;
# Now read the record value from cmdFile:
write(logFile,
    "Enter record value; end with blank line" & eol);
DOB read(cmdFile,t); IF NOT t THEN DONE;
    write(p.value,t,eol) END;
# This is the standard way to add a new record to the
# head of a list:
p.next := hashTable[i := hash(s)]; hashTable[i] := p;
END;



PROCEDURE lookupRecord (STRING s);
BEGIN
POINTER(rec) p;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF p := lookup(s) THEN write(logFile,p.value)
EL errMsg("No such record:",s);
END;



PROCEDURE deleteRecord (STRING s);
BEGIN
INTEGER i;
POINTER(rec) p,priorToP;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF p := lookup(s,priorToP) THENB # Unlink p from the list
    # If priorToP is NULLPOINTER, it means p is the first
    # record on its list
    IF priorToP THEN priorToP.next := p.next
    EL hashTable[i := hash(s)] := hashTable[i].next END
EL errMsg("No such record:",s);
END;
```

Example 12.2-1. Use of an Array as a Hash Table (continued)

```
BOOLEAN PROCEDURE processCommand (STRING s);
# Return false if s is the quit command, true otherwise.
# The commands are "Q" (quit), "C" (create a new record),
# "S" (show names of all existing records), "L" (look
# up an existing record), and "D" (delete a record).
# "C", "D", and "L" commands are followed by the record
# name.
BEGIN
s := cvu(s); # So we don't have to worry about case
CASE cRead(s) OFB
    [-1]  ; # Do nothing if blank line
    ['Q'] RETURN(FALSE);
    ['S'] showRecords;
    ['C'] createRecord(s);
    ['L'] lookupRecord(s);
    ['D'] deleteRecord(s);
    ['?'] ['H'] write(logFile,
        "Q               to quit" & eol &
        "S               show names of all records" & eol &
        "C recName       create record recName" & eol &
        "D recName       delete record recName" & eol &
        "L recName       look up record recName" & eol &
        "? or H          get this message" & eol);
    [ ] write(logFile,
            "Invalid command (? for help)" & eol);
    END;
RETURN(TRUE);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

new(hashTable); # Allocate the array

DOB write(logFile,"Command: "); read(cmdFile,s) END
    UNTIL NOT processCommand(s);
END;

END "pList"
```

Example 12.2-1. Use of an Array as a Hash Table (end)

12.3. The Init Statement

The Init Statement may be used to initialize an array that has already been allocated. It is often syntactically more convenient to use an Init Statement than to assign a value to each array element with an Assignment Statement. There is usually no great difference in execution speed between an Init Statement and multiple Assignment Statements.

The Init Statement consists of the keyword "INIT", the name of the array being initialized, and a series of constant values in parentheses. The values are assigned to the array elements, one value per element, starting with the lowest subscript, until either the Init Statement list runs out (in which case the remaining array elements are unaltered) or the highest array subscript is reached (in which case an error message is issued). In the most common case, there are exactly as many Init Statement constants as elements in the array.

Init Statement constants may be accompanied by a "repetition", in which case more than one array element receives the same value from the list. Consult the "MAINSAIL Language Manual" for the rules governing repetitions and the use of Init Statements for two- and three-dimensional arrays.

Example 12.3-1 shows an Init Statement and its equivalent Assignment Statements.

```
If an array a is declared as:

    REAL ARRAY(-3 TO 3) a;

then the Init Statement:

    INIT a (6.2,2.8E-10,-16.4,0.0,3.9,98.6,-11.0)

is equivalent to:

    a[-3] := 6.2;
    a[-2] := 2.8E-10;
    a[-1] := -16.4;
    a[0] := 0.0;
    a[1] := 3.9;
    a[2] := 98.6;
    a[3] := -11.0
```

Example 12.3-1. An Init Statement and Equivalent Assignment Statements

12.4. The System Procedure cmdMatch

The system procedure cmdMatch provides a facility that assists in the interpretation of commands. It attempts to match the first part of a given string or a string read from logFile with a string from an array of strings; if it finds a match in the array, it returns the index of the matching array element. The declaration of cmdMatch is shown in Figure 12.4-1. The use of "*" as an array bound indicates that any one-dimensional string array may be passed for the "commands" parameter; variable-bounded arrays are discussed in Section 12.5.

```
INTEGER PROCEDURE cmdMatch (STRING ARRAY(*) commands;
                            OPTIONAL STRING promptString;
                            OPTIONAL BITS ctrlBits;
                            PRODUCES OPTIONAL STRING s)
```

Figure 12.4-1. Declaration of cmdMatch

By default (i.e., if ctrlBits is Zero), cmdMatch writes promptString to logFile, and reads the string s from cmdFile. If the string s is an unambiguous abbreviation for one of the elements of the commands array, then the index of that element is returned. Otherwise, cmdMatch writes an error message and reprompts. If the string read from cmdFile begins with "?", the list of commands in the commands array is written to logFile, and cmdMatch reprompts.

Predefined bits that may be set in ctrlBits include errorOK (don't reprompt if no match; instead, return an invalid index); noResponse (don't write to logFile or read from cmdFile; instead, use promptString as the match string); and useKeyWord (useful if a command line may consist of several elements separated by blanks or tabs; it leaves the part after the first word parsed in the string s). See the "MAINSAIL Language Manual" for details.

Example 12.4-2 shows how the procedure processCommands and the initial procedure of Example 12.2-1 might be rewritten using cmdMatch.

```
BOOLEAN PROCEDURE processCommand;
# Return false if the "QUIT" command is given.
BEGIN
STRING s;
OWN STRING ARRAY(1 TO 5) cmds; # Own array sticks around
                               # from procedure invocation
                               # to procedure invocation

IF NOT cmds THENB # Allocate only if not yet allocated
    new(cmds); INIT cmds (
"QUIT                     quit",
"SHOW                     show the names of all records",
"CREATE recName           create record named recName",
"DELETE recName           delete record named recName",
"LOOKUP recName           show the value of record recName");
    END;

CASE cmdMatch(cmds,"Command: ",useKeyWord,s) OFB
    [1] RETURN(FALSE);
    [2] showRecords;
    [3] createRecord(cvu(s));
    [4] lookupRecord(cvu(s));
    [5] deleteRecord(cvu(s));
    END;
RETURN(TRUE);
END;



INITIAL PROCEDURE;
BEGIN

new(hashTable); # Allocate the array

DO UNTIL NOT processCommand; # Note the use of an Empty
                             # Statement as the iterated
                             # statement
END;
```

Example 12.4-2. The Use of cmdMatch

12.5. Variable-Bounded Arrays

An asterisk ("*") may replace the lower bound or upper bound in an array declaration. It signifies that the bound is not known at compiletime, and may be set or changed at runtime. If both the lower and upper bound of a bound pair are unknown at compiletime, the entire bound pair may be replaced by a single asterisk; i.e., "*" may replace "* TO *" ("*" is used this way in Figure 12.4-1).

When a variable-bounded array is allocated with "new", the bounds to be used must be specified. The bounds specified to "new" appear in the order: lower bound of first dimension, upper bound of first dimension, lower bound of second dimension (specified only if the array has at least two dimensions), upper bound of second dimension (only if two- or three-dimensional), lower bound of third dimension (if three-dimensional), upper bound of third dimension (if three-dimensional). If the array declaration contains a mixture of constant and "*" bounds, the constant bounds must be specified to "new" as they appear in the array declaration. See Example 12.5-1.

The upper bounds of the three dimensions of an array may be referred to with the array name followed by ".ub1", ".ub2", and ".ub3"; the lower bounds with the array name followed by ".lb1", ".lb2", and ".lb3". This is useful when the array bounds are not otherwise available. Examples appear in Example 12.6-1.

An array declared as:

```
    INTEGER ARRAY(1 TO *) ary
```

may be newed with:

```
    new(ary,1,10)
```

to give it ten elements. The bound parameter to "new" need not be a constant if it corresponds to a "*" in the declaration; if i were an integer variable:

```
    new(ary,1,i)
```

would also be allowed. However, upper bounds must be greater than lower bounds; otherwise, a runtime error message is issued.

An array declared as:

```
    REAL ARRAY(*,* TO 20) ary2
```

may be allocated with:

```
    new(ary2,-4,4,0,20)
```

which would allocate an array of the same size and shape as if ary2 had been declared as:

```
    REAL ARRAY(-4 TO 4,0 TO 20) ary2
```

and:

```
    new(ary2)
```

had been performed.

Example 12.5-1. Allocation of a Variable-Bounded Array

12.6. Multidimensional Arrays

Example 12.6-1 uses variable-bounded two-dimensional arrays to perform matrix multiplication. The sizes of the matrices are read from an input file, so the array bounds cannot be declared as constants within the program.

The rule for matrix multiplication is that two matrices, the first one m rows by n columns, the second n rows by p columns, have as a product an m-row by p-column matrix. The product element in row i at column j is the sum of the products of the elements of the ith row of the first matrix and the corresponding elements of the jth column of the second matrix.

12.7. newUpperBound

The system procedure newUpperBound may be used to change the upper bound (increasing it or decreasing it) of a one-dimensional array. Its declaration is shown in Figure 12.7-1. The upper bound of the array a is changed to n; if the upper bound is increased, the new elements at the end are given an initial value of Zero. A use of newUpperBound is shown in Example 16.4-1.

```
PROCEDURE newUpperBound (MODIFIES ARRAY(*) a; INTEGER n);
```

Figure 12.7-1. Declaration of newUpperBound

12.8. Long Arrays

Arrays may have long integer as well as integer subscripts; such arrays are called "long arrays". When array indices are large (i.e., when an array subscript calculation might produce an intermediate result with a magnitude greater than 32767), a long array may be required. The "MAINSAIL Language Manual" describes long array syntax and the circumstances under which long arrays must be used instead of short arrays.

```
BEGIN "matMul"

# Multiply two matrices with long real elements.

LONG REAL ARRAY(1 TO *,1 TO *) a,b,prod;

PROCEDURE readMatrix
    (PRODUCES LONG REAL ARRAY(1 TO *,1 TO *) ary);
BEGIN
INTEGER rows,cols,i,j;
STRING s;

write(logFile,"Number of rows: "); read(cmdFile,s);
rows := cvi(s);
write(logFile,"Number of colums: "); read(cmdFile,s);
cols := cvi(s);
new(ary,1,rows,1,cols);
# The elements are to be entered one row per line,
# separated from each other on the line by spaces
FOR i := 1 UPTO rows DOB
    write(logFile,"Row #",i,": "); read(cmdFile,s);
    FOR j := 1 UPTO cols DO read(s,ary[i,j]) END;
END;



PROCEDURE printMatrix
    (LONG REAL ARRAY(1 TO *,1 TO *) ary);
BEGIN
INTEGER i,j;

FOR i := 1 UPTO ary.ub1 DOB
    FOR j := 1 UPTO ary.ub2 DO
        # Make each element occupy seven characters:
        write(logFile,cvs(ary[i,j],fixed!'7)," ");
    write(logFile,eol) END;
write(logFile,eol);
END;
```

Example 12.6-1. Matrix Multiplication Using Variable-Bounded Two-Dimensional Arrays (continued)

```
PROCEDURE multiply
    (LONG REAL ARRAY(1 TO *,1 TO *) a1,a2;
     PRODUCES LONG REAL ARRAY(1 TO *,1 TO *) result);
BEGIN
INTEGER m,n,p,i,j,k;
LONG REAL sum;

m := a1.ub1; n := a1.ub2; p := a2.ub2;
new(result,1,m,1,p); # m x p result
FOR i := 1 UPTO m DO FOR j := 1 UPTO p DOB
        sum := 0.0L;
        FOR k := 1 UPTO n DO sum .+ (a1[i,k] * a2[k,j]);
        result[i,j] := sum END;
END;



INITIAL PROCEDURE;
BEGIN
write(logFile,"First matrix:" & eol);
readMatrix(a);
write(logFile,"Second matrix:" & eol);
readMatrix(b);
IF a.ub2 NEQ b.ub1 THENB
    # Arrays cannot be multiplied unless they are of the
    # proper shape
    errMsg("a.ub2 NEQ b.ub1"); RETURN END;
write(logFile,"First multiplicand matrix:" & eol);
printMatrix(a);
write(logFile,"Second multiplicand matrix:" & eol);
printMatrix(b);
multiply(a,b,prod);
write(logFile,"Product matrix:" & eol);
printMatrix(prod);
END;

END "matMul"
```

Example 12.6-1. Matrix Multiplication Using Variable-Bounded Two-Dimensional Arrays (end)

12.9. Exercises

Exercise 12-1.

Write a program that counts the number of occurrences of each character in an input file. Example 10.9-2 groups characters into broad categories; in your program, give a count for each character in the character set (unless no instances of the character were found in the file, in which case don't mention the character in your output). Character codes range from 0 through the predefined value $maxChar.

Exercise 12-2.

Write a program that solves mazes. The input file consists of lines of blanks and X's, as shown in Example 12.9-1, with one "S" (start position) and one "E" (ending position). A blank line terminates the maze input file. You may assume all lines are the same length, but the number of lines and the length of the lines must be determined by actually reading the file. You may also assume the input file contains no tab characters. You should try to find a path from "S" to "E" that passes only through blank spaces (the X's represent walls). If no such path exists, you should report so; otherwise, you should print the maze with the path marked by "." characters (if there is more than one path, you need not find all of them). The output should look something like Example 12.9-2 (of course, other paths are possible for the input given). The path must go horizontally or vertically only; i.e., you may not squeeze diagonally between two X's.

```
XXXXXXXXXXXXXXXXXXXXXXXXXXX
S          X      X     X X
XXXXXX XXXXX X XXXX XXXXX X
X      X X   E  X         X
X XXXX   X XXX  X X XXXXX X
X      X X X X    X X   X X
X XXXXXX X   XX XXXXX XXX X
X      X   X          X   X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Example 12.9-1. A Sample Maze Input File

```
XXXXXXXXXXXXXXXXXXXXXXXXXXX
S......    X      X     X X
XXXXXX.XXXXX X XXXX XXXXX X
X     .X X   E..X         X
X XXXX...X XXX .X X XXXXX X
X      X.X X X .  X X   X X
X XXXXXX.X...XX.XXXXX XXX X
X      X...X....      X   X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Example 12.9-2. Sample Output from the Maze Solver Program

13. Macros; Conditional Compilation; Comparison Chains

This chapter covers several important operations that may be performed by the MAINSAIL compiler on its input source text at compiletime. Macros provide a very general mechanism for textual substitution within programs. Conditional compilation allows selected pieces of program text to be ignored by the compiler. Comparison chains are a syntactic convenience for simplifying some expressions that contain comparison operators.

13.1. Macro Constants

A form of macro called a "macro constant" was introduced in Section 3.2. The form used there was:

```
        DEFINE <identifier> = <constant value>;
```

This definition contains a single "macro equate", which associates the value of the macro constant with the identifier. The constant value can be a single constant of the appropriate data type, or it may be an expression that can be evaluated at compiletime. Many operators and "simple" procedures may be evaluated at compiletime, provided that their operands or parameters are also evaluated at compiletime; such operators and procedures include "+", "-", "*", and "DIV" (for integers and long integers), many conversion procedures, "cvl" and "cvu" for strings, "first", "last", and a number of others. No operator or procedure with real or long real operands, parameters, or result value is evaluated at compiletime. The rules determining exactly which operations may be evaluated at compiletime are described in the "MAINSAIL Language Manual".

A macro definition may contain a series of macro equates separated by commas. The whole series, from the initial keyword "DEFINE" to the terminating semicolon, is referred to as a single "macro definition", even though more than one macro may be defined; the part that defines a single macro identifier is called a "macro equate". The macro equates in a single macro definition may be of different data types, as shown in Example 13.1-1. The part of a macro equate to the right of the equals sign is called the "macro body"; it is what the macro identifier is subsequently used to represent.

Using macro constants in a program instead of hardwiring the values they represent is good programming practice. An identifier like "numberOfLetters" makes clear to a human reader what "26" might fail to communicate. Also, if the value of a constant in a program needs to be changed, only the definition need be changed if the constant is represented by a macro identifier. Otherwise, for example, if an integer constant appears throughout a program in

```
# This macro definition includes four equates, and defines
# the constants a, b, c, s, and ss.
DEFINE
    a = 1, # An integer macro constant
    b = a, # Another
    c = a + b, # An expression (integer addition)
               # evaluated at compiletime
    s = "Hi there!" & eol, # & is also evaluated at
                           # compiletime
    ss = cvu(s); # So is cvu
```

Example 13.1-1. A Macro Definition Containing Several Macro Equates

numeric form, then every instance of that number must be changed whenever the constant is to be changed.

13.2. Bracketed Text (Textual Substitution)

A macro constant used in an expression may be thought of as being "replaced" with a constant representing its value. For example, if a macro x is defined as:

```
DEFINE x = 2 + 2;
```

then the expression "5 * x" has the same value as "5 * 4"; i.e., the expression is evaluated as if the string "4" replaced the string "x" in the program text.

Macros may be defined to contain almost any text by means of the use of "bracketed text". Although macro constants are constrained to represent a valid constant value of a MAINSAIL data type, bracketed text macro bodies may represent almost any string of characters. The macro body of a bracketed text macro consists of the text to be represented by the macro identifier enclosed in square brackets ("[" and "]"). See Example 13.2-1. Note that bracketed text macro equates may be intermixed with macro constant equates in the same macro definition. Section 5.18 of part II of the "MAINSAIL Tutorial" describes how to include brackets in bracketed text.

Macros may contain occurrences of other macro identifiers (the occurrence of a macro identifier that has already been defined is called a "macro call"). See Example 13.2-2. Note that the order of definition of the macros in Example 13.2-2 does not matter; i.e., no error occurs when the body of macro1 is seen in the macro definition, even though it contains the identifiers "macro2" and "macro3", which have not yet been defined. No processing is done on

```
If abc, def, ghi, and jkl are defined as shown:

    DEFINE
        abc = [i = 4],
        def = [IF],
        ghi = [THEN write(logFile,jkl)],
        jkl = "i is 4." & eol;

then the text:

    def abc ghi

is "replaced with" (or "expands to"):

    IF i = 4 THEN write(logFile,jkl)

which, since jkl has the value "i is 4." & eol, is
equivalent to:

    IF i = 4 THEN write(logFile,"i is 4." & eol)
```

Example 13.2-1. Bracketed Text Macros

the text of a bracketed text macro at the point of definition; all processing is performed at the point of the macro call.

```
If macro1, macro2, macro3, and macro4 are defined as:

    DEFINE
        macro1 = [macro2 macro3],
        macro2 = [IF macro4 THEN],
        macro3 = [write(logFile,"Oops")],
        macro4 = [errorsSeen];

then the text:

    macro1;

expands to:

    macro2 macro3;

which in turn expands to:

    IF macro4 THEN write(logFile,"Oops");

which in its turn expands to:

    IF errorsSeen THEN write(logFile,"Oops");

which cannot be further expanded, since it contains no
macro calls.
```

Example 13.2-2. Macro Bodies Containing Macro Calls

13.3. Macros with Parameters

A bracketed text macro may have parameters, something like a procedure. The parameters to a macro, however, do not need to have data types associated with them, since the substitution of a macro argument for its corresponding parameter is a textual substitution performed at compiletime, rather than the assignment of a value performed at runtime. Example 13.3-1 shows the definition of a macro with parameters along with the way the macro expands when it is called.

```
If maxCount and isTooBig are defined as:

    DEFINE
        maxCount     = 100,
        isTooBig(x)  = [(x > maxCount)];

then:

    IF isTooBig(i1) THEN errMsg("i1 too big")

expands to:

    IF (i1 > maxCount) THEN errMsg("i1 too big")

which, since maxCount has the value 100, is
equivalent to:

    IF (i1 > 100) THEN errMsg("i1 too big")

"i1" is substituted in the expansion of the macro
isTooBig wherever "x" appears in the macro body
definition.
```

Example 13.3-1. The Use of Macro Parameters

The keyword "REDEFINE" may be used to introduce a macro definition in which the identifiers defined may or may not already have been defined as macros ("DEFINE" gives an error if you try to define a macro that has already been defined). If the macro identifiers have already been defined, the new value replaces the old. If the macros have not already been defined, "REDEFINE" acts just like "DEFINE".

Macro bodies may contain macro definitions. This fact may be used in conjunction with "REDEFINE" to write a macro that defines a series of values, as shown in Example 13.3-2. The technique of Example 13.3-2 may be used to get something like the effect of enumerated types in Pascal and related languages, since MAINSAIL does not provide enumerated types directly.

```
The following text:

    DEFINE
        startVal = 1,
        color(xxx) =
            [DEFINE xxx = startVal;
             REDEFINE startVal = startVal + 1];

    color(red);
    color(yellow);
    color(orange);
    color(green);
    color(blue);
    color(violet);

defines the identifiers "red", "yellow", "orange",
"green", "blue", and "violet" to have the values
1, 2, 3, 4, 5, and 6, respectively.  For example:

    color(red);

expands to:

    DEFINE red = startVal;
    REDEFINE startVal = startVal + 1;

which is equivalent to:

    DEFINE red = 1;
    REDEFINE startVal = 1 + 1;

This makes the value of startVal equal to 2, so that
when "color(yellow)" is encountered, the identifier
"yellow" is defined as 2, and "startVal" redefined as
3, etc.
```

Example 13.3-2. A Macro to Define a Series of Values

Bracketed text macros in MAINSAIL may be used in place of short procedures (although inline procedures are usually better for this purpose; see Section 1.9 of part II of the "MAINSAIL Tutorial"). For example, the identifier "isInRange" of Example 13.3-3 may be used the same way whether it is declared as a procedure or defined as a macro (except that the macro form evaluates i twice. This is not important if i is a simple variable, but may be if, e.g., i is a procedure call). A macro call usually takes up more space in an object module than a procedure call, but is often faster at execution time.

```
For an integer i, "isInRange(i)" returns the same value
whether "isInRange" is declared as:

    PROCEDURE isInRange (INTEGER i);
    RETURN(i > 50 AND i < 100);

or defined as:

    DEFINE isInRange(i) = [(i > 50 AND i < 100)];
```

Example 13.3-3. A Macro Used Instead of a Procedure

13.4. Comparison Chains

Comparisons chains may be used to make a series of comparisons more readable. For example, in Example 13.3-3, "i > 50 AND i < 100" could be replaced with "50 < i < 100". In general, a chain like "e1 op1 e2 op2 e3 op3 e4..." may replace "e1 op1 e2 AND e2 op2 e3 AND e3 op3 e4...", where the ei represent expressions and opi comparison operators or bits test operators ("TST", "TSTA", "NTST", and "NTSTA"). See the "MAINSAIL Language Manual" for more details.

13.5. Conditional Compilation

Conditional compilation may be used to skip over or select portions of program text based on conditions that can be evaluated at compiletime. Text is selected with the "IFC" compiler directive, the syntax for which is:

```
          IFC <compiletime condition> THENC
            <text to select if condition is true> ENDC
```

or:

```
       IFC <compiletime condition> THENC
         <text to select if condition is true>
       ELSEC <text to select if condition is false> ENDC
```

or:

```
 IFC <condition1> THENC
   <text to select if condition1 is true>
 $EFC <condition2> THENC
   <text to select if condition1 is false and condition2 true>
 ELSEC <text to select if condition1 and condition2 are false>
 ENDC
```

Any number of "$EFC" parts may appear, analogously to "EF" in an If Statement.

These constructs are analagous to the runtime statement selectors:

```
          IF <runtime condition> THENB
            <code to execute if condition is true> END
```

and:

```
         IF <runtime condition> THENB
           <code to execute if condition is true>
         EB <code to execute if condition is false> END
```

and:

```
 IF <condition1> THENB
   <code to execute if condition1 is true>
 EF <condition2> THENB
   <code to execute if condition1 is false and condition2 true>
 EB <code to execute if condition1 and condition2 are false> END
```

The text selected by an "IFC" need not be statements; declarations, expression or statement fragments, macro definitions, or entire procedures are often surrounded by "IFC" and "ENDC". Inside text that is not selected, IFC's, $EFC's, ELSEC's, and ENDC's must be properly matched.

13.6. Interactive Macro Equates and the "MESSAGE" Compiler Directive

The MAINSAIL compiler can be made to prompt for a macro body during compilation. If, instead of an equals sign and a macro body, a macro equate contains just a string constant following the identifier to be defined, the string constant is written to logFile and the line read from cmdFile is used as the macro body.

The "MESSAGE" compiler directive writes a message to logFile during a compilation. Its format is:

```
MESSAGE <string constant expression>;
```

A compilation of the file of Example 13.6-1 produces a dialogue like that shown in Example 13.6-2. Note the use of the "NOGENCODE" compiler subcommand to suppress generation of an object module. Compiler subcommands are described in the "MAINSAIL Compiler User's Guide".

```
BEGIN "nichts"

DEFINE i "The integer i should be: ";

MESSAGE "i is " & cvs(i);

IFC i > 100 THENC
MESSAGE "i is more than a hundred";
ELSEC
MESSAGE "i is not too big";
ENDC

END "nichts"
```

Example 13.6-1. Interactive Macro Equates and "MESSAGE"

The example of Example 13.6-3 is a rather clever (and not very common) use of recursive macros. It is a program that adds pairs of numbers at compiletime rather than at runtime.

```
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): nichts.msl,<eol>
> nogencode<eol>
> <eol>
Opening intmod for $SYS...

nichts.msl 1
The integer i should be: 2<eol>

i is 2

i is not too big

Objmod for NICHTS not generated
Intmod for NICHTS not stored

compile (? for help): nichts.msl,<eol>
> nogencode<eol>
> <eol>
Opening intmod for $SYS...

nichts.msl 1
The integer i should be: 351<eol>

i is 351

i is more than a hundred

Objmod for NICHTS not generated
Intmod for NICHTS not stored

compile (? for help): <eol>
*
```

Example 13.6-2. Compilation of the File of Example 13.6-1

Recursive macros must be used with caution, since an endless recursion may cause the MAINSAIL compiler to go into an infinite loop trying to expand the macro.

```
BEGIN "nihil"

DEFINE recursiveAdd =
    [REDEFINE i1 "First integer (0 to stop): ";
     IFC i1 THENC
     REDEFINE i2 "Second integer: ";
     MESSAGE "Their sum is " & cvs(i1 + i2);
     recursiveAdd
     ENDC];

recursiveAdd # Call the recursive macro

END "nihil"
```

Example 13.6-3. Use of Recursive Macros and Interactive Definition

The program fragment of Example 13.6-4 uses interactive definition to determine whether a macro called "assert" should perform a test or do nothing. Presumably the test should be performed while the program is under development but can be removed once the program has been debugged. This can be done by answering "FALSE" instead of "TRUE" to the compiletime question; there is no need to edit the source file. Note how the macro parameter is substituted even within string quotes in the macro body.

Under some conditions, a macro argument with an unusual format may need to be enclosed in square brackets. See the "MAINSAIL Language Manual" for details.

```
DEFINE doDebugging
    "Enable assertion testing (TRUE or FALSE): ";

IFC doDebugging THENC
DEFINE assert(condition) =
    [BEGIN IF NOT condition THEN
        errMsg("Assertion failed: condition") END];
ELSEC
# No code is generated for the empty macro body:
DEFINE assert(condition) = [];
ENDC

        .
        .
        .

POINTER(someClass) p;

        .
        .
        .

assert(p NEQ NULLPOINTER);
    # If p is NULLPOINTER and doDebugging was defined
    # as TRUE, then an error message will be generated
    # that says "Assertion failed: p NEQ NULLPOINTER".
```

Example 13.6-4. Use of Interactive Definition to Determine Whether Debugging Tests Are to Be Performed

13.7. Compiletime Equivalents of Iterative and Case Statements

Just as "IFC" provides a compiletime analogue of the runtime If Statement, so "$DOC" and "$CASEC" provide compiletime analogues of the runtime Iterative and Case Statements. For example, the repeated addition of Example 13.6-3 could be performed in an iterative instead of recursive fashion as shown in Example 13.7-1. As with their runtime equivalents, compiletime iteration is often easier to understand than recursion.

The "MAINSAIL Language Manual" contains complete details on compiletime constructs.

```
BEGIN "nihil2"

$DOC REDEFINE i1 "First integer (0 to stop): ";
     IFC NOT i1 THENC $DONEC ENDC
     REDEFINE i2 "Second integer: ";
     MESSAGE "Their sum is " & cvs(i1 + i2); ENDC

END "nihil2"
```

Example 13.7-1. Use of Compiletime Iteration

13.8. Concatenation of Macros

Pieces of bracketed text may be concatenated together like strings. This allows macros to be written that create new identifiers; see Example 13.8-1. This is a capability that is not used very often in simple applications.

Bracketed text in macro definitions can be concatenated with string constants to produce even more complicated effects; see the explanation in the "MAINSAIL Language Manual" and Example 13.8-2. Macro parameters are substituted even within strings, as shown in the example with the strings "className" and "new(className);" in the body of newClassCode.

As may be apparent, overly clever use of macros can make MAINSAIL text nearly indecipherable.

```
If the macro newInteger is defined as:

    DEFINE newInteger(x) = [INTEGER x] & [Int];

then the text:

    newInteger(a);
    newInteger(b);
    newInteger(c);

expands to:

    INTEGER aInt;
    INTEGER bInt;
    INTEGER cInt;

thereby declaring the new identifiers aInt, bInt, and
cInt.

A macro with two parameters may be used to create an
identifier out of the identifiers.  For example, if the
macro newID is defined as:

    DEFINE newID(part1,part2) = [part1] & [part2];

then the text:

    newID(abc,def)

expands to:

    abcdef
```

Example 13.8-1. Macros That Create New Identifiers

13.9. Repeatable Macro Parameters, $numArgs, $arg, and $sArg

A few more macro constructs deserve mention, although there is not room to give a detailed explanation here:

```
It is possible to define a macro that defines an integer
code for a class and creates a list of statements that
allocate one pointer to each class on which the macro has
been called:

  DEFINE
      classCodeValue          =    1,
      allocationString        =    "",
      newClassCode(className) =    [
          DEFINE className] & [Code = classCodeValue;
          REDEFINE classCodeValue = classCodeValue + 1;
          REDEFINE allocationString = allocationString &
              "className" & Ptr := new(className);"
              & eol;
          REDEFINE doAllocations =
              [] & allocationString];

Following the above declarations, the macro calls:

        newClassCode(c1);
        newClassCode(c2);
        newClassCode(xyz);

have the effect of defining the identifiers "c1Code" as 1,
"c2Code" as 2, and "xyzCode" as 3; they also create a
string allocationString with value:

        c1Ptr := new(c1);
        c2Ptr := new(c2);
        xyzPtr := new(xyz);

and a macro doAllocations, the body of which is identical
to the value of allocationString.  The macro doAllocations
could be used in a statement, e.g.:

    IF NOT allocationsDone THENB doAllocations END;
```

Example 13.8-2. Concatenation of Bracketed Text and String Constants

- Macro parameters may be repeatable, using the keyword "REPEATABLE" as with procedure parameters.

- $numArgs returns the number of arguments passed for a repeatable macro parameter, or the number of items in a bracketed list (e.g., "$numArgs([a,b,c])" evaluates to 3).

- "$arg(v,i)" is the ith argument passed to a repeatable parameter v, or if v is a bracketed list, the ith element of the list. For example, "$arg([a,b,c],2)" evaluates to the text "b".

- $sArg works like arg, except that it returns a string constant instead of text.

The "MAINSAIL Language Manual" explains these facilities more thoroughly.

13.10. Common Macro Errors

It is easy to produce undesired effects through improper use of macros. Example 13.10-1 illustrates macros that may expand in the desired way in some contexts but not in others.

To avoid the errors of Example 13.10-1, use the following rules when writing a macro body:

1. A macro parameter used as the operand of an operator in the macro body should be enclosed in parentheses to prevent unexpected interaction between the operator and components of the macro argument.

2. A macro body that consists of one or more statements should not contain a trailing semicolon. The semicolon, if needed, should be supplied at the point of the macro call.

3. A macro body that consists of two or more statements or is an If Statement should be turned into a single Begin Statement by enclosing the body in a "BEGIN"-"END" pair.

The macros in Example 13.10-1 should be defined as in Example 13.10-2.

```
DEFINE
    dbl(a)        =      [(a) * 2],
    doMsg(a)      =      [write(logFile,a)],
    rd(f,a)       =      [BEGIN read(f,a); ttyWrite(a) END],
    checkA        =      [BEGIN IF a TST c THEN d := e END];
```

Example 13.10-2. Corrected Definitions of Macros

```
Given the definitions:

DEFINE
    dbl(a)          =        [a * 2],
    doMsg(a)        =        [write(logFile,a);],
    rd(f,a)         =        [read(f,a); ttyWrite(a)],
    checkA          =        [IF a TST c THEN d := e];
```

Example 13.10-1. Expansion-Context-Dependent Macros (continued)

```
the following problems occur:

    . The macro call "dbl(v + 1)" expands to:

        v + 1 * 2

      which is equivalent to:

        v + (1 * 2)

      rather than to the desired:

        (v + 1) * 2

    . The macro call "IF b THEN doMsg("error") ELSE ..."
      expands to:

        IF b THEN write(logFile,"error"); ELSE ...

      which is syntactically incorrect because of the
      semicolon before the "ELSE".

    . The macro call "IF b THEN rd(f,i)" expands to:

        IF b THEN read(f,i); ttyWrite(i)

      which does not have the desired effect, which is:

        IF b THENB read(f,i); ttyWrite(i) END

    . The macro call "IF b THEN checkA ELSE ..." expands
      to:

        IF b THEN IF a TST c THEN d := e ELSE ...

      which has the undesired effect of matching the
      "ELSE" with the second instead of the first "IF".
```

Example 13.10-1. Expansion-Context-Dependent Macros (end)

13.11. Exercises

Exercise 13-1.

Write a macro "strDecls" that takes an integer parameter N. When called, the macro expands to a series of N string variable declarations. The identifiers declared are "s1", "s2", ..., "s<N>". For example, "strDecls(4)" should expand as shown in Example 13.11-1.

You may want to use some of the keywords "$FORC", "$DONEC", and "$CONTINUEC", which provide compiletime analogues to "FOR", "DONE", and "CONTINUE".

```
STRING s1;
STRING s2;
STRING s3;
STRING s4;
```

Example 13.11-1. Expansion of "strDecls(4)"

14. Indirect Access to Modules; Bound Data Sections

This chapter describes modules, which are the basic MAINSAIL unit of compilation. Modules permit separate compilation and information hiding; they are also a runtime data structure. Modules often exist as bound data sections, which allows the names of module interface fields to be used without an explicit module prefix. Access to modules with a module prefix or with no prefix at all is called "indirect access" to modules; Chapter 15 describes uses of modules accessed by means of pointers (direct access).

14.1. The Role of Modules

Every one of the sample programs shown so far has consisted of a single module. It has been mentioned, however, that programs can consist of multiple modules. In fact, the sample programs shown so far do make implicit use of the modules of the MAINSAIL runtime system, since many system procedures reside in separate modules (rather than being compiled into a user module that needs them). You can write modules that explicitly communicate with each other or manipulate each other.

The MAINSAIL compiler outputs one object module (objmod) per source module; it cannot be made to output a partial objmod (except in the case of incremental recompilation, as described in the "MAINSAIL Compiler User's Guide"), so modules are referred to as the "unit of compilation".

More than one module may occur in a source file. When the MAINSAIL compiler compiles a source file with more than one source module in it, it compiles the modules in the order in which they appear in the file. It compiles each module from scratch; i.e., it does not remember any definitions or declarations encountered in previous modules in the same file. It is customary to separate modules in the same file with an end-of-page (eop) character, which is a separator character in MAINSAIL like the end-of-line character (or characters). If the file shown in Example 14.1-1 were named "twomod.msl", its compilation would look like Example 14.1-2.

Every module has zero or more "interface fields", which are like the fields of an ordinary record, except that module interface fields may be procedure fields as well as data fields. The modules encountered so far have not had any interface fields. Interface fields must be explicitly declared in order to be accessible from other modules. The interface of a module must be declared in the module itself and in any modules that use the interface fields; furthermore, the interface declarations must match, or MAINSAIL reports an error when the modules are brought into memory to be used.

```
BEGIN "mod1"

INITIAL PROCEDURE;
write(logFile,"Module 1" & eol);

END "mod1"
<page mark>
BEGIN "mod2"

INITIAL PROCEDURE;
write(logFile,"Module 2" & eol);

END "mod2"
```

Example 14.1-1. Two Modules in the Same Source File

```
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): twomod.msl<eol>
Opening intmod for $SYS...

twomod.msl 1 2
Objmod for MOD1 stored on mod1-xyz.obj
Intmod for MOD1 not stored
Opening intmod for $SYS...

Objmod for MOD2 stored on mod2-xyz.obj
Intmod for MOD2 not stored

compile (? for help): <eol>
*
```

Example 14.1-2. Compilation of the File of Example 14.1-1

Module interface fields provide "information hiding", since other modules know no more about a given module than what its interface declares to be available. In a large programming project, the form and function of the interface fields of the various modules composing the project may be specified early on, and the modules then developed separately in parallel. A program composed of many small parts with well-defined interfaces is often easier to debug than a single large, homogeneous program.

14.2. Binding a Module Explicitly

Before a module "indirectly" accesses (direct access is discussed in Chapter 15) a data interface field of another module, it must ensure that the other module is "bound", i.e., available for use. The simplest way to do this is to call the procedure "bind", which takes the module identifier of the module to be bound as its argument. Examples 14.2-1 and 14.2-2 show two modules, ITF1 and ITF2, which access each other's interface fields. These modules are shown residing in separate source files; it does not in fact matter whether they reside in the same source file or whether they are compiled at the same time, provided that both have been compiled by the time either is executed.

As shown in Examples 14.2-1 and 14.2-2, the module declarations are similar to class declarations. The name of the module follows the keyword "MODULE" and precedes a parenthesized list of fields.

A field of a module need be prefixed by its module name and a period only if it is the same as a local or outer identifier or interface field name of the current module, or if more than one module outside the current module has a field of the same name. In Examples 14.2-1 and 14.2-2, all references to fields of other modules are unambiguous, a common case.

A MAINEX session in which ITF1 is executed twice is shown in Example 14.2-3. Because of the way binding works, the module does not execute the same way twice in a row; ITF2 is already bound before the second execution.

```
BEGIN "itf1"

MODULE itf1 (
    PROCEDURE proc1 (MODIFIES INTEGER i2);
);

MODULE itf2 (
    INTEGER i2;
    PROCEDURE proc2 (STRING whereFrom);
);

PROCEDURE proc1 (MODIFIES INTEGER i2);
# Note that "i2" within this procedure refers to the
# parameter, not to the interface variable of ITF2; to
# access the latter here, you MUST say "itf2.i2"
.- i2; # Negate the integer



INITIAL PROCEDURE;
BEGIN
write(logFile,"ITF1: initial procedure" & eol);
bind(itf2);
itf2.i2 := 4; # Could write just "i2 := 4", since there
              # are no other i2's declared
proc2("ITF1, before calling PROC1");
    # Could say "itf2.proc2" instead of just "proc2"
proc1(i2); # Could just as well say "itf1.proc1(itf2.i2)"
itf2.proc2("ITF1, after calling PROC1");
    # Could say just "proc2" instead of "itf2.proc2"
write(logFile,"ITF1: end of initial procedure" & eol);
END;

END "itf1"
```

Example 14.2-1. The Module ITF1

```
BEGIN "itf2"

MODULE itf2 (
    INTEGER i2;
    PROCEDURE proc2 (STRING whereFrom);
);

MODULE itf1 (
    PROCEDURE proc1 (MODIFIES INTEGER i2);
);

PROCEDURE proc2 (STRING whereFrom);
write(logFile,"Proc2, from ",whereFrom,
    "; itf2.i2 = ",i2,eol);



INITIAL PROCEDURE;
BEGIN
write(logFile,"ITF2: initial procedure" & eol);
proc2("ITF2, before binding ITF1");
bind(itf1);
i2 := -3; # Could write "itf2.i2 := -3"
proc1(i2); # Could just as well say "itf1.proc1(itf2.i2)"
proc2("ITF2, after calling PROC1");
write(logFile,"ITF2: end of initial procedure" & eol);
END;

END "itf2"
```

Example 14.2-2. The Module ITF2

```
*itf1<eol>
ITF1: initial procedure
ITF2: initial procedure
Proc2, from ITF2, before binding ITF1; itf2.i2 = 0
Proc2, from ITF2, after calling PROC1; itf2.i2 = 3
ITF2: end of initial procedure
Proc2, from ITF1, before calling PROC1; itf2.i2 = 4
Proc2, from ITF1, after calling PROC1; itf2.i2 = -4
ITF1: end of initial procedure
*itf1<eol>
ITF1: initial procedure
Proc2, from ITF1, before calling PROC1; itf2.i2 = 4
Proc2, from ITF1, after calling PROC1; itf2.i2 = -4
ITF1: end of initial procedure
*
```

Example 14.2-3. Execution of ITF1

14.3. Bound Modules, and How Example 14.2-3 Works

Every module that has been brought into memory during an execution has a "control section" and zero or more "data sections". The control section consists of the data output by the MAINSAIL compiler into the objmod, and contains (among other things) the machine instructions that perform the actions of the statements in the module. A data section is allocated when the control section is brought into memory (it may also be explicitly reallocated thereafter; see Chapter 15). A data section contains (among other things) the outer, own, and interface variables of the module.

Every data section contains hidden, unnamed "implicit module pointers" to every module of which it references interface fields by means of indirect access, i.e., without prefixing the interface field with a pointer. A runtime error results if a module (module A) tries to indirectly access a data interface field of another module (module B) before module A's implicit module pointer to B has been initialized. When A binds B, A's implicit module pointer to B is set to point to B's data section. Furthermore, if B is not presently bound, a data section (the "bound data section") of B is allocated, and B's initial procedure (if it exists) is executed.

When a module's data section is allocated, the MAINSAIL runtime system automatically initializes any implicit module pointers in it that refer to modules that are already bound. If the new data section is itself a bound data section, it also initializes implicit module pointers in other modules to point to itself.

For every module in memory, there is exactly one control section and at most one bound data section (the allocation of nonbound data sections, of which there may be none or more than one, is discussed in Chapter 15). Unless the bound data section is explicitly removed from memory by means of the system procedure "unbind" or the system procedure "dispose", it remains in memory until the end of the MAINSAIL execution.

When MAINEX executes a module, it first checks to see whether it is already bound, and if so issues an error message. Otherwise, it binds it, which causes the bound data section to be allocated and the initial procedure to be executed; then, when the module's initial procedure finishes executing, MAINEX calls "dispose" to remove the data section (but not the control section) from memory. However, MAINEX does not unbind or dispose any module bound by the executed module. Therefore, since the first execution of ITF1 in Example 14.2-3 binds ITF2, and does not unbind it, ITF2 remains bound (and its data section remains in existence) after ITF1 has finished executing. Therefore, the call "bind(itf2)" made during the second execution of ITF1 does not reallocate ITF2's data section, and does not execute ITF2's initial procedure. That is why the messages printed out by ITF2's initial procedure do not appear in the second execution of ITF1.

Diagrammatically, the initial state of memory looks like Example 14.3-1 (there are other things in memory, like procedures' local variables, which are not shown). Note that in real life, control sections, data sections, and records of the bound module list may appear in any order in memory; they are not necessarily maintained contiguously. Also, the runtime system maintains other data structures associated with modules as well as those shown, but the ones shown are the only ones you need to know about in order to use modules correctly.

```
              Control sections
+--------+ +--------+          +--------+
|runtime | |runtime |          |runtime |
|module  | |module  |          |module  |
|control | |control |  ......  |control |
|section | |section |          |section |
|#1      | |#2      |          |#n      |
+--------+ +--------+          +--------+
    ^          ^                   ^
    |          |                   |
    |          |   Data sections   |
+---+---+ +---+---+           +---+---+
|runtime | |runtime |          |runtime |
|module  | |module  |          |module  |
|data    | |data    |  ......  |data    |
|section | |section |          |section |
|#1      | |#2      |          |#n      |
|(bound) | |(bound) |          |(bound) |
+--------+ +--------+          +--------+
```

Example 14.3-1. Before Execution of ITF1 or ITF2

After the first "itf1" command to MAINEX, ITF1's control section is read into memory, and its data section is allocated. ITF1's implicit module pointer to ITF2 is initially Zero. See Example 14.3-2.

When ITF1 issues the call "bind(itf2)", ITF2's control section is brought into memory. ITF2's implicit module pointer to ITF1 is initialized to point to ITF1's data section, since ITF1 is already in memory; furthermore, since ITF2 is being allocated, ITF1's implicit module pointer to ITF2 is made to point to ITF2's data section. See Example 14.3-3. When a module A's implicit module pointer to module B has been initialized, module A is said to "have linkage" to module B. In Example 14.3-3, ITF1 and ITF2 have established linkage to each other.

ITF2's initial procedure executes, then control returns to ITF1's initial procedure, which finishes. Control then returns to MAINEX, which disposes ITF1's data section. After ITF1's data section has been disposed, memory looks like Example 14.3-4. Note that the disposal of the bound data section of ITF1 causes all implicit module pointers to ITF1 to be set to Zero.

```
                   Control sections
+--------+ +--------+              +--------+ +--------+
|runtime | |runtime |              |runtime | |        |
|module  | |module  |              |module  | |ITF1    |
|#1      | |#2      |              |#n      | |control |
|control | |control |    ......    |control | |section |
|section | |section |              |section | |        |
+--------+ +--------+              +--------+ +--------+
    ^          ^                       ^          ^
    |          |                       |          |
    |          |                       |          |
    |          |                       |      +---+---+
    |          |   Data sections       |      |ITF1    |
+---+---+  +---+---+               +---+---+  |data    |
|runtime | |runtime |              |runtime | |section |
|module  | |module  |              |module  | |(bound) |
|#1      | |#2      |              |#n      | +--------+
|data    | |data    |    ......    |data    | |modPtr  |
|section | |section |              |section | |to ITF2 |
|(bound) | |(bound) |              |(bound) | |   o    |
+--------+ +--------+              +--------+ +--------+
```

Example 14.3-2. After the Allocation of ITF1's Data Section

At this point, ITF2 appears to be inaccessible; no runtime module has an implicit module pointer to it. However, it is not subject to garbage collection, since bound data sections are not collected. In Example 14.2-3, ITF2's control and data sections were still present in memory when ITF1 issued a "bind(itf2)" during its second execution. ITF2's data section was not reallocated, and its initial procedure was not reexecuted; all the call to "bind" did was to set ITF1's implicit module pointer to ITF2. Example 14.3-5 shows the situation after ITF1 has bound ITF2 during ITF1's second execution.

After ITF1 executes the second time, its data section is disposed as before.

If "itf2" were typed to the MAINEX prompt after Example 14.2-3, MAINEX would issue an error message because a bound data section for ITF2 is still in memory.

```
                Control sections
+-------+ +-------+          +-------+ +-------+ +-------+
|runtime| |runtime|          |runtime| |       | |       |
|module | |module |          |module | |ITF1   | |ITF2   |
|#1     | |#2     |          |#n     | |control| |control|
|control| |control|  ......  |control| |section| |section|
|section| |section|          |section| |       | |       |
+-------+ +-------+          +-------+ +-------+ +-------+
    ^         ^                  ^         ^         ^
    |         |                  |         |         |
    |         |                  |         |         |
    |         |                  |     +---+---+ +---+---+
    |         |   Data sections  |     |ITF1   | |ITF2   |
+---+---+ +---+---+          +---+---+ |data   | |data   |
|runtime| |runtime|          |runtime| |section| |section|
|module | |module |          |module | |(bound)| |(bound)|
|#1     | |#2     |          |#n     | +-------+ +-------+
|data   | |data   |  ......  |data   | |modPtr | |modPtr |
|section| |section|          |section| |to ITF2| |to ITF1|
|(bound)| |(bound)|          |(bound)| |   +---+>|   +   |
+-------+ +-------+          +-------+ +-------+ +---+---+
                                           ^         |
                                           +---------+
```

Example 14.3-3. After ITF1 Binds ITF2

```
                 Control sections
+--------+ +--------+          +--------+ +--------+ +--------+
|runtime | |runtime |          |runtime | |        | |        |
|module  | |module  |          |module  | |ITF1    | |ITF2    |
|#1      | |#2      |          |#n      | |control | |control |
|control | |control |  ......  |control | |section | |section |
|section | |section |          |section | |        | |        |
+--------+ +--------+          +--------+ +--------+ +--------+
    ^          ^                   ^                      ^
    |          |                   |                      |
    |          |                   |                      |
    |          |                   |                 +---+----+
    |          |   Data sections   |                 |ITF2    |
+---+----+ +---+----+          +---+----+            |data    |
|runtime | |runtime |          |runtime |            |section |
|module  | |module  |          |module  |            |(bound) |
|#1      | |#2      |          |#n      |            +--------+
|data    | |data    |  ......  |data    |            |modPtr  |
|section | |section |          |section |            |to ITF1 |
|(bound) | |(bound) |          |(bound) |            |   o    |
+--------+ +--------+          +--------+            +--------+
```

Example 14.3-4. After MAINEX Has Disposed ITF1

```
                 Control sections
+--------+ +--------+           +--------+ +--------+ +--------+
|runtime | |runtime |           |runtime | |        | |        |
|module  | |module  |           |module  | |ITF1    | |ITF2    |
|#1      | |#2      |           |#n      | |control | |control |
|control | |control |  ......   |control | |section | |section |
|section | |section |           |section | |        | |        |
+--------+ +--------+           +--------+ +--------+ +--------+
     ^          ^                    ^          ^          ^
     |          |                    |          |          |
     |          |                    |          |          |
     |          |                    |     +----+---+ +----+---+
     |          |   Data sections    |     |ITF1    | |ITF2    |
+----+---+ +----+---+           +----+---+ |data    | |data    |
|runtime | |runtime |           |runtime | |section | |section |
|module  | |module  |           |module  | |(bound) | |(bound) |
|#1      | |#2      |           |#n      | +--------+ +--------+
|data    | |data    |  ......   |data    | |modPtr  | |modPtr  |
|section | |section |           |section | |to ITF2 | |to ITF1 |
|(bound) | |(bound) |           |(bound) | |    +---+>|    +   |
+--------+ +--------+           +--------+ +--------+ +----+---+
                                                ^          |
                                                +----------+
```

Example 14.3-5. After the Second Execution of ITF1 Binds ITF2

14.4. Module Swapping

When the MAINSAIL runtime system runs low on memory space, it may remove some control sections from memory. It reads the control sections back in again if and when it needs to do so. Module control section swapping takes place automatically; it is completely invisible to your programs. Data sections and other data are never removed from memory in this fashion; if your program runs low on memory, you must explicitly organize your program to write data to a temporary file. See Section 2.1 of part II of the "MAINSAIL Tutorial" for more details on swapping.

In order to see when control sections of modules are brought into memory, you may give the MAINEX subcommand "SWAPINFO". In Example 14.4-1, the executions of Example 14.2-3 are repeated with the "SWAPINFO" subcommand given.

```
*itf1,<eol>
Enter subcommands (? for help).
>swapinfo<eol>
><eol>
Swapping in ITF1
Done swapping in ITF1
ITF1: initial procedure
Swapping in ITF2
Done swapping in ITF2
ITF2: initial procedure
Proc2, from ITF2, before binding ITF1; itf2.i2 = 0
Proc2, from ITF2, after calling PROC1; itf2.i2 = 3
ITF2: end of initial procedure
Proc2, from ITF1, before calling PROC1; itf2.i2 = 4
Proc2, from ITF1, after calling PROC1; itf2.i2 = -4
ITF1: end of initial procedure
*itf1<eol>
ITF1: initial procedure
Proc2, from ITF1, before calling PROC1; itf2.i2 = 4
Proc2, from ITF1, after calling PROC1; itf2.i2 = -4
ITF1: end of initial procedure
*
```

Example 14.4-1. Execution of ITF1 with the "SWAPINFO" MAINEX Subcommand

14.5. When Explicit Binding Is Necessary

Until now it has not been mentioned that calling a procedure in another module implicitly binds that module. Explicit binding need occur only if a DATA (i.e., non-procedure) field of a module is to be accessed before linkage is otherwise established to the module. This means that the "bind(itf1)" that appears in the initial procedure of ITF2 in Example 14.2-2 is not really necessary. The first access to ITF1 in ITF2's initial procedure is the call to proc1, which automatically binds ITF1. However, the "bind(itf1)" makes it clear to the reader what is going on. It does not do any harm, and if the program is rewritten to access a data field of ITF1 before a procedure field, the "bind" will prevent an error from occurring.

14.6. Declarations Shared by Several Modules and the "SOURCEFILE" Compiler Directive

The "SOURCEFILE" compiler directive causes the MAINSAIL compiler to continue the current compilation in another file. The format of the "SOURCEFILE" directive is:

```
SOURCEFILE <file name>;
```

Text is read from the named file (the "sourcefiled" file) as if the contents of the file had been encountered instead of the "SOURCEFILE" directive. When the compiler is finished with the sourcefiled file, it resumes compiling on the line following the "SOURCEFILE" directive. "SOURCEFILE" directives may be nested; i.e., a sourcefiled file may contain other "SOURCEFILE" directives.

The "DONESCAN" compiler directive causes the compiler to return from a sourcefiled file to the original file; so does the end of the sourcefiled file. The format of the "DONESCAN" directive is:

```
DONESCAN;
```

The modules of Examples 14.2-1 and 14.2-2 each declare both modules ITF1 and ITF2. If a change is made to either interface, the source text for both modules must be updated. It would be more convenient to maintain the interface declarations in a single file, so that updates could be made in only one place. Examples 14.6-1, 14.6-2, and 14.6-3 shows how the interface declarations may be placed in a file called "decls" that is sourcefiled by the source files for the modules ITF1 and ITF2.

In a large programming project with many modules, all the module interface declarations are usually maintained in a single file. Consider a program consisting of the modules A, B, C, D, E, and F. Each has interface variables and procedures. If the module A is used by C, D, and F, and a separate copy of A's interface declaration is maintained in the source modules for A, C, D, and F, then any change to A's interface would have to be propagated to four places. As

```
BEGIN "itf1"

SOURCEFILE "decls"; # Get interface declarations for
                    # ITF1 and ITF2

PROCEDURE proc1 (MODIFIES INTEGER i2);
# Note that "i2" within this procedure refers to the
# parameter, not to the interface variable of ITF2; to
# access the latter here, you MUST say "itf2.i2"
.- i2; # Negate the integer



INITIAL PROCEDURE;
BEGIN
write(logFile,"ITF1: initial procedure" & eol);
bind(itf2);
itf2.i2 := 4; # Could write just "i2 := 4", since there
              # are no other i2's declared
proc2("ITF1, before calling PROC1");
    # Could say "itf2.proc2" instead of just "proc2"
proc1(i2); # Could just as well say "itf1.proc1(itf2.i2)"
itf2.proc2("ITF1, after calling PROC1");
    # Could say just "proc2" instead of "itf2.proc2"
write(logFile,"ITF1: end of initial procedure" & eol);
END;

END "itf1"
```

Example 14.6-1. ITF1 with the "SOURCEFILE" Directive

anyone who has worked on a large programming project has observed, coordinating a group of programmers to make mutually consistent changes in four places can be very difficult! Therefore, the interfaces for all six modules should reside in a single common declarations file. There is no penalty for declaring the interface of the module A in the module B if B does not use A, so it does no harm to have B sourcefile the file in which A's interface is declared.

Even better than a common sourcefile for many purposes may be a common intmod; see Section 20.3.

```
BEGIN "itf2"

SOURCEFILE "decls"; # Get interface declarations for
                    # ITF1 and ITF2

PROCEDURE proc2 (STRING whereFrom);
write(logFile,"Proc2, from ",whereFrom,
    "; itf2.i2 = ",i2,eol);



INITIAL PROCEDURE;
BEGIN
write(logFile,"ITF2: initial procedure" & eol);
proc2("ITF2, before binding ITF1");
bind(itf1);
i2 := -3; # Could write "itf2.i2 := -3"
proc1(i2); # Could just as well say "itf1.proc1(itf2.i2)"
proc2("ITF2, after calling PROC1");
write(logFile,"ITF2: end of initial procedure" & eol);
END;

END "itf2"
```

Example 14.6-2. ITF2 with the "SOURCEFILE" Directive

```
MODULE itf1 (
    PROCEDURE proc1 (MODIFIES INTEGER i2);
);

MODULE itf2 (
    INTEGER i2;
    PROCEDURE proc2 (STRING whereFrom);
);
```

Example 14.6-3. The Sourcefiled File "decls"

14.7. Exercises

Exercise 14-1.

Create a calculator program that can accept either Reverse Polish Notation or the standard operator order. There should be two "top-level" modules, RPN and STD, that implement the two orders; the user may choose either by typing its name to MAINEX. Instead of performing arithmetic themselves, both RPN and STD should call a common third module, ARITH, and pass it a data structure describing which arithmetic operations to perform.

The MAINSAIL system procedure $removeInteger may be useful for removing an integer from a string. $removeLeadingBlankSpace removes blanks and tab characters from the beginning of a string.

15. Direct Access to Modules; Nonbound Data Sections

This chapter describes the use of pointers to manipulate module data sections explicitly. Access to fields of a module's data section by means of explcit pointers is called "direct access" to the module. Explicit manipulation of data sections can result in an "object-oriented" programming style.

15.1. The Nonbound Data Section

The modules of Chapter 14 were brought into memory by means of the procedure "bind", which also created the data sections for the modules. The procedure "new" may also be used to establish a control section for a module (if necessary) and create a corresponding data section. Such a call to "new" results in a nonbound data section. Nonbound data sections must always be accessed by means of an explicit pointer ("direct access"); i.e., the module field name must be preceded by a pointer (not a module name, as declared with the keyword "MODULE") and a period. Bound data sections were provided in MAINSAIL so that not all intermodule accesses would require the use of an explicit data section pointer prefix; nonbound data sections are, however, a more powerful facility.

Unlike "bind", "new" always creates a new data section. "bind" creates a new (bound) data section only if no bound data section currently exists for the module; "new" creates a nonbound data section whether or not there exists a bound data section or other nonbound data sections. "new" returns a pointer to the new data section.

15.2. HSHMOD Example of Nonbound Data Sections

In order to classify properly a pointer to a data section with procedure interface fields, it is necessary to declare a class that contains procedure fields. Since "new" for a data section takes a module name (the name of the module that provides the control section corresponding to the data section) rather than a class name as its argument, it is useful to declare a module of the class; see Example 15.2-2. The declarations of Example 15.2-2 are available to any MAINSAIL program, since HSHMOD is a standard utility module; to pick up the declarations, your program must include the code shown in Figure 15.2-1 (the real HSHMOD actually includes a few facilities not shown here).

HSHMOD maintains a hash table by means of the procedures shown in Example 15.2-2. HSHMOD is described in detail in the "MAINSAIL Utility User's Guide"; the (partial) source text for the utility module HSHMOD is shown in Example 15.2-3.

```
REDEFINE $scanName = "hshHdr";
SOURCEFILE "(system library)";
```

Figure 15.2-1. Code to Pick Up HSHMOD Declaration

```
# prefix class for hashed records
CLASS hashedRecord
    (STRING key; POINTER(hashedRecord) link);

# explicit class so user can classify pointers to it
CLASS hshCls (
    PROCEDURE hashInit (OPTIONAL INTEGER tableSize);
    PROCEDURE hashEnter (POINTER(hashedRecord) p);
    POINTER(hashedRecord)
    PROCEDURE hashLookUp (STRING key);
    POINTER(hashedRecord)
    PROCEDURE hashRemove (STRING key);
    POINTER(hashedRecord)
    PROCEDURE hashNext (POINTER(hashedRecord) p);
);

MODULE(hshCls) hshMod;
    # This is the syntax for declaring a module interface
    # as a named class:
    # MODULE(<class name>) <module name>
```

Example 15.2-2. Partial Declaration of HSHMOD and Associated Classes in the MAINSAIL System Source Library

It is intended that each hash table kept by a program be maintained as a separate HSHMOD data section. The module of Example 15.2-4 uses HSHMOD in this way; the three pointers byName, byAdr, and byPhone each point to a separate HSHMOD data section.

```
BEGIN "hshMod"

# this module maintains a general-purpose hash table

REDEFINE $scanName = "hshHdr"; # Pick up interface
SOURCEFILE "(system library)"; # declarations

DEFINE
    numCharsToHash      =       4,
    defaultTableSize    =       131;

INTEGER numberOfHashLists;

POINTER(hashedRecord) ARRAY(0 TO *) hashList;

#       +--------+
#    0  |      >-+----> linked list of all records
#       +--------+         whose keys hash to 0
#    1  |      >-+----> linked list of all records
#       +--------+         whose keys hash to 1
#    2  |      >-+----> linked list of all records
#       +--------+         whose keys hash to 2
#    3  |      >-+----> linked list of all records
#       +--------+         whose keys hash to 3
#       |        |
#
#       |        |
#       +--------+

PROCEDURE hashInit (OPTIONAL INTEGER tableSize);
BEGIN
IF tableSize LEQ 0 THEN tableSize := defaultTableSize;
new(hashList,0,tableSize - 1);
numberOfHashLists := tableSize;
END;



INTEGER PROCEDURE hash (STRING key);
BEGIN
INTEGER h,i,j;
```

Example 15.2-3. Partial Source Text for the MAINSAIL Utility Module HSHMOD (continued)

```
# s hashes to
#    (length(s) + 3 * char1 + 5 * char2 +
#       7 * char3 + 9 * char4)
#       MOD numberOfHashLists
#
# where chari represents ith character of s

h := length(key); i := h MIN numCharsToHash; j := 1;
WHILE i .- 1 GEQ 0 DO h .+ cRead(key) * (j .+ 2);
RETURN(h MOD numberOfHashLists) END;



POINTER(hashedRecord) PROCEDURE search
    (STRING key;
     PRODUCES OPTIONAL INTEGER hashValue;
     PRODUCES OPTIONAL POINTER(hashedRecord)
        beforeTarget);
BEGIN
POINTER(hashedRecord) target;

# general-purpose search procedure

IF NOT hashList THEN hashInit;  # automatic initialization

hashValue := hash(key);
beforeTarget := NULLPOINTER;
target := hashList[hashValue];

WHILE target AND target.key NEQ key DOB
    beforeTarget := target; target := target.link END;

RETURN(target) END;
```

Example 15.2-3. Partial Source Text for the MAINSAIL Utility Module HSHMOD (continued)

```
PROCEDURE hashEnter (POINTER(hashedRecord) p);
BEGIN # enter p at front of its hash list
INTEGER h;
IF NOT hashList THEN hashInit;
IF p THENB
    h := hash(p.key); p.link := hashList[h];
    hashList[h] := p END
EL errMsg("hashEnter: argument is NULLPOINTER") END;



POINTER(hashedRecord) PROCEDURE hashLookUp (STRING key);
RETURN(search(key)); # return record with given key
                     # (Zero if not found)



POINTER(hashedRecord) PROCEDURE hashRemove (STRING key);
BEGIN # remove record with given key
INTEGER                 h;
POINTER(hashedRecord)   target,beforeTarget;

IF target := search(key,h,beforeTarget) THEN
    IF beforeTarget THEN beforeTarget.link := target.link
    EL hashList[h] := target.link;

RETURN(target) END;



POINTER(hashedRecord) PROCEDURE hashNext
    (POINTER(hashedRecord) p);
BEGIN
INTEGER h;
POINTER(hashedRecord) q;

# generate next record in hashList (successive calls
# starting with p = NULLPOINTER will generate all records,
# then NULLPOINTER)
```

Example 15.2-3. Partial Source Text for the MAINSAIL Utility Module HSHMOD (continued)

```
IF NOT p THEN h := -1
EF q := p.link THEN RETURN(q)
EL h := hash(p.key);

DOB IF h .+ 1 GEQ numberOfHashLists THEN
        RETURN(NULLPOINTER);
    IF p := hashList[h] THEN RETURN(p) END END;

END "hshMod"
```

Example 15.2-3. Partial Source Text for the MAINSAIL Utility Module HSHMOD (end)

```
BEGIN "dirMod"

# This module maintains a directory of "people" by name,
# address and phone.  The user may enter new people from
# "TTY" or any file; remove people; change the
# information about a person; retrieve information by
# giving either a person's name, address or phone; and
# save the people directory on a file ("TTY" for a
# printout) in such a way that it can be later loaded and
# used.

# The hash module is used to store and retrieve the
# information.  Each person is stored as a record with
# prefix class hashedRecord so it can be manipulated by
# the hash module.  The remaining fields are name, adr,
# and phone.  Every person is stored three times, once
# each by name, adr, and phone.  This is done by
# utilizing three instances (separate data sections) of
# HSHMOD, each of which maintains a separate hash table.
# This example is not optimized for saving space since
# every person is represented by three nearly identical
# person records (only the hashedRecord fields differ).

REDEFINE $scanName = "hshHdr";
SOURCEFILE "(system library)"; # retrieve the declarations
```

Example 15.2-4. A Module That Uses HSHMOD (continued)

```
CLASS(hashedRecord) person (STRING name,adr,phone);

POINTER(hshCls)
    byName,                # stored by name
    byAdr,                 # stored by address
    byPhone;               # stored by phone

PROCEDURE doInit;
BEGIN # use default table size (no need to call hashInit)
byName := new(hshMod); byAdr := new(hshMod);
byPhone := new(hshMod);
END;


PROCEDURE doFlush;
# Discard current lists and reinitialize (discarded
# data sections will be reclaimed by garbage collector,
# since they become inaccessible)
BEGIN
byName := byAdr := byPhone := NULLPOINTER; doInit;
END;



PROCEDURE doOneEntry
    (POINTER(hshCls) p; STRING key,name,adr,phone);
BEGIN
POINTER(person) q;
q := new(person);
q.key := key; q.name := name; q.adr := adr;
q.phone := phone; p.hashEnter(q);
END;



PROCEDURE doEnter (STRING s);
BEGIN
STRING          name,adr,phone;
POINTER(person) p;

# s of form: name <tabs> address <tabs> phone
```

Example 15.2-4. A Module That Uses HSHMOD (continued)

```
name := scan(s,tab,discard); scan(s,tab,proceed!omit);
adr :=  scan(s,tab,discard); scan(s,tab,proceed!omit);
phone := s;

doOneEntry(byName,name,name,adr,phone);
doOneEntry(byAdr,adr,name,adr,phone);
doOneEntry(byPhone,phone,name,adr,phone);
END;



PROCEDURE doRestore (STRING fileName);
BEGIN
STRING                  s;
POINTER(textFile)       f;
open(f,fileName,input);
DOB read(f,s); IF NOT s THEN DONE; doEnter(s) END;
close(f) END;



PROCEDURE doSave (STRING fileName);
BEGIN
STRING                  s;
POINTER(textFile)       f;
POINTER(person)         p;

open(f,fileName,output); p := NULLPOINTER;

WHILE p := byName.hashNext(p) DO
    write(f,p.name,tab,p.adr,tab,p.phone,eol);

close(f) END;
```

Example 15.2-4. A Module That Uses HSHMOD (continued)

```
PROCEDURE doLookUp (POINTER(hshCls) p; STRING key);
BEGIN
POINTER(person) q;
IF NOT q := p.hashLookUp(key) THEN
    ttyWrite("Not found" & eol)
EL ttyWrite(
        "Name:" & tab,  q.name, eol &
        "Adr:"  & tab,  q.adr,  eol &
        "Phone:" & tab, q.phone,eol);
END;



PROCEDURE doRemove (STRING s);
BEGIN
POINTER(person) p;
IF p := byName.hashLookUp(s) THENB
    byName.hashRemove(p.name);
    byAdr.hashRemove(p.adr);
    byPhone.hashRemove(p.phone) END END;



PROCEDURE doChange (STRING s);
BEGIN
doRemove(scan(s,tab,discard));
scan(s,tab,proceed!omit);
doEnter(s) END;



INITIAL PROCEDURE;
BEGIN
STRING  s;

DEFINE
    x         = 0,
    def(name) = [REDEFINE x = x + 1; DEFINE name = x;];
```

Example 15.2-4. A Module That Uses HSHMOD (continued)

```
def(restoreCase)
def(enterCase)
def(removeCase)
def(nameCase)
def(adrCase)
def(phoneCase)
def(changeCase)
def(saveCase)
def(flushCase)
def(exitCase)

STRING ARRAY(1 TO x) commands;

doInit; new(commands);

INIT commands # must be kept in same order as def's
   ("RESTORE    fileName",
    "ENTER      name <tabs> adr <tabs> phone",
    "REMOVE     name",
    "NAME       name",
    "ADR        adr",
    "PHONE      phone",
    "CHANGE     name <tabs> name <tabs> adr <tabs> phone",
    "SAVE       fileName",
    "FLUSH",
    "EXIT");

DO CASE cmdMatch(commands,"*",useKeyWord,s) OFB
        [restoreCase]   doRestore(s);
        [enterCase]     doEnter(s);
        [removeCase]    doRemove(s);
        [nameCase]      doLookUp(byName,s);
        [adrCase]       doLookUp(byAdr,s);
        [phoneCase]     doLookUp(byPhone,s);
        [changeCase]    doChange(s);
        [saveCase]      doSave(s);
        [flushCase]     doFlush;
        [exitCase]      exit;
        END;
END;

END "dirMod"
```

Example 15.2-4. A Module That Uses HSHMOD (end)

15.3. Prefix Classes and Explicit Class Specifications

Example 15.2-4 contains an example of a prefix class declaration; the class "hashedRecord" is a prefix class of the class "person". This means that the class "person" has all the fields of hashRecord as well as the fields declared in person's own declaration. The general form of a prefixed class declaration is:

```
    CLASS(<prefix class>) <class name> (<additional fields>)
```

Sometimes the programmer knows that a pointer variable is of a prefixed class even though it is declared to be of the prefixed class's prefix class. If you need to refer to a field of the prefixed class that does not belong to the prefix class, you may use the form:

```
            <pointer variable>:<prefixed class name>
              .<prefixed class field>
```

An example of such explicit class specification appears in the procedure "processRoomCmd" of Example 15.5-2. The programmer knows that the pointer "player.where" is of the class "roomCls", even though it is declared to be of roomCls's prefix class objectCls. Therefore, to refer to player.where's field "roomList", the form "player.where:roomCls.roomList" is used.

Explicit class specifications may be used to specify any class for a pointer variable, even a class that is incompatible with the actual class of the pointer. You must therefore be very careful in using explicit class specifications; using a field that does not actually exist in the record pointed to by a pointer can lead to bugs that are extremely difficult to track.

15.4. Explanation of Example 15.2-4

The three separate data sections created by the calls to "new" in the procedure doInit of Example 15.2-4 are all associated with the same control section. Whenever a procedure of HSHMOD is called through one of the pointers byName, byAdr, and byPhone, the same procedure code is executed, but it operates on the data of only the appropriate data section. See Example 15.4-1, which shows separate copies of the HSHMOD outer variable "hashList" maintained for each instance of HSHMOD. Separate copies of the variable "numberOfHashLists" are also maintained but not shown.

When "byAdr.hashEnter" is called, only the copy of hashList in the data section pointed to by byAdr is modified by the hashEnter procedure in HSHMOD; the copies of hashList for byName and byPhone are not altered.

```
                Control sections
+--------+    +--------+ +--------+     +--------+
|runtime |    |runtime | |        |     |        |
|module  |    |module  | |DIRMOD  |     |HSHMOD  |
|#1      |    |#n      | |control |     |control |
|control | ...|control | |section |     |section |
|section |    |section | |        |     |        |
+--------+    +--------+ +--------+     +--------+
     ^             ^          ^              ^
     |             |          |              |
     |      +------|----------|--------------+-------+
     |      |      |          |              |       |
     |      |      |          |      +>+-----+----+  |
     |      |      |          |      | |HSHMOD    |  |
     |      |      |          |      | |data      |  |
     |      |      |     +----+----+ | |section   |  |
     |Data sections|     |DIRMOD    | | |(nonbound)|  |
+----+---+  | +----+---+ |data      | | +----------+  |
|runtime |  | |runtime | |section   | | |hashList >+----+
|module  |  | |module  | |(bound)   | | +----------+  | |
|#1      |  | |#n      | +---------+ |               | |
|data    | ...|data    | |byName >-+-+       +-------+ |
|section |  | |section | +---------+         |         |
|(bound) |  | |(bound) | |byAdr  >-+-->+-----+----+    |
+--------+  | +--------+ +---------+   |HSHMOD    |    |
      +-----+-----+<-----+< byPhone|   |data      |    |
      |HSHMOD     |      +---------+   |section   |    |
      |data       |                    |(nonbound)|    |
      |section    |                    +----------+    |
      |(nonbound) |  +------------+<---+< hashList|    |
      +-----------+  |hashList for|    +----------+    |
+-----+< hashList |  |byAdr       |                    |
|     +-----------+  +------------+                    |
|                                                      |
+->+--------------------+      +-------------------+<--+
   |hashList for byPhone|      |hashList for byName|
   +--------------------+      +-------------------+
```

Example 15.4-1. The Three Separate HSHMOD Data Sections

15.5. Dungeon Game Example

Since hashList is an outer variable, not an interface variable, the forms "byName.hashList", "byAdr.hashList", and "byPhone.hashList" would not be valid in the module of Example 15.2-4. The variable "hashList" is hidden from modules other than HSHMOD. However, when data interface fields of modules are declared, they may be accessed by means of pointers, just like procedure interface fields. Data interface fields of data sections resemble fields of records in both syntax and function.

The program of Examples 15.5-1, 15.5-2, and 15.5-3 makes extensive use of explicit pointers to data sections to reference both procedure and data fields. The program is a simplified "dungeon" game. In such a game, a player plays against the computer by moving from room to room in the dungeon, accumulating treasure and fighting monsters. The example here is too small to have either treasure or monsters, although some provision has been made for their implementation.

Every object (player, room, or other item) in the game is implemented as a module; each module is of a prefixed class of the common prefix class "objectCls". Because all of the objects share common fields, many operations may be performed on any of the objects in the dungeon without reference to its particular type or form; this is the essence of an "object-oriented" programming style.

The module DNGN of Example 15.5-2 provides a number of "global" interface fields. Most of these are utility procedures called by the objects in the dungeon to do things in a standard way; some globally accessible data are also provided. DNGN is also the top-level module, i.e., the one invoked by the user from MAINEX. DNGN calls the module GTPL (in Example 15.5-3) to initialize the list of creatures in the dungeon. Each creature is given an initial location, which has the effect of initializing the rooms in the dungeon. GTPL functions therefore as a sort of "configuration" module; it is intended to be changed if a more sophisticated dungeon is created. In this example, GTPL allocates only one creature, the player.

The game is designed so that each room module allocates the objects in the room. However, rooms accessible from a given room are not allocated until they are first entered (or otherwise needed). Rooms are allocated by calls to "bind" rather than calls to "new", since it is intended that there be only one instance of each room, and that each room be globally accessible. In the example shown, creatures and other objects are also all allocated by calls to "bind", although it is easy to imagine an object or creature of which one would want many instances (gold coins or goblins, perhaps). It would be more appropriate to allocate such objects by means of "new" rather than "bind". The decision to use "new" or "bind" is left to the code of the room in which the object is to appear, although "bindThings", which is the default version of the object allocator provided by the module DNGN, uses "bind" rather than "new".

Commands are processed depending on context. First, the current room is allowed to examine a command to determine whether it is applicable in the room. Most rooms call the DNGN procedure "processRoomCmd", which checks to see whether the given command matches any of the commands to move the player to an adjacent room. If not, the objects in the room and the objects carried by the player are given an opportunity to process the given command. If no object processes the command, the command processor issues the message "I don't understand".

DNGN is intended to be executed only once during a MAINEX session, since it makes no attempt to unbind the modules bound during execution. An attempt could be made to keep track of these modules and dispose them at the end of DNGN's initial procedure.

Example 15.5-1 shows the file "dngn.dcl", which is sourcefiled by every module in the game. The file contains the declarations of all common classes, modules, and macros. Example 15.5-2 shows the module DNGN, and Example 15.5-3 shows the rest of the modules in the dungeon.

```
# Rooms and portable items are both "objects"; each is
# represented by a data section of the class objectCls.

CLASS objectCls (
    BOOLEAN PROCEDURE processCmd (STRING s);
        # Returns true if was able to process the command
        # string.

    STRING announcement,shortAnnouncement;
        # What to say if room is entered or object
        # encountered.
    BOOLEAN encountered;
        # After first encounter, use shortAnnouncement
    POINTER(objectCls) what,next;
        # Used to link objects in the same place
    POINTER(objectCls) where;
        # Back pointer to place where object is
    STRING name;
        # For debugging and getting/dropping objects
    BOOLEAN illuminated;
        # Whether it gives off light
);
```

Example 15.5-1. The Sourcefiled Declaration File "dngn.dcl" (continued)

```
CLASS roomListCls (
    STRING roomName; # Module name of room
    STRING cmdToGetThere; # Command used to get to it
    POINTER(roomCls) roomPtr;
        # Used to avoid re-bind's: if initialized, don't
        # bind using string roomName just to get the
        # pointer to the bound data section
    POINTER(roomListCls) nextRoom;
);

CLASS(objectCls) roomCls ( # Prefixed class for rooms
    POINTER(roomListCls) roomList;
        # Rooms accessible from this room

    PROCEDURE doOnEntry;
        # Something to do upon entering the room
);

CLASS(objectCls) creatureCls ( # Prefixed class for player
                               # (and potential other
                               # creatures)
    PROCEDURE doTurn; # Do one turn for the creature

    PROCEDURE attack (INTEGER howHard); # Attack it

    POINTER(creatureCls) nextCreature;

    BOOLEAN dead; # Should be removed from creatureList
    BOOLEAN realPlayer; # Quit when no real players left
);

CLASS(objectCls) thingCls ( # Prefixed class for portable
    INTEGER weight,value;   # items
);

MODULE dngn ( # Utility procedures and global variables
    PROCEDURE say (STRING s);
    # Write s to logFile, followed by eol

    BOOLEAN PROCEDURE isSameCmd (STRING s,t);
    # True if s and t are the same except for spacing and
    # case
```

Example 15.5-1. The Sourcefiled Declaration File "dngn.dcl" (continued)

```
BOOLEAN PROCEDURE processRoomCmd (STRING s);
# Called by most room modules to process commands in
# a standard way

PROCEDURE enterRoom (POINTER(roomCls) p;
                     OPTIONAL BOOLEAN noPreviousRoom);
# Move to the specified room
# noPreviousRoom is set if player.where is not
# expected to be set, as may be the case when
# initializing

BOOLEAN PROCEDURE processThingCmd
    (STRING s; POINTER(objectCls) p);
# Called by most portable item modules to process
# commands in a standard way

PROCEDURE standardPlayerTurn;
# Standard command processor

BOOLEAN PROCEDURE contains (POINTER(objectCls) p,q);
# Returns true if p's .next list contains q

PROCEDURE takeAwayFrom (POINTER(objectCls) p,q);
# q is currently in p's .next list; remove it

PROCEDURE addTo (POINTER(objectCls) p,q);
# put q into p.next's .next list

BOOLEAN PROCEDURE thereIsLight;
# True if something in the surroundings is giving off
# enough light to see by.

PROCEDURE doLook (OPTIONAL BOOLEAN longAnnoucements);
# Describe the surroundings.

PROCEDURE doShow;
# List what the player is holding.
```

Example 15.5-1. The Sourcefiled Declaration File "dngn.dcl" (continued)

```
    PROCEDURE setUpRoom (POINTER(roomCls) p;
                         REPEATABLE STRING otherRoom);
    # Set up this room's roomList.  Each otherRoom string
    # is of the form "<module name> <command>", where the
    # command is the command to get to the room named.

    PROCEDURE bindThings (POINTER(roomCls) p;
                          REPEATABLE STRING thingName);
    # The specified things are placed in the room.  The
    # thing names are the names of the appropriate
    # modules.

    POINTER(creatureCls) creatureList,player;
    # player is the current creature on the creature list
    # (the creature whose turn it is).  In this version,
    # player is the only creature on the list.
    # player.what is what the player is carrying;
    # player.where is the current room
);

MODULE gtpl (
    PROCEDURE getPlayers;
        # Initialize the creature list
);

# Assign values to standard fields (of this module):
DEFINE setUp (n,a,s) = [
    BEGIN
    name := n; announcement := a;
    shortAnnouncement := s END];
```

Example 15.5-1. The Sourcefiled Declaration File "dngn.dcl" (end)

```
BEGIN "dngn"

SOURCEFILE "dngn.dcl";
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (continued)

```
PROCEDURE say (STRING s);
BEGIN
IF NOT player.realPlayer THEN RETURN;
    # Don't say anything if phony player
write(logFile,s,eol);
END;



BOOLEAN PROCEDURE isSameCmd (STRING s,t);
BEGIN
WHILE s AND t DOB
    scan(s," " & tab,proceed!omit);
    scan(t," " & tab,proceed!omit);
    IF scan(s," " & tab,upperCase) NEQ
       scan(t," " & tab,upperCase) THEN RETURN(FALSE) END;
RETURN(NOT (s OR t));
END;



BOOLEAN PROCEDURE processRoomCmd (STRING s);
BEGIN
POINTER(roomListCls) p;
POINTER(thingCls) q;
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (continued)

```
# First: is the command applicable to the current room?
p := player.where:roomCls.roomList;
WHILE p DOB
    IF isSameCmd(s,p.cmdToGetThere) THENB
        IF NOT p.roomPtr THEN p.roomPtr :=
            bind(p.roomName); # Bind if not already bound
        enterRoom(p.roomPtr); RETURN(TRUE) END;
    p := p.nextRoom END;
# Now is it applicable to any object in the room, or to
# something being carried?
q := player.where.what; # list of things in room
WHILE q DOB
    IF q.processCmd(s) THEN RETURN(TRUE); q := q.next END;
q := player.what; # list of things carried
WHILE q DOB
    IF q.processCmd(s) THEN RETURN(TRUE); q := q.next END;
# Command not applicable to anything around:
RETURN(FALSE);
END;



PROCEDURE announce (POINTER(objectCls) p;
                    OPTIONAL BOOLEAN longAnnouncement);
IF p.encountered AND NOT longAnnouncement THEN
    say(p.shortAnnouncement)
EB  say(p.announcement);
    p.encountered := TRUE END;
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (continued)

```
BOOLEAN PROCEDURE processThingCmd
    (STRING s; POINTER(objectCls) p);
# Standard things to do are "get" ("take") and "drop"
BEGIN
IF isSameCmd(s,"GET " & p.name)
    OR isSameCmd(s,"TAKE " & p.name) THENB
    IF p.where = player THEN
        say("You already have the " & p.name & ".")
    EB  takeAwayFrom(p.where,p); addTo(player,p);
        say("Gotten.") END;
    RETURN(TRUE) END
EF isSameCmd(s,"DROP " & p.name) THENB
    IF p.where NEQ player THEN
        say("You don't have the " & p.name & ".")
    EB  takeAwayFrom(player,p); addTo(player.where,p);
        say("Dropped.") END;
    RETURN(TRUE) END;
RETURN(FALSE);
END;



BOOLEAN PROCEDURE contains (POINTER(objectCls) p,q);
BEGIN
p := p.what;
WHILE p AND p NEQ q DO p := p.next;
RETURN(p NEQ NULLPOINTER);
END;



PROCEDURE takeAwayFrom (POINTER(objectCls) p,q);
# remove q from p.what list
BEGIN
POINTER(objectCls) pp;
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (continued)

```
IF (pp := p.what) = q THENB
    p.what := p.what.next; q.where := NULLPOINTER;
    RETURN END;
WHILE pp AND pp.next NEQ q DO pp := pp.next;
IF NOT pp THEN
    errMsg(q.name & " not found in",p.name,fatal);
pp.next := pp.next.next; q.where := NULLPOINTER;
END;



PROCEDURE addTo (POINTER(objectCls) p,q);
# put q into p's .what list
BEGIN
q.where := p; q.next := p.what; p.what := q;
END;



BOOLEAN PROCEDURE thereIsLight;
BEGIN
POINTER(objectCls) p;

# Is there light from the room?
IF player.where.illuminated THEN RETURN(TRUE);
# Is there light from something in the room?
p := player.where.what;
WHILE p DOB
    IF p.illuminated THEN RETURN(TRUE); p := p.next END;
# Is there light from something the player is holding?
p := player.what;
WHILE p DOB
    IF p.illuminated THEN RETURN(TRUE); p := p.next END;
RETURN(FALSE);
END;



PROCEDURE doLook (OPTIONAL BOOLEAN longAnnouncements);
BEGIN
BOOLEAN wroteExtraLine;
POINTER(objectCls) p;
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (continued)

```
IF NOT thereIsLight THENB say("It is dark."); RETURN END;
say(""); announce(player.where,longAnnouncements);
wroteExtraLine := FALSE; # If no announcement, don't add
p := player.where.what;  # an extra line
WHILE p DOB
    IF p.announcement THENB
        IF NOT wroteExtraLine THEN say("");
        wroteExtraLine := TRUE;
        announce(p,longAnnouncements) END;
    p := p.next END;
END;



PROCEDURE enterRoom (POINTER(roomCls) p;
                     OPTIONAL BOOLEAN noPreviousRoom);
BEGIN
IF NOT noPreviousRoom THEN
    takeAwayFrom(player.where,player);
addTo(p,player); doLook; p.doOnEntry;
END;



PROCEDURE setUpRoom (POINTER(roomCls) p;
                     REPEATABLE STRING otherRoom);
BEGIN
POINTER(roomListCls) q;

q := new(roomListCls);
q.roomName := scan(otherRoom," " & tab);
scan(otherRoom," " & tab,proceed!omit);
q.cmdToGetThere := otherRoom;
q.nextRoom := p.roomList; p.roomList := q;
END;



PROCEDURE bindThings (POINTER(roomCls) p;
                      REPEATABLE STRING thingName);
BEGIN
POINTER(thingCls) q;
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (continued)

```
# Perhaps should use "new" instead of "bind" if there
# might be more than one instance of the same object
# in the dungeon.
addTo(p,q := bind(thingName));
END;



PROCEDURE doShow;
BEGIN
POINTER(objectCls) p;

say("You are holding:");
p := player.what;
WHILE p DOB say(p.name); p := p.next END;
END;



PROCEDURE standardPlayerTurn;
BEGIN
STRING s;

write(logFile,">> "); read(cmdFile,s);
IF isSameCmd(s,"QUIT") THENB
     player.dead := TRUE; RETURN END
EF isSameCmd(s,"LOOK") THEN doLook(TRUE)
EF isSameCmd(s,"SHOW") THEN doShow
EF isSameCmd(s,"HELP") THEN
    say("
You are in a dungeon game.  Commands like NORTH (to go
to the next room to the north) or GET KEY (if there is
a key around to get) will work.  QUIT gets you out.  LOOK
gives a complete description of your surroundings. SHOW
gives a list of what you are carrying.

You must find out more by experimentation.")
EF s AND NOT player.where.processCmd(s) THEN
    say("I don't understand.");
IF player.dead THEN
    say(eol & "You died.  So sorry.");
END;
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (continued)

```
BOOLEAN PROCEDURE getPlayerCommands;
# Return false when no realPlayer creatures left
BEGIN
BOOLEAN realOneSeen;

player := creatureList; realOneSeen := FALSE;
WHILE player DOB
    realOneSeen := realOneSeen OR player.realPlayer;
    player.doTurn;
    IF player.dead THENB
        IF player = creatureList THEN
            creatureList := creatureList.nextCreature
        EL player := player.nextCreature END;
    player := player.nextCreature END;
RETURN(realOneSeen);
END;



INITIAL PROCEDURE;
BEGIN
getPlayers;
DO UNTIL NOT getPlayerCommands;
END;

END "dngn"
```

Example 15.5-2. The Dungeon Top-Level Module DNGN (end)

```
BEGIN "gtpl"

# Get player list
# This module may be changed to reconfigure the game;
# could, for example, read a list of players from a file

SOURCEFILE "dngn.dcl";
```

Example 15.5-3. The Modules in the Dungeon Other Than DNGN (continued)

```
PROCEDURE getPlayers;
creatureList := bind("PLAYER"); # Only one player

END "gtpl"
<page mark>
BEGIN "player"

# This module represents the player.  It is mostly of
# interest for its data fields rather than its procedure
# fields.

SOURCEFILE "dngn.dcl";
MODULE(creatureCls) player;

BOOLEAN PROCEDURE processCmd (STRING s);
RETURN(FALSE); # No commands operate on the player



PROCEDURE doTurn;
standardPlayerTurn;



PROCEDURE attack (INTEGER howHard); # Attack it
player.dead := TRUE; # Not prepared for attack



INITIAL PROCEDURE;
BEGIN
name := "PLAYER"; realPlayer := TRUE;
dngn.player := thisDataSection;
enterRoom(bind("START"),TRUE); # Starting room
END;

END "player"
<page mark>
BEGIN "start"
```

Example 15.5-3. The Modules in the Dungeon Other Than DNGN (continued)

```
# This is the starting room.
SOURCEFILE "dngn.dcl";
MODULE(roomCls) start;

BOOLEAN trapDoorOpen;
POINTER(thingCls) key;

DEFINE basicAnnouncement =
"You are in the waiting room of a railway station.  There
does not seem to be anyone behind the ticket counter.  It
is raining heavily outside.
";

BOOLEAN PROCEDURE processCmd (STRING s);
BEGIN
IF isSameCmd(s,"UNLOCK TRAPDOOR") OR
    isSameCmd(s,"UNLOCK") OR
    isSameCmd(s,"OPEN TRAPDOOR") THENB
    IF contains(player,key) THENB
        IF trapdoorOpen THEN
            say("The trapdoor is already open.")
        EB  trapdoorOpen := TRUE;
            say("The trapdoor is now open.");
            announcement := basicAnnouncement & "
There is an open trapdoor in the floor.";
            shortAnnouncement :=
"You are in the railway station waiting room.  The
trapdoor is unlocked." END END
    EL  say("You do not have the key.");
    RETURN(TRUE) END;
IF isSameCmd(s,"DOWN") AND NOT trapdoorOpen THENB
    say("You cannot go down; the trapdoor is locked.");
    RETURN(TRUE) END;
RETURN(processRoomCmd(s));
END;



PROCEDURE doOnEntry;;
    # Nothing to do (note double semicolon; the procedure
    # body is an Empty Statement).
```

Example 15.5-3. The Modules in the Dungeon Other Than DNGN (continued)

```
INITIAL PROCEDURE;
BEGIN
setUp("station",
basicAnnouncement & "
There is a trapdoor in the floor.  It is locked.",
"You are in the railway station waiting room.  The
trapdoor is locked.");
setUpRoom(thisDataSection,"PLTFRM OUT","BSMNT DOWN");
bindThings(thisDataSection,"LAMP","KEY");
illuminated := TRUE; # Light comes in from the windows
key := bind("KEY");
END;

END "start"
<page mark>
BEGIN "lamp"

SOURCEFILE "dngn.dcl";
MODULE(thingCls) lamp;

BOOLEAN PROCEDURE processCmd (STRING s);
BEGIN
IF isSameCmd(s,"LIGHT LAMP") THENB
    say("The lamp is now lit.");
    illuminated := TRUE; RETURN(TRUE) END;
IF isSameCmd(s,"EXTINGUISH LAMP") THENB
    say("The lamp is now out.");
    illuminated := FALSE; RETURN(TRUE) END;
RETURN(processThingCmd(s,thisDataSection));
END;



INITIAL PROCEDURE;
setUp("lamp",
"There is a shiny brass lamp here.",
"There is a lamp here.");

END "lamp"
<page mark>
BEGIN "key"
```

Example 15.5-3. The Modules in the Dungeon Other Than DNGN (continued)

```
SOURCEFILE "dngn.dcl";
MODULE(thingCls) key;

BOOLEAN PROCEDURE processCmd (STRING s);
RETURN(processThingCmd(s,thisDataSection));

INITIAL PROCEDURE;
setUp("key",
"There is a large old-fashioned key here.",
"There is a key here.");

END "key"
<page mark>
BEGIN "pltfrm"

# The railway station platform

SOURCEFILE "dngn.dcl";
MODULE(roomCls) pltfrm;

BOOLEAN PROCEDURE processCmd (STRING s);
RETURN(processRoomCmd(s));



PROCEDURE doOnEntry;
BEGIN
say(eol &
    "It is raining so hard out here that you drown!");
player.dead := TRUE;
END;
```

Example 15.5-3. The Modules in the Dungeon Other Than DNGN (continued)

```
INITIAL PROCEDURE;
BEGIN
setUp("platform",
"This is the railway platform.  No train is in sight.  It
is pouring rain.  There is no place to go but back inside
the station.",
"This is the railway platform.  It is pouring rain.  There
is no place to go but back inside the station.");
illuminated := TRUE;
END;

END "pltfrm"
<page mark>
BEGIN "bsmnt"

# The basement beneath the railway station waiting room

SOURCEFILE "dngn.dcl";
MODULE(roomCls) bsmnt;

BOOLEAN PROCEDURE processCmd (STRING s);
RETURN(processRoomCmd(s));



PROCEDURE doOnEntry;;



INITIAL PROCEDURE;
BEGIN
setUp("basement",
"You are in a small cobwebby room.  There is a flight of
stairs that leads up to a square of light.",
"You are in a small cobwebby room.  There is a flight of
stairs that leads up to a square of light.");
setUpRoom(thisDataSection,"START UP");
END;

END "bsmnt"
```

Example 15.5-3. The Modules in the Dungeon Other Than DNGN (end)

A sample execution of the dungeon game with the "SWAPINFO" subcommand is shown in Example 15.5-4.

```
*dngn,<eol>
Enter subcommands (? for help).
>swapinfo<eol>
><eol>
Swapping in DNGN
DNGN swapped in
Swapping in GTPL
GTPL swapped in
Swapping in PLAYER
PLAYER swapped in
Swapping in START
START swapped in
Swapping in LAMP
LAMP swapped in
Swapping in KEY
KEY swapped in

You are in the waiting room of a railway station.  There
does not seem to be anyone behind the ticket counter.  It
is raining heavily outside.

There is a trapdoor in the floor.  It is locked.
```

Example 15.5-4. An Execution of DNGN with the "SWAPINFO" MAINEX Subcommand (continued)

```
There is a large old-fashioned key here.
There is a shiny brass lamp here.
>> get key<eol>
Gotten.
>> get lamb<eol>
I don't understand.
>> get lamp<eol>
Gotten.
>> light lamp<eol>
The lamp is now lit.
>> unlock trapdoor<eol>
The trapdoor is now open.
>> down<eol>
Swapping in BSMNT
BSMNT swapped in

You are in a small cobwebby room.  There is a flight of
stairs that leads up to a square of light.
>> up<eol>

You are in the railway station waiting room.  The
trapdoor is unlocked.
>> out<eol>
Swapping in PLTFRM
PLTFRM swapped in

This is the railway platform.  No train is in sight.  It
is pouring rain.  There is no place to go but back inside
the station.

It is raining so hard out here that you drown!

You died.  So sorry.
*
```

Example 15.5-4. An Execution of DNGN with the "SWAPINFO" MAINEX Subcommand (end)

15.6. String Forms of "bind" and "new"

As shown in Examples 15.5-2 and 15.5-3, "bind" may take a string argument instead of a module identifier. The same is true of "new". This permits a module (module A) to bind another module (module B) even though B is not declared in A. For example, the module DNGN binds each room as the player enters it, using the string module name of the room as supplied by the previous room. It would be inconvenient to have to include a separate module declaration for every possible room that might be bound by DNGN.

Unlike the module identifier forms, the string forms of "bind" and "new" do not perform any interface checking. Inconsistent interfaces can lead to bugs that are very difficult to track. When you update the interface of a module, be sure to recompile all modules that use that module.

15.7. thisDataSection and Unclassified Pointers

The dungeon game example makes extensive use of the system procedure "thisDataSection". thisDataSection returns a pointer to the current data section. thisDataSection is an example of an unclassified pointer; its value may be assigned to any pointer variable or passed as any pointer parameter. The declaration of thisDataSection is shown in Figure 15.7-1.

```
POINTER PROCEDURE thisDataSection;
```

Figure 15.7-1. Declaration of thisDataSection

Unclassified pointer variables may also exist; a sample declaration is shown in Example 15.7-2. Such variables may be assigned to or from any other pointer. Just as with pointers to prefixed classes, care must be made not to access fields that do not actually exist in a given record.

```
POINTER p; # No parenthesized class name is used
```

Example 15.7-2. An Unclassified Pointer Variable

15.8. Exercises

Exercise 15-1.

Any number of enhancements to the dungeon game program can be imagined. Among them are:

1. Keep track of bound data sections, disposing them all at the end of DNGN's initial procedure.
2. Implement some creatures other than the player.
3. Devise a scoring mechanism and maintain a score for the player.
4. Create more rooms and more treasures. Some treasures may perform their functions only in certain rooms or under certain circumstances.
5. Implement global commands (which take effect in any room).
6. You may want to maintain a list of dynamically created variables or counters that are incremented on each turn. A module can call a utility procedure to create a new variable or to look up an existing variable by name.
7. Implement more sophisticated command parsing; e.g., "it" might be understood to refer to the last object referenced.
8. How about allowing a train to stop at the station platform and take the player somewhere?

16. More on Modules

This chapter describes module search rules. An example of a module that may be invoked either from MAINEX or from another module is shown. The MAINSAIL display modules are discussed.

16.1. Toy Editor Example

This chapter contains a large sample module TOYED (in Example 16.4-1). TOYED uses most of the concepts introduced some far, plus a few new ones.

By this point in the tutorial, you should be able to understand most of the material in the "MAINSAIL Language Manual" without difficulty. Therefore, from this point on, system procedures and other language constructs are not always explained when they are first encountered in examples; you should look them up in the manual if they are unfamiliar.

16.2. Module Search Rules

Whenever a module's control section is established because the module is being bound or newed, the following default search is used to find the control section:

1. Module-to-module associations are searched. There are several ways to make a module-to-module association. It can be made from a MAINSAIL module by calling the procedure "setModName" (see the "MAINSAIL Language Manual") or with the MAINEX "SETMODULE" subcommand (see the "MAINSAIL Utilities User's Guide"). A sample use of the procedure setModName is shown in Example 16.4-1.

2. The exeList (module-to-file associations) is searched. The exeList entries are made from a MAINSAIL module by calling the procedure "setFileName" (see the "MAINSAIL Language Manual") or with the MAINEX "EXEFILE" or "EXELIB" subcommand (see the "MAINSAIL Utilities User's Guide"). If an entry on the exeList is found, the search terminates, and the module contained in the specified file or objmod library is executed.

3. By default, objmod libraries opened for execution are searched (module libraries are further described in Chapter 20). The MAINSAIL system objmod library is initially open. User libraries can be opened from a MAINSAIL module by calling the procedure "openLibrary" (see the "MAINSAIL Language Manual") or with the MAINEX "OPENEXELIB" subcommand (see the "MAINSAIL Utilities User's

Guide"). Open objmod libraries are searched in order of most recently opened to least recently opened. If the module is found in an open library, the search terminates, and the module is executed. If more than one module with the same name exists, the first occurrence found is the one that is selected.

4. The foreign module table is searched. This table is a list of modules that have been declared to be "foreign" (see Section 20.4). A foreign module is one that has been written in a language other than MAINSAIL. MAINSAIL supports invocation of such modules through the Foreign Language Interface (see the "MAINSAIL Compiler User's Guide" and the appropriate operating-system-dependent user's guide). If a foreign module with the proper name is found, the search terminates, and the module is executed.

5. If all of the above searches fail, MAINSAIL forms a file name from the module name (see Table 16.2-1). If the resulting file exists, it is assumed to contain a MAINSAIL object module. This module is executed.

```
An objmod file name for a particular system is made from a
module name by prefixing:

    <1st 3 characters of $systemNameAbbreviation>-obj:

to the module name.  The resulting name is typically
transformed further by a "SEARCHPATH" entry (see the
"MAINSAIL Utilities User's Guide"):

               SEARCHPATH *-obj:* *2-*1.obj

For example, a module BAR compiled for an M68000 UNIX
system (where $systemNameAbbreviation is "um68") is
compiled into an objmod file named "um6-obj:bar".  The
standard searchpath would map "um6-obj:bar" into
"bar-um6.obj".
```

Table 16.2-1. Default Objmod File Names

If the "EXEFILE" MAINEX subcommand with no arguments is in effect, the file search is made before the open library and foreign module searches.

16.3. $useProgramInterface

The boolean macro $useProgramInterface may be used to determine whether a module should be executed in an interactive ("program-like" or "independent") fashion, or in a "dependent" fashion. $useProgramInterface must be called only in a module's initial procedure, before the module makes any procedure calls; otherwise, its value is unpredictable. $useProgramInterface is true if and only if the initial procedure is being invoked for one of the following reasons:

- An interface procedure is being called.
- "bind(m,b)" or "new(m,b)" was called, where the optional bits parameter b has the $programInterface bit set.

Normal uses of "bind" and "new" within the MAINSAIL runtime system do not set the $programInterface bit. For example, when a module is invoked from MAINEX, the bit is not set, so that $useProgramInterface is false if queried by the module's initial procedure.

The module of Example 16.4-1 is set up so that it may be invoked in a "dependent" fashion from another module. An invoking module, USEED, is shown in Example 16.4-2. USEED may be invoked from MAINEX, in which case it in turn calls TOYED, or it may be invoked from TOYED by means of TOYED's "M" command.

16.4. The MAINSAIL Display Modules

The module of Example 16.4-1 uses the MAINSAIL "display modules". Most display terminals accept special character sequences as display commands, e.g., to position the cursor or insert or delete lines on the screen. The display modules permit a program to be independent of a particular terminal type because the display functions are implemented as terminal-independent display module interface procedures rather than as terminal-dependent character sequences. XIDAK supports a variety of display modules for different terminals; a list of the supported terminals may be found in the "MAINEDIT User's Guide". The MAINSAIL display modules are included as a standard part of every MAINSAIL system.

The interface declaration of the display modules is contained in the standard system source library. You may include the interface declaration in your program by means of the following code:

```
REDEFINE $scanName = "dpyHdr"; SOURCEFILE "(system library)";
```

Most of the display module interface procedures have names that are descriptive of what they do; e.g., "setCursorOnScreen" positions the cursor, and "clearScreen" clears the display.

XIDAK reserves the right to change the display module interface without notice. However, the interface is not expected to change a great deal. If you wish to use the display modules in your own program, you should contact XIDAK for information.

```
BEGIN "toyEd"

# ToyEd is a "toy editor" that provides a full-screen
# editing capability with commands such as delete
# character and line, insert character, and text search.
# It is meant to demonstrate many of the features of
# MAINSAIL rather than to be a production editor.

# The file to be edited is read into a string array, one
# line per element.  Tabs are converted to spaces on
# input.  Delete-line and insert-line move all elements
# below the affected line up or down by one element.
# Delete-character and insert-character build a new
# string for the current line.

# The screen update employs the display modules, which
# are part of MAINEDIT.  A separate display module is
# provided for each terminal.  The proper display module
# is dynamically brought into memory when toyEd
# starts execution.  Execution starts in the initial
# procedure, which dynamically determines which display
# module to employ.  It then prompts for the input file,
# and calls the procedure executeCommands to process the
# user commands.  User keystrokes are processed
# immediately upon entry, so that the screen is updated
# as soon as a complete command is entered; the
# keystrokes for the commands are not echoed.

# The M command allows the user to invoke any MAINSAIL
# module during the edit session.  Control is returned to
# the editor when the invoked module finishes.

#                    ToyEd Command Summary
```

Example 16.4-1. Toy Editor Program (continued)

```
# Note: the MAINSAIL display modules return special codes
# for arrow keys if they exist on the terminal.  Every
# terminal also has a special "enterCommandMode" key
# that is used to enter command mode from overstrike or
# insert mode.

# Command Mode
# ------------
# C           Copy line (push it onto delete stack)
# D           Delete line (and push it onto delete stack)
# F           Finish (save file and exit)
# Gn          Go to line n
# I           enter Insert mode
# K           delete (Kill) character
# Ms          execute Module s (dispose-bind-unbind)
# N           refresh screen (New screen)
# O           enter Overstrike mode
# Q           Quit without saving file
# R           Recover line (pop from delete stack)
# Ts<CR>      Text search right and all lines down for s
# W           scroll up one line (Window)
# -W          scroll down one line (Window)
# <, <del>, left arrow
#             move left one column
# >, right arrow
#             move right one column
# ^, up arrow
#             move up one row
# \, <lf>, down arrow
#             move down one row
# <cr>        move to first column of next line

# Overstrike Mode
# ---------------
# <lf>        move down 1 row, enter command mode
# <cr>        move to first column of next line
# <del>, left arrow
#             move left 1 column
# right arrow
#             move right one column
# up arrow    move up one row
# down arrow move down one row
```

Example 16.4-1. Toy Editor Program (continued)

```
# Insert Mode
# -----------
# <lf>        move down 1 row, enter command mode
# <cr>        break line, move cursor to start of new line
# <del>       delete character to left
# left arrow move left 1 column
# right arrow
#             move right one column
# up arrow    move up one row
# down arrow move down one row

MODULE toyEd (
    # The data structures are accessible from other
    # modules so that a module may set up the data
    # structure, then invoke TOYED to display and edit it

    PROCEDURE executeCommands
        (OPTIONAL STRING commands;
         OPTIONAL BOOLEAN returnWhenExhausted);

    INTEGER PROCEDURE setLastLine (INTEGER newLastLine);

    STRING PROCEDURE msg
        (OPTIONAL STRING promptString;
         OPTIONAL BOOLEAN justPrompt);

    PROCEDURE finish (OPTIONAL BOOLEAN dontWriteFile);

    INTEGER
        curLine,            # line index for current line
        firstLineOnScreen, # index for 1st line on screen
        curCol;             # current column (0-origin)

    STRING ARRAY(0 TO *)
        line;                # major data structure
);

REDEFINE $scanName = "dpyHdr"; # Pick up standard display
SOURCEFILE "(system library)"; # module stuff

MODULE(dpyCls) dpy;
```

Example 16.4-1. Toy Editor Program (continued)

```
BOOLEAN
    executingAnotherModule; # M command has been invoked

INTEGER
    lastLine;           # index for last line

DEFINE
    firstRowOfScreen = 0,  # rows, columns on
    firstColOfScreen = 0,  # screen are 0-origin
    lineInc          = 500,    # arbitrary
    error(m)         = [BEGIN msg(m,TRUE); ringBell END],
    setCursor(r,c)   = [dpy.setCursorOnScreen(
                          (r) - firstLineOnScreen + 1,c)];

# Set up for ASCII or EBCDIC character set ($charSet is
# a predefined constant that may be examined by any
# program)
IFC $charSet = $ascii THENC
DEFINE
    cr                  =        13,
    lf                  =        10,
    del                 =        127;
ELSEC IFC $charSet = $ebcdic THENC
DEFINE
    cr                  =        13,
    lf                  =        37,
    del                 =        7;
ELSEC
MESSAGE "Unknown character set";
ENDC ENDC

CLASS stack (STRING line; POINTER(stack) link);

POINTER(stack)
    deleteStack;        # copied and deleted lines

FORWARD PROCEDURE scrollUp;
FORWARD PROCEDURE scrollDown;
FORWARD PROCEDURE insertLine;
FORWARD PROCEDURE refreshScreen;
```

Example 16.4-1. Toy Editor Program (continued)

```
INTEGER PROCEDURE setLastLine (INTEGER newLastLine);
# set the value of lastLine, increasing the line array in
# size if necessary
BEGIN
IF newLastLine > line.ubl
    THEN newUpperBound(line,newLastLine + lineInc);
RETURN(lastLine := newLastLine);
END;



PROCEDURE moveLeft;
IF curCol THEN setCursor(curLine,curCol .- 1);



PROCEDURE moveRight;
IF curCol < lastColOfScreen THEN
    setCursor(curLine,curCol .+ 1);



PROCEDURE moveUp;
IF curLine > 0 THENB
    IF curLine = firstLineOnScreen THEN scrollDown;
    setCursor(curLine .- 1,curCol) END;



PROCEDURE moveDown;
BEGIN
IF curLine = firstLineOnScreen + lastRowOfScreen - 1 THEN
    scrollUp;
setCursor(curLine .+ 1,curCol);
setLastLine(lastLine MAX curLine) END;
```

Example 16.4-1. Toy Editor Program (continued)

```
PROCEDURE overStrikeChar (INTEGER char);
BEGIN
STRING s;
IF curCol GEQ lastColOfScreen THEN
    error("Cannot overstrike here")
EB  s := line[curLine];
    WHILE length(s) < curCol DO cWrite(s,' ');
    line[curLine] := s[1 TO curCol] & cvcs(char) &
        s[curCol + 2 TO INF];
    dpy.overStrikeChar(char); curCol .+ 1 END END;



PROCEDURE insertChar (INTEGER char);
BEGIN
STRING s;
IF curCol GEQ lastColOfScreen THEN
    error("Cannot insert here")
EB  s := line[curLine];
    WHILE length(s) < curCol DO cWrite(s,' ');
    line[curLine] := s[1 TO curCol] & cvcs(char) &
        s[curCol + 1 TO INF];
    dpy.insertChar(char); curCol .+ 1 END END;



PROCEDURE deleteChar;
BEGIN
DEFINE s = [line[curLine]];
dpy.deleteChars(1);
s := s[1 TO curCol] & s[curCol + 2 TO INF] END;



PROCEDURE copyLine;
BEGIN
POINTER(stack) p;
p := new(stack); p.link := deleteStack; deleteStack := p;
p.line := line[curLine]; moveDown END;
```

Example 16.4-1. Toy Editor Program (continued)

```
PROCEDURE deleteLine;
BEGIN
INTEGER         i;
POINTER(stack)  p;
p := new(stack); p.link := deleteStack; deleteStack := p;
p.line := line[curLine];
FOR i := curLine UPTO lastLine - 1 DO
    line[i] := line[i + 1];
line[lastLine] := ""; setLastLine(lastLine - 1);
dpy.deleteLines(1);
dpy.setCursorOnScreen(lastRowOfScreen,0);
dpy.clearToEndOfLine;
dpy.overStrikeChars
    (line[firstLineOnScreen + lastRowOfScreen - 1]);
setCursor(curLine,curCol := firstColOfScreen) END;



PROCEDURE breakLine;
BEGIN
STRING s;
s := line[curLine][curCol + 1 TO INF];
line[curLine] := line[curLine][1 TO curCol];
dpy.clearToEndOfLine; moveDown; insertLine;
IF line[curLine] := s THENB
    dpy.overStrikeChars(s);
    setCursor(curLine,curCol) END;
END;



STRING PROCEDURE popLine;
BEGIN
STRING          s;
POINTER(stack)  p;
IF NOT p := deleteStack THEN RETURN("");
deleteStack := p.link; s := p.line; RETURN(s) END;
```

Example 16.4-1. Toy Editor Program (continued)

```
PROCEDURE recoverLine;
BEGIN
insertLine;
dpy.overStrikeChars(line[curLine] := popLine);
setCursor(curLine,curCol := firstColOfScreen) END;



PROCEDURE insertLine;
BEGIN
INTEGER i;
FOR i := setLastLine(lastLine + 1) DOWNTO curLine DO
    line[i] := line[i - 1];
line[curLine] := "";
setCursor(curLine,curCol := firstColOfScreen);
dpy.insertLines(1) END;



PROCEDURE scrollUp;
BEGIN
setLastLine(lastLine MAX
    (firstLineOnScreen + lastRowOfScreen));
dpy.setCursorOnScreen(firstRowOfScreen + 1,0);
dpy.deleteLines(1);
dpy.setCursorOnScreen(lastRowOfScreen,0);
dpy.overStrikeChars
    (line[firstLineOnScreen + lastRowOfScreen]);
curLine .MAX (firstLineOnScreen .+ 1);
setCursor(curLine,curCol) END;



PROCEDURE scrollDown;
BEGIN
IF firstLineOnScreen = 0 THEN RETURN;
dpy.setCursorOnScreen(firstRowOfScreen + 1,0);
dpy.insertLines(1);
dpy.overStrikeChars(line[firstLineOnScreen .- 1]);
curLine .MIN (firstLineOnScreen + lastRowOfScreen - 1);
setCursor(curLine,curCol) END;
```

Example 16.4-1. Toy Editor Program (continued)

```
PROCEDURE goto
    (OPTIONAL INTEGER lineOfBuffer,colOfScreen);
BEGIN
curLine := 0 MAX lineOfBuffer MIN lastLine;
curCol := 0 MAX colOfScreen MIN lastColOfScreen;
IF firstLineOnScreen LEQ curLine <
    firstLineOnScreen + lastRowOfScreen THEN
    setCursor(curLine,curCol)
EB  firstLineOnScreen := curLine;
    setLastLine(lastLine MAX
        (firstLineOnScreen + lastRowOfScreen - 1));
    refreshScreen END;
END;



PROCEDURE refreshScreen;
BEGIN
INTEGER i;
dpy.clearScreen;
FOR i := firstRowOfScreen + 1 UPTO lastRowOfScreen DOB
    dpy.setCursorOnScreen(i,0);
    dpy.overStrikeChars
        (line[firstLineOnScreen + i - 1]) END;
setCursor(curLine,curCol) END;



PROCEDURE search (STRING searchString);
BEGIN
INTEGER     i,t,u;
BITS        b;
STRING      s;
OWN STRING  previousSearchString;

IF searchString THEN previousSearchString := searchString
EF NOT searchString := previousSearchString THEN RETURN;

u := length(searchString);
b := scanSet(cvu(searchString[1 FOR 1]));
```

Example 16.4-1. Toy Editor Program (continued)

```
FOR i := curLine UPTO lastLine DOB "out"
    s := line[i];
    IF i = curLine THEN s := s[curCol + 2 TO INF];
    WHILE length(s) GEQ u DOB
        scan(s,b,uppercase!omit!retain,t);
        IF s AND equ(searchString,s[1 FOR u],upperCase)
            THEN DONE "out"; # Note use of named loop
        cRead(s) END END "out";

scanRel(b);
IF i > lastLine THEN
    error("Did not find """ & searchString & """")
EL goto(i,length(line[i]) - length(s)) END;



PROCEDURE setupFile (OPTIONAL BOOLEAN noFile);
BEGIN
BOOLEAN           newFile;
INTEGER           ch,col;
STRING            r,s;
POINTER(textFile) f;

IF NOT noFile THEN
    DO UNTIL
        open(f,"File to edit: ",prompt!input!errorOk)
        OR newFile := confirm("New file")
        OR confirm("Do you want to exit");

IF f THENB
    setLastLine(-1);
    DOB # expand tabs to spaces on input
        read(f,r); IF NOT $gotValue(f) THEN DONE;
        s := ""; col := 0;
        WHILE r DOB
            IF ch := cread(r) = first(tab) THEN
                DO cWrite(s,' ')
                UNTIL NOT (col .+ 1) MOD 8
            EB  cWrite(s,ch); col .+ 1 END END;
        line[setLastLine(lastLine + 1)] := s END;
    close(f) END
EF NOT (newFile OR noFile) THEN exit;
```

Example 16.4-1. Toy Editor Program (continued)

```
curLine := curCol := 0;
initializeTerminal; refreshScreen END;



PROCEDURE finish (OPTIONAL BOOLEAN dontWriteFile);
BEGIN
INTEGER             i;
POINTER(textFile)   f;
STRING              r,s;

IF NOT dontWriteFile THENB
    f := NULLPOINTER;
    r := s := "Output file (just eol for none): ";
    WHILE s := msg(s) AND NOT
        open(f,s,output!errorOk!alterOK) DO
        s := "Could not open " & s & ". " & r;

    IF f THENB
        WHILE lastLine AND NOT line[lastLine] DO
            setLastLine(lastLine - 1);
        FOR i := 0 UPTO lastLine DO write(f,line[i],eol);
        close(f) END END;

IF executingAnotherModule THEN RETURN;
    # Don't deinitialize terminal if may return to TOYED

dpy.clearScreen;
dpy.setCursorOnScreen
    (firstRowOfScreen,firstColOfScreen);
deInitializeTerminal END;
```

Example 16.4-1. Toy Editor Program (continued)

```
PROCEDURE executeModule (STRING moduleName);
BEGIN
# The executed module may call back into TOYED
executingAnotherModule := TRUE;
dpy.clearScreen;
dpy.setCursorOnScreen
    (firstRowOfScreen,firstColOfScreen);
dispose(moduleName); bind(moduleName); unBind(moduleName);
executingAnotherModule := FALSE;
refreshScreen END;



STRING PROCEDURE msg (OPTIONAL STRING  promptString;
                      OPTIONAL BOOLEAN justPrompt);
BEGIN
INTEGER t,u;
STRING  s;
dpy.setCursorOnScreen
    (firstRowOfScreen,firstColOfScreen);
dpy.clearToEndOfLine;
dpy.overStrikeChars(promptString);
IF NOT justPrompt THENB
    u := length(promptString); s := "";
    DOB t := dpy.dpycRead;
        IF t = dpy.dpyEol THEN DONE;
        IF t NEQ del THENB
            cWrite(s,t); u .+ 1;
            dpy.overStrikeChar(t) END
        EF s THENB
            rcRead(s);
            dpy.setCursorOnScreen
                (firstRowOfScreen,u .- 1);
            dpy.deleteChars(1) END END END;
setCursor(curLine,curCol); RETURN(s) END;
```

Example 16.4-1. Toy Editor Program (continued)

```
PROCEDURE executeCommands
    (OPTIONAL STRING commands;
     OPTIONAL BOOLEAN returnWhenExhausted);
BEGIN
BOOLEAN minus;
INTEGER mode,t;

DEFINE
    commandMode    = 1,
    insertMode     = 2,
    overStrikeMode = 3,

    setMode(m)     = [msg(
                       IF mode := m = commandMode THEN "C"
                       EF mode = insertMode THEN "I"
                       EL "O",TRUE)];

mode := commandMode; minus := FALSE;

DOB "outerLoop"
    IF commands THEN t := cRead(commands)
    EF returnWhenExhausted THENB
        setMode(commandMode); DONE END
    EL t := dpy.dpycRead;

    CASE mode OFB "outerCase"
```

Example 16.4-1. Toy Editor Program (continued)

```
[insertMode]
    CASE t OFB
        [cr]    breakLine;
        [lf]    BEGIN
                moveDown;
                setMode(commandMode) END;
        [del]   IF curCol THENB
                    moveLeft; deleteChar END;
        [dpyLeft]
                moveLeft;
        [dpyRight]
                moveRight;
        [dpyUp] moveUp;
        [dpyDown]
                moveDown;
        [dpyEnterCommandMode]
                setMode(commandMode);
        [ ]     insertChar(t);
        END;

[overStrikeMode]
    CASE t OFB
        [cr]    BEGIN
                moveDown;
                goto(curLine,firstColOfScreen)
                END;
        [lf]    BEGIN moveDown;
                setMode(commandMode) END;
        [del][dpyLeft]
                moveLeft;
        [dpyRight]
                moveRight;
        [dpyUp] moveUp;
        [dpyDown]
                moveDown;
        [dpyEnterCommandMode]
                setMode(commandMode);
        [ ]     overStrikeChar(t);
        END;
```

Example 16.4-1. Toy Editor Program (continued)

```
        [commandMode]
            CASE cvu(t) OFB
                ['C']   copyLine;
                ['D']   deleteLine;
                ['F']   BEGIN finish; DONE END;
                ['G']   goto(cvi(msg("Line: ")));
                ['I']   setMode(insertMode);
                ['K']   deleteChar;
                ['M']   executeModule(msg("Module: "));
                ['N']   refreshScreen;
                ['O']   setMode(overStrikeMode);
                ['Q']   IF equ(msg("Quit (Y or N): "),
                                "Y",upperCase) THENB
                            finish(TRUE); DONE END;
                ['R']   recoverLine;
                ['T']   search(msg("Search string: "));
                ['W']   IF minus THEN scrollDown
                        ELSE scrollUp;
                ['-']   BEGIN minus := TRUE; CONTINUE END;
                ['<'][del][dpyLeft]
                        moveLeft;
                ['>'][dpyRight]
                        moveRight;
                ['^'][dpyUp]
                        moveUp;
                [cr]    BEGIN
                        moveDown;
                        goto(curLine,firstColOfScreen)
                        END;
                ['\'][lf][dpyDown]
                        moveDown;
                [dpyEnterCommandMode] ;
                [   ]   ringBell;
                END;

        END "outerCase";

    minus := FALSE END "outerLoop";

END;
```

Example 16.4-1. Toy Editor Program (continued).

```
INITIAL PROCEDURE;
BEGIN
BOOLEAN useProgramInterface;

useProgramInterface := $useProgramInterface;
    # Must call $useProgramInterface first thing in the
    # initial procedure
ttyWrite("Display module: ");
setModName("dpy",ttyRead);
new(line,0,lineInc);
# Ask for file name and continue only if called
# interactively:
setUpFile(useProgramInterface);
IF NOT useProgramInterface THEN executeCommands END;

END "toyEd"
```

Example 16.4-1. Toy Editor Program (end)

```
BEGIN "useEd"

MODULE toyEd (
    PROCEDURE executeCommands
        (OPTIONAL STRING commands;
         OPTIONAL BOOLEAN returnWhenExhausted);

    INTEGER PROCEDURE setLastLine (INTEGER newLastLine);

    STRING PROCEDURE msg
        (OPTIONAL STRING promptString;
         OPTIONAL BOOLEAN justPrompt);

    PROCEDURE finish (OPTIONAL BOOLEAN dontWriteFile);

    INTEGER
        curLine,           # line index for current line
        firstLineOnScreen, # index for 1st line on screen
        curCol;            # current column (0-origin)
```

Example 16.4-2. A Module That Uses TOYED (continued)

```
    STRING ARRAY(0 TO *)
        line;                   # major data structure
);

INITIAL PROCEDURE;
# Write 21 lines into the TOYED data structure.  Set
# the cursor at the fifth line, fourth column of the
# buffer; put the third line of the buffer at the top
# of the screen.  Then call TOYED with the "N" command,
# which displays the data structure.  Then prompt the
# user for a string and echo it back.  Then execute
# the "<" command, and continue executing TOYED commands
# until TOYED exits ("Q" or "F" command).

# If USEED was invoked by means of the TOYED "M" command,
# it will return to TOYED, and a second "Q" or "F" command
# will be needed to exit from TOYED to MAINEX.
BEGIN
INTEGER i;

setLastLine(20);
FOR i := 0 UPTO 20 DO line[i] := "This is line " & cvs(i);
curLine := 5; firstLineOnScreen := 3; curCol := 4;
executeCommands("N",TRUE);
msg("You said " & msg("Say something: "),TRUE);
executeCommands("<");
END;

END "useEd"
```

Example 16.4-2. A Module That Uses TOYED (end)

17. Exceptions

Exceptions provide a mechanism for temporarily interrupting or permanently aborting the normal flow of control in a program. The program may deal with the exception in a manner determined at an earlier point in the program's execution, or it may choose not to deal with it at all. Exceptions are generated automatically when many kinds of errors occur. If a program chooses not to deal with an exception, the MAINSAIL runtime system usually calls the system procedure errMsg to report that the exception has not been handled.

17.1. Rationale behind Exceptions

Imagine a program that parses a complicated input file (e.g., a compiler that parses a program). The parser may be a recursive-descent parser (as in the program CALC of Example 7.2.2-2). Complicated input languages require complicated recursive-descent parsers; at any given time, such parsers may be many levels deep in procedure calls.

If an error occurs in the input file, the parser may become confused. It may be desirable to skip subsequent input until some recognizable terminating token is found (e.g., an "END" in a MAINSAIL source program). It is certainly possible for the procedure that recognizes the error in the source to skip forward in the input to the terminating token; however, it is then also desirable to abort all the procedures that were parsing things that were supposed to appear in the input before that terminating token.

Without exceptions, some sort of failure code must be returned by a procedure that recognizes that an error has occurred in the input, and examined by each calling procedure, until the procedure is resumed that was trying to parse the construct ending with the terminating token to which the parser skipped. The code to do this can be complex and difficult to read. For example, consider a procedure to parse the following partial grammar:

```
x => a b c "end"
  => d "end"
```

That is, an item x is composed of the series of three items a, b, and c, followed by the keyword "end", or composed of the single item d followed by the keyword "end". The procedure to parse each item may return in one of three ways:

1. The item may be parsed correctly.

2. An error may occur that requires that the parser skip to the keyword "end".

3. Some other sort of error may occur, requiring the calling procedure (the procedure to parse the item x) to be aborted.

The code to parse an item x in the above grammar might look something like Example 17.1-1. Although the task to be accomplished by the procedure parseX is not conceptually very complicated, the code for parseX is complicated because of the bookkeeping required to keep track of error codes.

```
<return code> PROCEDURE parseX;
BEGIN
<return code> errorCode;
IF <what's in the input is the beginning of an a-item>
    THENB CASE errorCode := parseA OFB
         [<failure, skip to "end">] <skip to "end">;
         [<success>]
             CASE errorCode := parseB OFB
                 [<failure, skip to "end">]
                     <skip to "end">;
                 [<success>]
                     CASE errorCode := parseC OFB
                         [<failure, skip to "end">]
                             <skip to "end">;
                         [<success>]
                             RETURN(IF checkFor("end") THEN
                                     <success>
                                 EL <missing "end" error>);
                         [ ] END;
                 [ ] END;
         [ ] END
EF <what's in the input is the beginning of a d-item>
    THEN CASE errorCode := parseD OFB
         [<failure, skip to "end">] <skip to "end">;
         [<success>] RETURN(
                 IF checkFor("end") THEN <success>
                 EL <missing "end" error>);
         [ ] END
EL RETURN(<some error code: don't see a-item or d-item>);
RETURN(IF <skipped to "end"> THEN <appropriate error code>
    EL errorCode);
END;
```

Example 17.1-1. Recursive-Descent Parsing without Exceptions

What the writer of the procedure parseX would really like to have done is something like:

```
If any error occurs in the following:

          <parse a, b, and c, or parse
           d, as appropriate>

then handle it in the following way:

          <skip to "end" if we should,
           otherwise pass the error on
           to the calling procedure>
```

Exceptions provide a mechanism for doing exactly this. With exceptions, the procedure parseX of Example 17.1-1 can be rewritten as shown in Example 17.1-2.

```
PROCEDURE parseX;
BEGIN
$HANDLE
    IF <what's in the input is the beginning of an a-item>
        THENB parseA; parseB; parseC END
    EF <what's in the input is the beginning of a d-item>
        THEN parseD
    EL $raise(<some error>)
$WITH
    IF $exceptionName = <failure, skip to "end"> THEN
        <skip to "end">
    EL $raise; # Propagate the exception
IF NOT checkFor("end") THEN $raise(<missing "end">);
END;
```

Example 17.1-2. Recursive-Descent Parsing with Exceptions

17.2. The Handle Statement

The form of the Handle Statement is:

```
$HANDLE <handled statement> $WITH <handler statement>
```

"$HANDLEB" and "$WITHB" are abbreviations for "$HANDLE BEGIN" and "$WITH BEGIN", respectively.

If an exception occurs within the handled statement (or within any procedures invoked from the handled statement), then the execution of the handled statement is temporarily suspended and the handler statement (or just "handler") is executed. If no exception occurs within the handled statement, the handler is ignored; i.e., after the handled statement terminates, control resumes at the statement following the Handle Statement.

The most direct way to cause (raise) an exception is to call the system procedure $raise with at least one argument:

```
$raise(<exception name>)
```

The exception name (the first argument to $raise) is any string.

A handler may deal with an exception in any of the following ways:

1. It can propagate (pass on) the exception to the next handler, if any, by calling the system procedure $raise with no arguments. The handler should always do this if it does not recognize the exception (as identified by $exceptionName).

2. It can resume the suspended handled statement by calling the system procedure $raiseReturn.

3. If the end of the handler statement is reached (execution "falls out" of the handler) and neither $raise nor $raiseReturn has been called (or if the handler is exited by means of a Done, Continue, or Return Statement), then the execution of the suspended handled statement is aborted, and execution resumes immediately following the Handle Statement (or at the location implied by the Done, Continue, or Return Statement).

The handler is said to have "handled" the exception if it deals with it in the second or third way listed above.

The three ways to deal with an exception are illustrated by Examples 17.2-1, 17.2-2, and 17.2-3. The three modules differ only in the actions taken by the handlers in the Handle Statements. The fatal error message in Example 17.2-1 is generated by the MAINSAIL runtime system, which intercepts the exception after it is propagated by the call to $raise with no arguments (the runtime system then also generates the exception "MAINSAIL: System exception").

```
The module:

    BEGIN "excpt1"

    INITIAL PROCEDURE;
    BEGIN
    write(logFile,"Before Handle Statement." & eol);
    $HANDLEB
        write(logFile,
            "About to raise an exception..." & eol);
        $raise("Exception!");
        write(logFile,
            "In handled statement, after exception."
            & eol) END
    $WITHB
        write(logFile,"Intercepted exception ",
            $exceptionName,eol);
        $raise END;
    write(logFile,"After Handle Statement." & eol);
    END;

    END "excpt1"

executes as follows:

    *excpt1<eol>
    Before Handle Statement.
    About to raise an exception...
    Intercepted exception Exception!
    Intercepted exception MAINSAIL: System exception


    FATAL: No handler for exception Exception!
    In module EXCPT1 at offset 164 (decimal)
    Error Response:
```

Example 17.2-1. Propagating an Exception with $raise

```
The module:

    BEGIN "excpt2"

    INITIAL PROCEDURE;
    BEGIN
    write(logFile,"Before Handle Statement." & eol);
    $HANDLEB
        write(logFile,
            "About to raise an exception..." & eol);
        $raise("Exception!");
        write(logFile,
            "In handled statement, after exception."
            & eol) END
    $WITHB
        write(logFile,"Intercepted exception ",
            $exceptionName,eol);
        $raiseReturn END;
    write(logFile,"After Handle Statement." & eol);
    END;

    END "excpt2"

executes as follows:

    *excpt2<eol>
    Before Handle Statement.
    About to raise an exception...
    Intercepted exception Exception!
    In handled statement, after exception.
    After Handle Statement.
```

Example 17.2-2. Resuming a Suspended Handled Statement with $raiseReturn

```
The module:

    BEGIN "excpt3"

    INITIAL PROCEDURE;
    BEGIN
    write(logFile,"Before Handle Statement." & eol);
    $HANDLEB
        write(logFile,
            "About to raise an exception..." & eol);
        $raise("Exception!");
        write(logFile,
            "In handled statement, after exception."
            & eol) END
    $WITH
        write(logFile,"Intercepted exception ",
            $exceptionName,eol);
        # Just fall out of handler; no $raise or
        # $raiseReturn.
    write(logFile,"After Handle Statement." & eol);
    END;

    END "excpt3"

executes as follows:

    Before Handle Statement.
    About to raise an exception...
    Intercepted exception Exception!
    After Handle Statement.

The call to write immediately after the call to $raise is
never reached, since the handled statement is aborted
when the handler terminates.
```

Example 17.2-3. Falling Out of a Handler

17.3. Exceptions Raised Automatically and Predefined Exceptions

In a variety of circumstances, MAINSAIL automatically raises an exception. Exceptions may be raised automatically by:

- A MAINSAIL program error, e.g., an array subscript error, a nullPointer data access, a case index error, falling out of a typed procedure without returning a value, etc.

- An arithmetic error, e.g., overflow, underflow, division by zero, etc. (the exceptions signaled depend on the operating system; not all operating systems permit MAINSAIL to intercept all arithmetic errors).

- End-of-file on cmdFile or on "TTY" during a call to ttyRead (see the descriptions of $cmdFileEofExcpt and $ttyEofExcpt in the "MAINSAIL Language Manual").

- The system procedure errMsg (see Section 17.6).

- Falling out of a handler of which the handled statement has invoked one or more procedures. The invoked procedures, being part of the handled statement, are aborted; the $abortProcedureExcpt exception is raised in each active (currently executing) Handle Statement in the aborted procedures.

A list of the predefined exceptions and their significances may be found in the "MAINSAIL Language Manual".

One predefined exception that is never raised automatically is the $abortProgramExcpt exception. It is raised by the "MAINSAIL: Abort program" (or an abbreviation thereof; "a p" usually suffices) response to the "Error response:" prompt. When it is raised, the currently executing MAINSAIL program is aborted, usually returning control to the program from which the current program was invoked; first, however, each active handler in the program is given a chance to handle $abortProcedureExcpt (be sure to distinguish between $abortProgramExcpt and $abortProcedureExcpt!).

Handling $abortProcedureExcpt is usually the correct method of allowing a procedure or program to clean up after itself if it is unexpectedly aborted. See the program fragment of Example 17.3-1.

17.4. Multiple Handlers and Multiple Exceptions

More than one Handle Statement may be executing at any given time (e.g., if a Handle Statement is contained inside another (see Example 17.4-1) or contained in a procedure called by another). Furthermore, more than one exception may be active at a given time (e.g., if an exception is raised from a handler that is handling another exception).

```
PROCEDURE cleanUp;
BEGIN
<close files if open>
<dispose of data structures if necessary>
<unbind or dispose of modules>
<release scan bits or integers if any used>
END;



INITIAL PROCEDURE;
$HANDLEB <perform the actions of the program>; cleanUp END
$WITHB
    IF $exceptionName = $abortProcedureExcpt THEN cleanUp;
    $raise END;
```

Example 17.3-1. Cleaning Up by Handling $abortProcedureExcpt

```
In the code:

    $HANDLEB
        ...
        $HANDLEB
            ...
            $HANDLE xxx
            $WITH ...
            ... END
        $WITH ...
        ... END
    $WITH ...

three Handle Statements are executing when the statement
"xxx" is executed.
```

Example 17.4-1. Simultaneously Active Handle Statements

It is often the case that several Handle Statements are executing simultaneously. Multiple Handle Statements might be used in a recursive-descent parser, for example, to implement skipping to terminating tokens of several different constructs in case of error, as in Example 17.1-2. For example, if the grammar parsed in Example 17.1-2 is expanded to:

```
x => a b c "end"
  => d "end"
d => y z "."
```

then it might be parsed by a program fragment resembling Example 17.4-2.

```
PROCEDURE parseD;
BEGIN
$HANDLEB
    parseY; parseZ END
$WITH
    IF $exceptionName = <failure, skip to "."> THEN
        <skip to ".">
    EL $raise; # Propagate the exception
IF NOT checkFor(".") THEN $raise(<missing ".">);
END;



PROCEDURE parseX;
BEGIN
$HANDLE
    IF <what's in the input is the beginning of an a-item>
        THENB parseA; parseB; parseC END
    EF <what's in the input is the beginning of a d-item>
        THEN parseD
    EL $raise(<some error>)
$WITH
    IF $exceptionName = <failure, skip to "end"> THEN
        <skip to "end">
    EL $raise; # Propagate the exception
IF NOT checkFor("end") THEN $raise(<missing "end">);
END;
```

Example 17.4-2. Two Simultaneously Active Handle Statements

At the time the procedure parseY is called in Example 17.4-2, two Handle Statements are active: the one in the procedure parseD, and the one in the procedure parseX. Since the Handle Statement in parseD has been entered more recently, its handler will intercept any exceptions before the handler in parseX. If the handler in parseD propagates an exception with an argumentless call to $raise, the handler in parseX then receives control. In general, when an exception is propagated, it is intercepted in the order of most recently to least recently entered relevant Handle Statement.

Simultaneously active Handle Statements may be called "nested" Handle Statements; an exception raised while another is being processed may be referred to as a "nested exception". Example 17.4-3 shows a module that uses nested Handle Statements and nested exceptions; Example 17.4-4 shows its output.

```
C: C exception
B1: C exception
D: D exception
B2: D exception
D: returned from D exception
B3: C exception
C: MAINSAIL: Abort procedure
D: D exception
A: D exception
D: MAINSAIL: Abort procedure
```

Example 17.4-4. Output from Example 17.4-3

Example 17.4-5 shows conceptual views of the procedure call stack at a series of moments during the execution of Example 17.4-3. Moment 1 shows the entry into the procedure a. Moment 2 shows the (handled) call to procedure b; moment 3, the (handled) call from b to c. At moment 4, the Handle Statement in c is entered; at moment 5, the exception "C exception" has been raised, and the handler part of c's Handle Statement is entered. At this point, c writes "C: C exception" to logFile. The handled part of c's Handle Statement has been suspended (as indicated by "suspnd hndld stmt"). At moment 6, the call to $raise in c's handler has propagated the exception "C exception" to b's (outer) Handle Statement, the handled part of which is thereby suspended. At this point, b writes "B1: C exception" to logFile. At moment 7, b calls d from b's inner Handle Statement, and d enters the handled part of its Handle Statement. At moment 8, d has raised the exception "D exception", and d's handler has intercepted the exception. At this point, d writes "D: D exception" to logFile. At moment 9, the call to $raise propagates the exception "D exception" to b, d's caller (not to c, which is not in d's call chain, although it is shown on the stack between b and d). At this point, b writes "B2: D exception" to logFile. Moment 10 looks like moment 7; the call to $raiseReturn in b restores control to d's handled statement at the point where it was suspended. The call to $raiseReturn handles the exception "D exception" so that it is no longer pending. At this point, d writes "D: returned from D exception" to logFile. If another exception were raised at this point from d's handled statement, the call stack would again go through states similar to moments 8, 9, and 10.

```
BEGIN "mulExc"

PROCEDURE d;
$HANDLEB
    $raise("D exception");
    write(logFile,"D: returned from D exception" & eol);
    END
$WITHB
    write(logFile,"D: ",$exceptionName,eol); $raise END;



PROCEDURE c;
$HANDLE $raise("C exception")
$WITHB
    write(logFile,"C: ",$exceptionName,eol); $raise END;



PROCEDURE b;
BEGIN
$HANDLE c
$WITHB
    write(logFile,"B1: ",$exceptionName,eol);
    $HANDLE d
    $WITHB
        write(logFile,"B2: ",$exceptionName,eol);
        $raiseReturn END;
    write(logFile,"B3: ",$exceptionName,eol) END;
d;
END;



INITIAL PROCEDURE a;
$HANDLE b
$WITH write(logFile,"A: ",$exceptionName,eol);

END "mulExc"
```

Example 17.4-3. Nested Handlers and Exceptions

```
                                              +------+
                                           -> |  c   |
                                              +------+
                      +------+             handled stmt
                   -> |  b   |                |  b   |
                      +------+                +------+
     +------+      handled stmt            handled stmt
  -> |  a   |         |  a   |                |  a   |
     +------+         +------+                +------+
        1                2                       3

                      +------+                +------+
                   <C exception>        suspnd hndld stmt
     +------+       -> handler                |  c   |
  -> handled stmt  suspnd hndld stmt          +------+
     |  c   |         |  c   |             <C exception>
     +------+         +------+              -> handler
   handled stmt     handled stmt        suspnd hndld stmt
     |  b   |         |  b   |                |  b   |
     +------+         +------+                +------+
   handled stmt     handled stmt           handled stmt
     |  a   |         |  a   |                |  a   |
     +------+         +------+                +------+
        4                5                       6
```

Example 17.4-5. Explanation of Example 17.4-3 with Stack Diagrams (continued)

```
                       +------+          +------+
                    <D exception>   suspnd hndld stmt
     +------+       -> handler           |  d   |
-> handled stmt   suspnd hndld stmt      +------+
     |  d   |          |  d   |     suspnd hndld stmt
     +------+          +------+          |  c   |
suspnd hndld stmt suspnd hndld stmt      +------+
     |  c   |          |  c   |       <D exception>
     +------+          +------+       -> handler
   handled stmt         handled stmt   suspnd hndld stmt
 <C exception>      <C exception>     <C exception>
     handler               handler           handler
suspnd hndld stmt suspnd hndld stmt suspnd hndld stmt
     |  b   |          |  b   |          |  b   |
     +------+          +------+          +------+
   handled stmt         handled stmt      handled stmt
     |  a   |          |  a   |          |  a   |
     +------+          +------+          +------+
        7                 8                 9

     +------+
-> handled stmt
     |  d   |
     +------+
suspnd hndld stmt      +------+
     |  c   |     suspnd hndld stmt      +------+
     +------+          |  c   |     <Abort procedure>
   handled stmt        +------+       -> handler
 <C exception>     <C exception>    suspnd hndld stmt
     handler        -> handler           |  c   |
suspnd hndld stmt suspnd hndld stmt      +------+
     |  b   |          |  b   |          |  b   |
     +------+          +------+          +------+
   handled stmt         handled stmt      handled stmt
     |  a   |          |  a   |          |  a   |
     +------+          +------+          +------+
       10                11                12
```

Example 17.4-5. Explanation of Example 17.4-3 with Stack Diagrams (continued)

```
                                          +-----+
                                       <D exception>
                       +-----+         -> handler
                  -> handled stmt    suspnd hndld stmt
                       |  d  |            |  d  |
     +-----+           +-----+            +-----+
  -> |  b  |           |  b  |            |  b  |
     +-----+           +-----+            +-----+
   handled stmt      handled stmt       handled stmt
     |  a  |           |  a  |            |  a  |
     +-----+           +-----+            +-----+
       13                14                 15

     +-----+
suspnd hndld stmt      +-----+
     |  d  |      <Abort procedure>
     +-----+        -> handler
     |  b  |      suspnd hndld stmt
     +-----+           |  d  |
  <D exception>        +-----+
  -> handler           |  b  |
suspnd hndld stmt      +-----+            +-----+
     |  a  |           |  a  |         -> |  a  |
     +-----+           +-----+            +-----+
       16                17                 18
```

Example 17.4-5. Explanation of Example 17.4-3 with Stack Diagrams (end)

At moment 11, the procedure d has returned normally to its caller b, where control resumes in the outer handler. The exception "C exception" has not yet been handled, and is therefore still pending. At this point, b writes "B3: C exception" to logFile. At moment 12, control falls out of b's outer handler, thereby handling the exception "C exception". Since the procedure C is still suspended, the exception "MAINSAIL: Abort procedure" ($abortProcedureExcpt) must be raised within it. At this point, c writes "C: MAINSAIL: Abort procedure" to logFile. At moment 13, c has been aborted. At moment 14, b has made its call to d, and d enters its handled statement. At moment 15, d has raised the exception "D exception", and thereby entered its handler. At this point, d writes "D: D exception" to logFile. At moment 16, the call to $raise in d propagates the exception "D exception" to the still active handler in a. At this point, a writes "A: D exception" to logFile. At moment 17, control falls out of a's handler, handling "D exception". $abortProcedureExcept is raised in d, and d writes "D: MAINSAIL: Abort procedure" to logFile. At moment 18, d has been aborted, along with b, which had no active Handle Statement, and therefore did not intercept $abortProcedureExcpt. At this point, a returns normally to its caller.

17.5. Information about the Current Exception

A handler can obtain the name of the current exception by calling $exceptionName. Other information about the exception (e.g., whether or not the handler can resume execution at the place where the exception occurred) can be obtained by calling $exceptionBits; consult the description of $exceptionBits in the "MAINSAIL Language Manual".

When an exception is caused by means of the system procedure $raise, extra information about the exception can be passed by means of the parameters exceptionStringArg1, exceptionStringArg2, and exceptionPointerArg. A handler can access the values passed as these arguments by means of $exceptionStringArg1, $exceptionStringArg2, and $exceptionPointerArg.

If, for efficiency, a handler does not want to perform string comparison on an exception name to see which exception has occurred, it can compare the pointer passed in exceptionPointerArg; see Section 1.14 of part II of the "MAINSAIL Tutorial".

17.6. errMsg and Exceptions

Whenever the system procedure errMsg is called, it raises the exception $systemExcpt ("MAINSAIL: System exception"). This allows any program currently running to examine the error message and determine whether it should take some action. The program may decide to prevent the error message from being displayed.

Before writing a message to logFile, errMsg raises the exception $systemExcpt, passing the two string arguments to errMsg in exceptionStringArg1 and exceptionStringArg2. A handler may examine these as shown in Example 17.6-1. If all handlers propagate the exception with an argumentless call to $raise, the error message is written to logFile as usual, and errMsg returns true. If a handler calls $raiseReturn, the error message is suppressed; errMsg assumes that appropriate action has been taken to handle the error condition, and returns false.

A program may define additional responses that are valid at the "Error response:" prompt by calling $registerException. This is the mechanism whereby the MAINSAIL compiler defines the "MAINSAIL: Abort compilation" response. See the description of $registerException in the "MAINSAIL Language Manual".

17.7. Exceptions and Coroutines

Exceptions in separate coroutines are handled as described in Section 19.5.

```
The module:

    BEGIN "errExc"

    INITIAL PROCEDURE;
    BEGIN
    $HANDLE
        write(logFile,IF errMsg("This is an","error") THEN
            "TRUE" EL "FALSE",eol)
    $WITHB write(logFile,
            "$exceptionName = ",$exceptionName,eol &
            "$exceptionStringArg1 = ",
                $exceptionStringArg1,eol &
            "$exceptionStringArg2 = ",
                $exceptionStringArg2,eol);
        $raise END;
    $HANDLE
        write(logFile,IF errMsg("This is another","error")
            THEN "TRUE" EL "FALSE",eol)
    $WITHB write(logFile,
            "$exceptionName = ",$exceptionName,eol &
            "$exceptionStringArg1 = ",
                $exceptionStringArg1,eol &
            "$exceptionStringArg2 = ",
                $exceptionStringArg2,eol);
        $raiseReturn END;
    END;

    END "errExc"

executes as follows:

    $exceptionName = MAINSAIL: System exception
    $exceptionStringArg1 = This is an
    $exceptionStringArg2 = error
```

Example 17.6-1. errMsg and $systemExcpt (continued)

```
ERROR: This is an error
Error Response: <eol>
TRUE
$exceptionName = MAINSAIL: System exception
$exceptionStringArg1 = This is another
$exceptionStringArg2 = error
FALSE
```

Example 17.6-1. errMsg and $systemExcpt (end)

17.8. Exception Caveats

XIDAK reserves the right to create new predefined exceptions. MAINSAIL programmers should be aware that system predefined exceptions, or undocumented exceptions used internally by the MAINSAIL runtime system, may be raised by many MAINSAIL constructs. Exception handlers must therefore be written to check that they are actually handling the exceptions they expect to handle. Do not issue a $raiseReturn or terminate a handler on an exception you don't recognize.

The proper format for a Handle Statement in a real program is thus always something like:

```
$HANDLE ...
$WITH
    IF <check for exception1> THEN
        ... deal with exception1...
    EF <check for exception2> THEN
        ... deal with exception2...
    ...
    EF <check for exceptionN> THEN
        ... deal with exceptionN...
    EL  $raise;
```

That is, it should include a $raise for any exception it doesn't recognize.

18. Low-Level Data Types; Memory Management

MAINSAIL provides two data types not described so far: address and charadr. Address and charadr are used for direct manipulation of the contents of memory. Such manipulation is not need by all programs; some programs require it because:

- higher-level facilities provided by MAINSAIL are not sufficiently efficient for the task at hand, or

- a MAINSAIL program must interface with a foreign language, an operating system, or a hardware-dependent special memory location, such as a memory-mapped I/O register.

MAINSAIL also provides facilities that direct the course of memory management, which is otherwise automatic. The most frequently used facility is the ability to dispose data structures, i.e., explicitly recycle the memory they occupy.

18.1. Organization of Memory

Memory is organized in different ways on different processors. The type of memory is generally classified according to the number of bits accessible at a given address. Many processors organize memory as a series of eight-bit bytes; some processors instead address words of 16, 32, or 64 bits (processors with other sizes have existed as well, but as explained below, MAINSAIL runs only on processors where the word size is a multiple of eight bits). In order to make it possible to write portable programs, MAINSAIL works with the abstract notions of "storage units" and "character units", and provides ways of determining how many bits each of these units contains.

18.1.1. Storage Units and Character Units

A storage unit is the amount of memory addressed by a MAINSAIL address; when the address is incremented by one, it addresses the next storage unit (although on some machines, due to alignment considerations imposed by the processor, it is not legal to load or store using certain address values, e.g., odd addresses). A character unit is the amount of memory occupied by a single character (as stored in a MAINSAIL string), which is the amount of memory addressed by a MAINSAIL charadr. When the charadr is incremented by one, it addresses the next character unit (there is no restriction on character alignments; each consecutive charadr may be used to load or store a character).

The character unit size on every machine that supports MAINSAIL is eight bits, and the storage unit size is always an exact multiple of the character unit size.

Memory can be viewed as a contiguous, linear sequence of storage units or character units (although if MAINSAIL is ever supported on a machine with a segmented address space, memory will appear as several distinct linear segments). The storage units and character units are superimposed on top of one another. On a typical eight-bit byte machine, the storage unit and character unit coincide; each is eight bits:

```
         c.u.     c.u.     c.u.     c.u.
       +--------+--------+--------+--------+
 . . . | 8 bits | 8 bits | 8 bits | 8 bits | . . .
       +--------+--------+--------+--------+
         s.u.     s.u.     s.u.     s.u.
```

On a typical 16-bit word machine (e.g., the Data General ECLIPSE MV), the storage unit is 16 bits (since incrementing an address by one accesses the next 16-bit word), and the character unit is eight bits; therefore, there are two character units per storage unit:

```
         c.u.     c.u.     c.u.     c.u.
       +--------+--------+--------+--------+
 . . . | 8 bits | 8 bits | 8 bits | 8 bits | . . .
       |     16 bits     |     16 bits     |
       +-----------------+-----------------+
              s.u.              s.u.
```

On the 64-bit Cray, if such a MAINSAIL implementation were ever written, eight eight-bit characters would fit in one 64-bit storage unit.

The number of bits per storage unit on the target processor is given by the system macro $bitsPerStorageUnit; the number of bits per character unit, by $bitsPerChar (which is always 8). $charsPerStorageUnit is the number of character units per storage unit, i.e., $bitsPerStorageUnit DIV $bitsPerChar.

MAINSAIL data types (boolean, (long) integer, (long) real, (long) bits, string, pointer, address, and charadr) each occupy an integral number of storage units. As mentioned in Section 10.11, the system procedure "size" returns the number of storage units occupied by each type. Another form of size, which takes a class or pointer, returns the number of storage units occupied by the fields of the class or by the record to which the pointer points. $ioSize returns the number of character or storage units occupied by a data type in a data file.

18.1.2. Loading, Storing, Reading, and Writing

Addresses and charadrs are frequently used for loading or storing values of individual data types from or to memory. The address load procedure for each data type is named "xxLoad", where xx is the prefix for the data type; the complete list of load procedures is:

boLoad	boolean
iLoad	integer
liLoad	long integer
rLoad	real
lrLoad	long real
bLoad	bits
lbLoad	long bits
sLoad	string
pLoad	pointer
aLoad	address
cLoad	charadr

Each load procedure takes an address and optional integer offset (number of storage units) from that address.

For example, assume an eight-bit-byte-organized memory contains hexadecimal eight-bit bytes at the hex addresses shown:

'H4EFCA2	E2
'H4EFCA3	14
'H4EFCA4	06
'H4EFCA5	9A
'H4EFCA6	37
'H4EFCA7	00
'H4EFCA8	21
'H4EFCA9	CC

Assume also that the data type integer occupies two storage units (bytes), the data type long bits occupies four storage units, higher-order bytes are stored at lower addresses, the address variable a represents the address 'H4EFCA4, and the address variable aa represents the address 'H4EFCA6. Then:

```
iLoad(a)      = cvi('H069A)
iLoad(aa)     = cvi('H3700)
lbLoad(a)     = 'H069A3700L
iLoad(a,2)    = iLoad(aa)      = cvi('H3700)
iLoad(a,-2)   = iLoad(aa,-4)   = cvi('HE214)
lbLoad(a,2)   = lbLoad(aa)     = 'H370021CCL
```

The address forms of store modify the contents of memory. store is generic; each instance takes an address, the value to be stored, and an optional integer offset from the address. For example, if the above memory layout is modified by:

```
store(a,cvi('H4567),-2);
store(aa,'HFEDCBA98L)
```

then the new memory layout looks like:

```
'H4EFCA2        45
'H4EFCA3        67
'H4EFCA4        06
'H4EFCA5        9A
'H4EFCA6        FE
'H4EFCA7        DC
'H4EFCA8        BA
'H4EFCA9        98
```

Load and store forms also exist for characters; the only data type that may be loaded or stored using a charadr is an integer (since integers are used to represent character codes). The charadr load is called "cLoad" (like the form that loads a charadr from an address); the charadr store form is called "store". If characters are one byte, then, given the above (modified) memory layout and a charadr c representing the charadr 'H4EFCA5:

```
cLoad(c)     = cvi('H9A)
cLoad(c,3)   = cvi('HBA)
cLoad(c,-2)  = cvi('H67)
```

When the memory layout is modified by:

```
store(c,cvi('H01));
store(c,cvi('H03),3)
```

it becomes:

```
'H4EFCA2        45
'H4EFCA3        67
'H4EFCA4        06
'H4EFCA5        01
'H4EFCA6        FE
'H4EFCA7        DC
'H4EFCA8        03
'H4EFCA9        98
```

If several consecutive data are to be loaded or stored, the address forms of read and write or the charadr forms of cRead and cWrite may be used. read (or cRead) increments its address (or charadr) parameter after each value loaded; write (or cWrite) increments after each value stored. For example, if a initially represents the address 'H4EFCA2, then:

```
write(a,'H42C1,'H33991234L)
```

modifies the above memory layout to:

```
'H4EFCA2        42
'H4EFCA3        C1
'H4EFCA4        33
'H4EFCA5        99
'H4EFCA6        12
'H4EFCA7        34
'H4EFCA8        03
'H4EFCA9        98
```

and modifies a to represent the address 'H4EFCA8. If c initially represents the charadr 'H4EFCA3, then:

```
i := cRead(c); j := cRead(c)
```

sets i to cvi('HC1), j to cvi('H33), and c to 'H4EFCA5.

18.1.3. Address Constants

There is no way to represent an address or charadr constant other than the Zero values, designated by the keywords "NULLADDRESS" and "NULLCHARADR". When an address value must be specified as a constant (which is rare; it is usually computed as an offset from the address of a scratch area allocated with newScratch or newPage), it is usually specified as:

```
cva(<long bits constant>)
```

Likewise, when an address must be printed, it is converted to a long bits (with cvlb), and the long bits converted to a string. Each processor has a radix in which addresses are customarily printed; this radix is given by $preferredRadix, which should be used when a portable program displays addresses.

18.1.4. Loading and Storing Examples

The procedure newScratch returns an address pointing to a region of cleared memory of the size, in storage units, specified by its integer parameter. Such scratch space may be used for a variety of purposes. When a program has finished using scratch space, it must return it to the MAINSAIL memory manager by means of the procedure scratchDispose; scratch space is not reclaimed by the garbage collector.

$newScratchChars allocates the same kind of scratch space as newScratch, but measured in character units.

The overhead for an array index operation in MAINSAIL is usually negligible when considered in the context of the algorithm in which the array is used. In certain tight loops in code compiled with array subscript checking on, however, it can be advantageous to load data from an address rather than from an array, and avoid the subscript checking overhead. Likewise, although the string forms of cRead and cWrite from a string usually perform adequately, cRead has to do a null string check that may not be necessary if the source string is known to be non-null, and cWrite has to ensure that the string to which the character is added is at the end of MAINSAIL's string space (see Section 18.5) so that the new character can be appended.

The procedure deVowel of Example 18.1.4-1 uses a boolean pseudo-array implemented as an address and charadr operations on strings to perform a fast string scan, removing vowels from the source string to produce a result string. newString and $getInArea are described in Section 18.5. $maxChar is defined as the highest character code in the target system's character set. The string form of cvc returns the first charadr at which the characters of its argument are stored.

18.1.5. Copy and Clear

When a block of memory more than a few storage units long must be copied or cleared, the most efficient way to do so is to call the built-in procedure copy or clear. For example:

```
copy(a,aa,30 * size(longIntegerCode))
```

is at least as efficient as (and certainly more succinct than):

```
aaa := a; aaaa := aa;
FOR i := 1 UPTO 30 DOB
    read(aaa,li); write(aaaa,ii) END
```

Forms of copy and clear that operate on arrays and records also exist; consult the "MAINSAIL Language Manual" for more details.

18.1.6. Alignment of Addresses

It may not be possible to store or retrieve data at certain addresses. Some processors require that only even addresses be used for data access; others require that addresses be multiples of

```
BEGIN "deVowl"

ADDRESS vowelTbl;
CHARADR buf;
INTEGER bufLen;

STRING PROCEDURE deVowel (STRING s);
# Compute a vowel-free version of s as fast as possible
BEGIN
INTEGER i,j,ch;
CHARADR c1,c2;
IF bufLen < length(s) THENB # buffer is not big enough
    IF buf THEN scratchDispose(buf);
    buf := $newScratchChars(bufLen := length(s)) END;
c1 := cvc(s); c2 := buf; i := length(s); j := 0;
WHILE i .- 1 GEQ 0 DO
    IF NOT boLoad(vowelTbl,
        (ch := cRead(c1)) * size(booleanCode)) THENB
        cWrite(c2,ch); j .+ 1 END;
RETURN($getInArea(newString(buf,j)));
END;
```

Example 18.1.4-1. Using Addresses and Charadrs for a Fast String Scan (continued)

```
INITIAL PROCEDURE;
BEGIN
INTEGER i,ch;
STRING s;
ADDRESS a;
# Initialize vowel table:
a := vowelTbl :=
    newScratch(size(booleanCode) * ($maxChar + 1));
    # $maxChar + 1 is the total number of characters in
    # the character set.
FOR i := 0 UPTO $maxChar DO
    write(a,ch := cvu(i) = 'A' OR ch = 'E' OR ch = 'I'
        OR ch = 'O' OR ch = 'U');
DOB write(logFile,"String to de-vowel (<eol> to quit): ");
    read(cmdFile,s); IF NOT s THEN DONE;
    write(logFile,deVowel(s),eol) END;
scratchDispose(vowelTbl);
IF a := cva(buf) THEN scratchDispose(a);
END;

END "deVowl"
```

Example 18.1.4-1. Using Addresses and Charadrs for a Fast String Scan (end)

four. For this reason, addresses in portable programs should always be computed in terms of linear combinations of the sizes of MAINSAIL data types, rather than specified with explicitly integer constants.

When a charadr is converted to an address with the system procedure "cva", cva may round the charadr down to the nearest correctly aligned address. Thus, "cvc(cva(c))", where c is a charadr, does not necessarily equal c.

18.1.7. Pages

MAINSAIL groups storage units into contiguous groups called pages. The number of storage units in a page is usually a power of 2, and typically varies from 256 to 4096. The MAINSAIL runtime system often performs file I/O in units of pages, so the page size is usually chosen to be the same as the underlying operating system's disk page size, so as to optimize the data transfer rate to and from disk. The number of storage units in a page is given by the system macro $pageSize. $charsPerPage, the number of character units per page, is equal to $charsPerStorageUnit * $pageSize.

Example 18.1.4-1 shows uses of the procedures newScratch and $newScratchChars for obtaining scratch space. When large continuous blocks of memory must be obtained, it is often convenient to request a number of pages rather than a number of storage or character units. In such a case, the system procedure newPage may be called instead of newScratch or $newScratchChars.

At any given time, the MAINSAIL runtime system recognizes certain portions of the MAINSAIL process's address space as being under its control (usually because it explicitly requested the memory from the operating system). These areas of memory are referred to as being "within the MAINSAIL page map" (because it is the portion of memory displayed by the utility MEMMAP) or (more loosely) "within the MAINSAIL address space". The coarse divisions of MAINSAIL's memory management algorithms are based on pages, which is why MEMMAP prints out usage information on a per-page basis.

18.2. dispose

Chapter 14 introduced one use of the system procedure "dispose". Forms of dispose also exist for records and arrays. "dispose(p)", where p is a pointer to a record, or "dispose(ary)", where ary is an array, tells MAINSAIL to free the memory occupied by the record or array immediately, instead of waiting for a garbage collection.

It is important to use dispose in programs that generate a lot of garbage in order to prevent too-frequent garbage collections. But dispose is dangerous! Many, if not most, bugs that are difficult to track are due to improper use of dispose, either disposing of a data structure that has already been disposed or accessing a field or element of a disposed data structure. Section 4.2 of part II of the "MAINSAIL Tutorial" contains an extensive discussion of the techniques that may be used to track such bugs.

Programs that clean up after themselves by disposing of data structures allocated during execution should handle $abortProcedureExcpt (see Section 17.3) with a routine that performs the clean-up, as shown in Example 17.3-1. In Example 18.2-1, the module of Example 11.5-2 is modified to dispose of the tree it constructs. The file pointer f has been moved to be an outer variable, so as to be accessible both to the initial procedure and to the procedure cleanUp.

The Structure Blaster procedure $structureDispose provides a less efficient but very convenient alternative to the procedure infixDispose in Example 18.2-1; see the "MAINSAIL Structure Blaster User's Guide" for details. Alternatively, the entire structure could have been maintained in an area, and the entire area disposed with $disposeArea, as described in the "MAINSAIL Language Manual".

When scratch space (obtained with newPage or newScratch) is disposed, special procedures must be called to free the space. scratchDispose is called for space acquired with newScratch; pageDispose for space acquired with newPage.

```
BEGIN "binTr2"

CLASS bin (
    POINTER(bin) left,right;
    STRING here;
);

POINTER(bin) root;
POINTER(textFile) f;

PROCEDURE alphabetize (STRING s; MODIFIES POINTER(bin) p);
BEGIN
IF NOT p THENB # create the node
    p := new(bin); p.here := s; RETURN END;
CASE compare(s,p.here,upperCase) OFB
    [-1] alphabetize(s,p.left);
    [0] RETURN;
    [1] alphabetize(s,p.right);
    END;
END;



PROCEDURE infixPrint (POINTER(bin) p);
BEGIN
IF NOT p THEN RETURN;
infixPrint(p.left);
write(logFile,p.here,eol);
infixPrint(p.right);
END;



PROCEDURE infixDispose (POINTER(bin) p);
BEGIN
IF p.left THEN infixDispose(p.left);
IF p.right THEN infixDispose(p.right);
dispose(p); # Modifies p to NULLPOINTER
END;
```

Example 18.2-1. Binary Tree Program Modified to Clean Up after Itself (continued)

```
PROCEDURE cleanUp;
BEGIN
IF f THEN close(f);
IF root THEN infixDispose(root);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

$HANDLEB
    open(f,"Input file: ",input!prompt);
    DOB read(f,s); IF NOT $gotValue(f) THEN DONE;
        IF s THEN alphabetize(s,root) END;
    infixPrint(root); cleanUp END
$WITHB
    IF $exceptionName = $abortProcedureExcpt THEN cleanUp;
    $raise END;
END;

END "binTr2"
```

Example 18.2-1. Binary Tree Program Modified to Clean Up after Itself (end)

18.3. High-Volume I/O

The procedures read and write are satisfactory methods of file I/O for most purposes. Some applications, however, may need to read in large quantities of data at a time, usually for efficiency; for example, it is more efficient to read in the contents of a 5000-element integer array all at once than to use a loop that reads one integer and stores it in the array on each iteration.

The procedures $storageUnitRead and $storageUnitWrite or $characterRead and $characterWrite may be used to read or write arbitrary quantities of data. $pageRead and $pageWrite read and write an integral number of pages at a time. Each of these procedures reads into or writes from scratch memory allocated with newPage or newScratch. Some uses of $storageUnitRead are shown in Example 18.3-1.

When high-volue I/O is used on a file, it is often advantageous to set the $unbuffered bit when the file is opened. Only certain I/O calls may be used when a file is opened $unbuffered, however; consult the description of "open" in the "MAINSAIL Language Manual" for details.

In order to read 5000 integers from a data file f into an appropriately sized area of scratch space, assuming a is an address variable (and that f is not opened for PDF I/O; see Section 10.12):

```
    a := newScratch(5000 * size(integerCode));
    $storageUnitRead(f,cvli(5000 * size(integerCode)),
        NULLPOINTER,0L,a);
```

In order to read the integers directly into an array ary:

```
    new(ary,1,5000);
    $storageUnitRead(f,cvli(5000 * size(integerCode)),
        cvp(ary),
        lDisplacement(cva(cvp(ary)),
                      $adrOfFirstElement(ary)));
```

where $adrOfFirstElement returns the address of the first element of the array. Don't do this:

```
    new(ary,1,5000);
    $storageUnitRead(f,cvli(5000 * size(integerCode)),
        NULLPOINTER,0L,$adrOfFirstElement(ary));
```

because a garbage collection might occur during the call $storageUnitRead. ary could be moved around in memory, and the address of first element changed; then the address passed to $storageUnitRead would no longer be valid.
The address parameter of $storageUnitRead should be specified only when reading into scratch space, and the pointer parameter only when reading into a high-level data structure (array, record, or data section).

Example 18.3-1. Use of $storageUnitRead

18.4. Control of Garbage Collection

The timing of garbage collections may be governed by modifying the system variable $collectLock and calling the system procedure $collect. Collections are "locked out" (do not occur automatically) whenever $collectLock has a non-zero value. Running with $collectLock set can prevent unnecessary collections, but running for too long without collecting can cause MAINSAIL to run out of memory if garbage is being generated.

18.5. $getInArea and newString

Most MAINSAIL strings are created (automatically) in a part of memory called "string space". The system procedures that perform cWrites, concatenations, and other functions that extend a string generally operate on strings in string space; they may copy their string arguments into string space if they are not there already. Only strings located in string space can be collected by the garbage collector.

It is possible to create strings not located in string space. This is useful, for example, when a string is returned by the operating system or by a foreign procedure (which do not have access to MAINSAIL's string space, and so cannot construct strings there), or when a string is constructed (for whatever reason) in scratch space.

The procedure newString constructs a string given a charadr at which the characters are located and a length for the string:

```
        STRING PROCEDURE newString (CHARADR c; INTEGER len);
```

newString does not actually copy the characters anywhere; it just returns a string that points at the characters at location c. If the charadr points into scratch space or into an area returned by a foreign procedure, it may be advisable to copy the characters into string space, so that the scratch space or foreign area can be reused to construct more strings (otherwise, the new strings would overwrite part or all of the previously constructed strings in the same location). The procedure $getInArea copies its string argument into string space:

```
                STRING PROCEDURE $getInArea (STRING s);
```

(As implied by the name, string space is actually maintained on a per-area basis. Consult the "MAINSAIL Language Manual" for details.) If a foreign procedure has returned a charadr c and a length l for a string it is returning to MAINSAIL, the proper way to construct a MAINSAIL string s in string space is:

```
                    s := $getInArea(newString(c,l));
```

18.6. Runtime Construction of Classes

A class is typically declared in the source text of a program; i.e., the class is constructed by the programmer before the program is compiled. Classes may, however, be built up at runtime; the fields of a class may also be explicitly examined at runtime. The relevant procedures are $classInfo, $className, $createClassDscr, $createRecord, and $dscrPtr; consult the "MAINSAIL Language Manual" for details.

19. Coroutines

Coroutines provide a way of maintaining multiple threads of execution within a MAINSAIL program. Coroutines do not execute simultaneously, but if properly orchestrated, can give the impression of simultaneous execution (like multiple processes on a time-sharing operating system).

Facilities are provided to initialize, resume (change the execution thread to that of another coroutine), and kill coroutines. The procedures $createCoroutine, $resumeCoroutine, and $killCoroutine are described in detail in the "MAINSAIL Language Manual".

The MAINSAIL STREAMS package provides some convenient facilities that schedule coroutines automatically. See the "MAINSAIL STREAMS User's Guide" for details.

19.1. Rationale behind Coroutines

Coroutines are useful when a program must perform several complex tasks that are interleaved with each other. It is difficult to this without coroutines, since the context of each task must be saved when another task is resumed. In theory, it is possible to save any amount of context in own or outer variables, but it is often inconvenient to do so. In particular, it may be convenient for two tasks to resume each other from several different procedures; this is impossible without coroutines.

Coroutines permit a cleaner formulation of producer-consumer algorithms than programs that do not use coroutines, particularly if the production or consumption occurs at several different points in the algorithm. Examples 19.1-1 and 19.1-2 show two programs that do the same thing. One uses coroutines; the other does not. In this example, the program that uses coroutines is actually longer than the one that does not; however, the individual initializing procedures of the coroutines are simpler and more self-contained than the corresponding procedures in the non-coroutine version.

The programs of Examples 19.1-1 and 19.1-2 each accept a stream of characters and must modify them to produce an output stream according to the following rules, applied in the order shown:

1. Every uppercase letter must be replaced by the caret character ("^") followed by the corresponding lowercase letter.

2. Every third pair of characters (that is, every pair of characters numbered 4 and 5 modulo 6, if the first character is numbered 0) must be reversed.

3. After every seventh character, a blank must be inserted into the output stream.

The structure of the algorithm is a series of filters, each accepting and producing a stream of characters. The module NOCO of Example 19.1-1 implements the first filter as the initial procedure; subsequent filters are driven by the input, i.e., do not have top-level loops of their own. By contrast, the bulk of each of the initializing procedures of the coroutines in the module WITHCO of Example 19.1-2 is a loop, as if each initializing procedure were the initial procedure of an independent module. Such an architecture is more appropriate if the order and number of the procedures is subject to change; each can actually be implemented as an independent module, and the filter can be restructured at runtime.

One way to think of a coroutine is as a "generator" of values. For example, suppose a program is to visit each node in an arbitrary graph structure and process the nodes in various ways. The problem can be viewed as consisting of a "node generator" that provides a pointer to the next node to be processed, and any number of "node processors" that process a node in various ways. Suppose that the node generator is non-trivial; i.e., given a pointer to a node, it is not obvious how to find the "next" node without remembering how the previous node was found. The programmer would like to write the node generator once so that it can be used by any node processor.

Coroutines provide a uniformly simple way to structure this task, which is otherwise difficult to phrase in MAINSAIL, since when the node generator finds a node, it may be deeply nested in logic that needs to continue in order to find the next node. If MAINSAIL provided procedures as parameters, then the appropriate node processor could be passed as a procedure to the node generator, so that the node generator could call the anonymous node processor whenever a node was found, passing a pointer to the node as an argument. However, the node processor may itself be non-trivial in that knowing how to process a node depends on how the previous node was processed, so that entering the node processor at the start each time is inappropriate. In this case the node generator and the node processor would like to "call" each other in such a way that each can resume execution where it last left off. This is exactly what is provided by coroutines.

```
BEGIN "noCo"

POINTER(textFile) f;

INTEGER PROCEDURE getChar;
RETURN(cRead(f));
```

Example 19.1-1. A Program without Coroutines (continued)

```
PROCEDURE putChar (INTEGER char);
cWrite(logFile,char);



PROCEDURE doPutChar2 (REPEATABLE INTEGER ch);
BEGIN
OWN INTEGER charNum;
putChar(ch);
IF NOT (charNum .+ 1) MOD 7 THEN putChar(' ');
END;



PROCEDURE doPutChar (INTEGER ch);
BEGIN
OWN INTEGER charNum,prevChar;
IF charNum MOD 6 = 4 THEN prevChar := ch
EF charNum MOD 6 = 5 THEN doPutChar2(ch,prevChar)
EL doPutChar2(ch);
charNum .+ 1;
END;



INITIAL PROCEDURE;
BEGIN
INTEGER ch;
open(f,"Input file: ",input!prompt);
DOB IF ch := getChar < 0 THEN DONE;
    IF isUpperCase(ch) THEN doPutChar('^');
    doPutChar(cvl(ch)) END;
close(f);
END;

END "noCo"
```

Example 19.1-1. A Program without Coroutines (end)

```
BEGIN "withCo"

POINTER(textFile) f;
POINTER($coroutine) co1,co2,co3;
INTEGER chBuf;

INTEGER PROCEDURE getChar;
RETURN(cRead(f));



PROCEDURE putChar (INTEGER char);
cWrite(logFile,char);



PROCEDURE coPutChar (POINTER($coroutine) co;
                     INTEGER char);
# Put char into the buffer for co.
BEGIN
chBuf := char; $resumeCoroutine(co);
END;



INTEGER PROCEDURE coGetChar (POINTER($coroutine) co);
# Read the next character produced by co.
BEGIN
INTEGER ch;
IF chBuf < 0 THEN $resumeCoroutine(co);
ch := chBuf; chBuf := -1;
RETURN(ch);
END;
```

Example 19.1-2. A Program with Coroutines (continued)

```
PROCEDURE co1Proc;
BEGIN
INTEGER ch;
POINTER($coroutine) parent;
parent := $thisCoroutine.$up;
DOB IF ch := getChar < 0 THEN DONE;
    IF isUpperCase(ch) THEN coPutChar(co2,'^');
    coPutChar(co2,cvl(ch)) END;
$resumeCoroutine(parent);
END;



PROCEDURE co2Proc;
BEGIN
INTEGER i,ch;
DOB FOR i := 1 UPTO 4 DO coPutChar(co3,coGetChar(co1));
    ch := coGetChar(co1);
    coPutChar(co3,coGetChar(co1));
    coPutChar(co3,ch) END;
END;



PROCEDURE co3Proc;
BEGIN
INTEGER i;
DOB FOR i := 1 UPTO 7 DO putChar(coGetChar(co2));
    putChar(' ') END;
END;
```

Example 19.1-2. A Program with Coroutines (continued)

```
INITIAL PROCEDURE;
BEGIN
open(f,"Input file: ",input!prompt);
co1 := $createCoroutine(thisDataSection,"co1Proc");
co2 := $createCoroutine(thisDataSection,"co2Proc");
co3 := $createCoroutine(thisDataSection,"co3Proc");
$resumeCoroutine(co1);
$killCoroutine(co1);
$killCoroutine(co2);
$killCoroutine(co3);
close(f);
END;

END "withCo"
```

Example 19.1-2. A Program with Coroutines (end)

19.2. Diagrammatic Example of Coroutines

The module shown in Example 19.2-1 produces the output shown in Example 19.2-2. Example 19.2-3 shows stack diagrams of the various coroutines at the labeled points in Example 19.2-2. There are three stacks, since each of the three coroutines has a separate stack.

At point 1 in Examples 19.2-2 and 19.2-3, the coroutines C1 and C2 have been created, but not initialized, since they have not been resumed for the first time. The procedure stacks of C1 and C2 are therefore empty. At point 2, the initializing procedure, b, of C1 is called when C1 is resumed. b then resumes the parent coroutine, returning control to c in the coroutine MAINSAIL. c then resumes C2, which calls e, creating a stack frame for e in C2. e then resumes C1, which returns control to b. At point 6, b resumes its most recent resumer, $thisCoroutine.$next, returning control to e, which returns to d. d resumes MAINSAIL, which resumes C1, in which b calls a. i being odd, a resumes C2, in which d resumes C1 again, returning control to a. After point 13, a makes a recursive call to itself, creating another stack frame for a in C1. a then calls itself again, then resumes d in C2; d calls e, which resumes C1. At point 18, a makes its final call to itself. The calls to a then unwind, and control returns to b. b then calls e (this is a distinct invocation from the e still active in C2). When e resumes C1 from C1, at point 22, nothing happens; $thisCoroutine.$next is unchanged, so e reports that it was resumed from C2 at point 23. At point 24, e returns to b, and b kills C2, and then commits suicide by calling $resumeCoroutine with the delete bit set. At point 25, only the coroutine MAINSAIL remains.

The use of $thisCoroutine.$next in this example depends on the fact that no exceptions were raised (searching for an exception handler reorders the coroutine $next list) and no coroutines created by the runtime system during the execution of the module of Example 19.2-1. A real application should not depend on $next to determine which coroutine to resume next; it should itself keep track of the order in which coroutines should be scheduled.

```
BEGIN "corout"

POINTER($coroutine) parent,c1,c2;

STRING PROCEDURE resumer;
RETURN(IF $thisCoroutine.$next THEN
           $thisCoroutine.$next.$name
       EL "<no next coroutine>");
```

Example 19.2-1. Three Coroutines (continued)

```
PROCEDURE e;
BEGIN
write(logFile,"In e, resuming c1" & eol);
$resumeCoroutine(c1);
write(logFile,"In e, back from coroutine ",resumer,eol);
END;



PROCEDURE d;
DOB write(logFile,"In d, calling e" & eol);
    e;
    write(logFile,"In d, back from e, resuming parent"
        & eol);
    $resumeCoroutine(parent);
    write(logFile,"In d, back from coroutine ",
        resumer,", resuming c1" & eol);
    $resumeCoroutine(c1);
    write(logFile,"In d, back from coroutine ",
        resumer,eol) END;



PROCEDURE a (INTEGER i);
BEGIN
write(logFile,"In a, i = ",i,eol);
IF i MOD 2 THENB
    write(logFile,"In a, resuming c2" & eol);
    $resumeCoroutine(c2);
    write(logFile,"In a, back from coroutine ",
        resumer,eol) END;
IF i THENB
    write(logFile,"In a, calling a" & eol);
    a(i - 1);
    write(logFile,"In a, back from a" & eol) END;
END;
```

Example 19.2-1. Three Coroutines (continued)

```
PROCEDURE b;
BEGIN
write(logFile,"In b, resuming parent" & eol);
$resumeCoroutine(parent);
write(logFile,"In b, back from coroutine ",
    resumer,eol);
write(logFile,"In b, resuming resumer" & eol);
$resumeCoroutine($thisCoroutine.$next);
write(logFile,"In b, back from coroutine ",
    resumer,eol);
write(logFile,"In b, calling a" & eol);
a(3);
write(logFile,"In b, back from a" & eol);
write(logFile,"In b, calling e" & eol);
e;
write(logFile,"In b, back from e" & eol);
$killCoroutine(c2);
$resumeCoroutine(parent,delete);
END;



INITIAL PROCEDURE c;
BEGIN
parent := $thisCoroutine;
c1 := $createCoroutine(thisDataSection,"b","c1");
c2 := $createCoroutine(thisDataSection,"d","c2");
DOB write(logFile,"In c, resuming c1" & eol);
    $resumeCoroutine(c1);
    write(logFile,"In c, back from coroutine ",
        resumer,eol);
    IF $killedCoroutine(c2) THEN DONE;
    write(logFile,"In c, resuming c2" & eol);
    $resumeCoroutine(c2);
    write(logFile,"In c, back from coroutine ",
        resumer,eol) END
    UNTIL $killedCoroutine(c1);
END;

END "corout"
```

Example 19.2-1. Three Coroutines (end)

```
(1)   In c, resuming c1
(2)   In b, resuming parent
      In c, back from coroutine C1
(3)   In c, resuming c2
(4)   In d, calling e
(5)   In e, resuming c1
      In b, back from coroutine C2
(6)   In b, resuming resumer
(7)   In e, back from coroutine C1
(8)   In d, back from e, resuming parent
      In c, back from coroutine C2
(9)   In c, resuming c1
      In b, back from coroutine MAINSAIL
(10)  In b, calling a
      In a, i = 3
(11)  In a, resuming c2
(12)  In d, back from coroutine C1, resuming c1
      In a, back from coroutine C2
(13)  In a, calling a
      In a, i = 2
(14)  In a, calling a
      In a, i = 1
(15)  In a, resuming c2
      In d, back from coroutine C1
(16)  In d, calling e
(17)  In e, resuming c1
      In a, back from coroutine C2
(18)  In a, calling a
(19)  In a, i = 0
(20)  In a, back from a
      In a, back from a
      In a, back from a
      In b, back from a
(21)  In b, calling e
(22)  In e, resuming c1
(23)  In e, back from coroutine C2
(24)  In b, back from e
(25)  In c, back from coroutine <no next coroutine>
```

Example 19.2-2. Output from Example 19.2-1

```
              Coroutine            Coroutine          Coroutine
               MAINSAIL                C1                 C2
          (parent coroutine)

         +------+
(1)   -> |  c   |
         +------+          +------+          +------+

         +------+          +------+
(2)      |  c   |       -> |  b   |
         +------+          +------+          +------+

         +------+          +------+
(3)   -> |  c   |          |  b   |
         +------+          +------+          +------+

         +------+          +------+          +------+
(4)      |  c   |          |  b   |       -> |  d   |
         +------+          +------+          +------+

                                             +------+
                                          -> |  e   |
(5)      +------+          +------+          +------+
         |  c   |          |  b   |          |  d   |
         +------+          +------+          +------+

                                             +------+
                                             |  e   |
(6)      +------+          +------+          +------+
         |  c   |       -> |  b   |          |  d   |
         +------+          +------+          +------+

                                             +------+
                                          -> |  e   |
(7)      +------+          +------+          +------+
         |  c   |          |  b   |          |  d   |
         +------+          +------+          +------+

         +------+          +------+          +------+
(8)      |  c   |          |  b   |       -> |  d   |
         +------+          +------+          +------+
```

Example 19.2-3. Explanation of Example 19.2-1 with Stack Diagrams (continued)

```
              +------+          +------+          +------+
(9)       ->  |  c   |          |  b   |          |  d   |
              +------+          +------+          +------+

              +------+          +------+          +------+
(10)          |  c   |      ->  |  b   |          |  d   |
              +------+          +------+          +------+

                                +------+
                            ->  |  a   |
(11)          +------+          +------+          +------+
              |  c   |          |  b   |          |  d   |
              +------+          +------+          +------+

                                +------+
                                |  a   |
(12)          +------+          +------+          +------+
              |  c   |          |  b   |      ->  |  d   |
              +------+          +------+          +------+

                                +------+
                            ->  |  a   |
(13)          +------+          +------+          +------+
              |  c   |          |  b   |          |  d   |
              +------+          +------+          +------+

                                +------+
                            ->  |  a   |
                                +------+
(14)                            |  a   |
              +------+          +------+          +------+
              |  c   |          |  b   |          |  d   |
              +------+          +------+          +------+
```

Example 19.2-3. Explanation of Example 19.2-1 with Stack Diagrams (continued)

```
                                +------+
                            ->  |  a   |
                                +------+
                                |  a   |
(15)                            +------+
                                |  a   |
           +------+             +------+             +------+
           |  c   |             |  b   |             |  d   |
           +------+             +------+             +------+

                                +------+
                                |  a   |
                                +------+
                                |  a   |
(16)                            +------+
                                |  a   |
           +------+             +------+             +------+
           |  c   |             |  b   |         ->  |  d   |
           +------+             +------+             +------+

                                +------+
                                |  a   |
                                +------+
                                |  a   |
(17)                            +------+             +------+
                                |  a   |         ->  |  e   |
           +------+             +------+             +------+
           |  c   |             |  b   |             |  d   |
           +------+             +------+             +------+

                                +------+
                            ->  |  a   |
                                +------+
                                |  a   |
(18)                            +------+             +------+
                                |  a   |             |  e   |
           +------+             +------+             +------+
           |  c   |             |  b   |             |  d   |
           +------+             +------+             +------+
```

Example 19.2-3. Explanation of Example 19.2-1 with Stack Diagrams (continued)

```
                                +------+
                             -> |  a   |
                                +------+
                                |  a   |
                                +------+
                                |  a   |
 (19)                           +------+                +------+
                                |  a   |                |  e   |
               +------+         +------+                +------+
               |  c   |         |  b   |                |  d   |
               +------+         +------+                +------+

                                +------+
                             -> |  a   |
                                +------+
                                |  a   |
 (20)                           +------+                +------+
                                |  a   |                |  e   |
               +------+         +------+                +------+
               |  c   |         |  b   |                |  d   |
               +------+         +------+                +------+

                                                        +------+
                                                        |  e   |
 (21)          +------+         +------+                +------+
               |  c   |      -> |  b   |                |  d   |
               +------+         +------+                +------+

                                +------+                +------+
                             -> |  e   |                |  e   |
 (22)          +------+         +------+                +------+
               |  c   |         |  b   |                |  d   |
               +------+         +------+                +------+

                                +------+                +------+
                             -> |  e   |                |  e   |
 (23)          +------+         +------+                +------+
               |  c   |         |  b   |                |  d   |
               +------+         +------+                +------+
```

Example 19.2-3. Explanation of Example 19.2-1 with Stack Diagrams (continued)

```
                                                      +------+
                                                      |  e   |
(24)      +------+             +------+               +------+
          |  c   |          -> |  b   |               |  d   |
          +------+             +------+               +------+

          +------+
(25)   -> |  c   |
          +------+
```

Example 19.2-3. Explanation of Example 19.2-1 with Stack Diagrams (end)

19.3. Primitive Scheduler Example

If applications cooperate, they can be scheduled in such a way that several can run (apparently) simultaneously. Each application can be run in a separate coroutine, and a special coroutine, the scheduler, switches among them whenever the applications allow the scheduler to take control.

In Example 19.3-1, the original (scheduling) coroutine of the module SCHED takes control whenever any of its applications calls any of the module's interface procedures. Control is then given to another application, after the requested task, if any, is performed. The applications perform all terminal input and output through readInput and writeOutput, instead of reading or writing directly from cmdFile or to logFile, in order to give the scheduler a chance to switch between applications.

Some applications for the primitive scheduler are shown in Examples 19.3-2, 19.3-3, and 19.3-4. ADDNUM adds a series of numbers input from cmdFile. BKWRD2 reverses strings read from cmdFile. SMSQRT prints out the square roots of the sums of the squares from one to ten.

```
BEGIN "addNum"

MODULE sched (
    STRING PROCEDURE readInput;
    PROCEDURE writeOutput (REPEATABLE STRING s);
    PROCEDURE poll;
);

INITIAL PROCEDURE;
BEGIN
INTEGER i;
STRING s;
i := 0;
DOB writeOutput("Type a number (<eol> to quit): ");
    IF NOT s := readInput THEN DONE;
    writeOutput("Sum so far = ",cvs(i .+ cvi(s)),eol) END;
END;

END "addNum"
```

Example 19.3-2. ADDNUM Scheduler Application

```
BEGIN "sched"

MODULE sched (
    STRING PROCEDURE readInput;
    PROCEDURE writeOutput (REPEATABLE STRING s);
    PROCEDURE poll;
);

CLASS coroutLstCls (
    POINTER($coroutine) co;
    STRING applicationName,inBuffer,outBuffer;
    POINTER dataSec;
    BOOLEAN wantsInput;
    POINTER(coroutLstCls) next,prev; # circular list
);

POINTER(coroutLstCls) coroutLst,curCorout;
POINTER($coroutine) parent;

STRING PROCEDURE readInput;
# This is called from the application.
BEGIN
STRING s;
curCorout.wantsInput := TRUE;
$resumeCoroutine(parent);
s := curCorout.inBuffer; curCorout.inBuffer := "";
RETURN(s);
END;



PROCEDURE writeOutput (REPEATABLE STRING s);
# This is called from the application.
BEGIN
curCorout.outBuffer .& s;
$resumeCoroutine(parent);
END;
```

Example 19.3-1. Primitive Scheduler Module (continued)

```
PROCEDURE poll;
# This is called from the application.
$resumeCoroutine(parent);



PROCEDURE runApplication;
# This is each application coroutine's initializing
# procedure.
BEGIN
$HANDLE
    IF NOT curCorout.dataSec :=
        new(curCorout.applicationName,errorOK) THEN
        write(logFile,"Couldn't find module ",
            curCorout.applicationName,eol)
$WITH
    write(logFile,
        "Aborted application ",curCorout.applicationName,
        " because of exception ",$exceptionName,eol);
IF curCorout.dataSec THEN dispose(curCorout.dataSec);
$resumeCoroutine(parent,delete);
    # when done, coroutine dies
END;



STRING PROCEDURE doReadInput;
# A more sophisticated scheduler would wait until it sees
# some indication of the coroutine to which the user
# wants to talk.  In a window system, might require the
# user to move the mouse into the current window, for
# example.  This is clearly a primitive and not very
# user-friendly version.
BEGIN
STRING s;
read(cmdFile,s); RETURN(s);
END;
```

Example 19.3-1. Primitive Scheduler Module (continued)

```
PROCEDURE doWriteOutput;
# A more sophisticated scheduler would display the output
# in such a way that it is apparent what application is
# talking.
write(logFile,curCorout.outBuffer);
```

Example 19.3-1. Primitive Scheduler Module (continued)

```
INITIAL PROCEDURE;
BEGIN
STRING s;
POINTER(coroutLstCls) p,endP;
parent := $thisCoroutine;
DOB write(logFile,
        "Application to run (<eol> to end list): ");
    read(cmdFile,s); IF NOT s THEN DONE;
    p := new(coroutLstCls);
    p.co := $createCoroutine
        (thisDataSection,"runApplication");
    p.applicationName := s;
    IF coroutLst THENB
        (p.next := coroutLst).prev := p;
        (endP.next := p).prev := endP;
        coroutLst := p END
    EL  coroutLst := p.next := p.prev := endP := p END;
curCorout := coroutLst;
# Strictly round-robin scheduling.  Not very smart.
DOB $resumeCoroutine(curCorout.co);
    # Assume we are resumed from the same coroutine (i.e.,
    # curCorout.co).  A more general scheduler would have
    # to handle the case where the application switched
    # coroutines.
    IF curCorout.outBuffer THENB
        doWriteOutput; curCorout.outBuffer := "" END;
    IF curCorout.wantsInput THENB
        curCorout.inBuffer .& doReadInput;
        curCorout.wantsInput := FALSE END;
    IF $killedCoroutine(curCorout.co) THENB
        IF curCorout.prev = curCorout THEN DONE;
            # no more coroutines if killing last one
        curCorout.prev.next := curCorout.next;
        curCorout.next.prev := curCorout.prev END;
    curCorout := curCorout.next END;
END;

END "sched"
```

Example 19.3-1. Primitive Scheduler Module (end)

The scheduler as written has several substantial drawbacks:

```
BEGIN "bkwrd2"

MODULE sched (
    STRING PROCEDURE readInput;
    PROCEDURE writeOutput (REPEATABLE STRING s);
    PROCEDURE poll;
);

INITIAL PROCEDURE;
BEGIN
STRING s,ss;
DOB writeOutput("Type in a string (<eol> to stop): ");
    IF NOT s := readInput THEN DONE;
    ss := "";
    WHILE s DO cWrite(ss,rcRead(s));
    writeOutput("Backwards string is ",ss,eol) END;
END;

END "bkwrd2"
```

Example 19.3-3. BKWRD2 Scheduler Application

1. The input and output of the different applications are all intermingled (see Example 19.3-5). It is very hard to tell which application has printed a given output or is currently requesting input.

2. The scheduler blocks on input from cmdFile. It should really test to see whether there is any cmdFile input waiting, and if not, then run an application that does not currently need any input.

3. The scheduler applications must be written in a particular format; they do not work as stand-alone MAINSAIL programs.

4. The scheduler does not make any provision for exceptions in the applications (see Section 19.5).

An improved scheduler is left as an exercise for the reader.

```
BEGIN "smSqrt"

MODULE sched (
    STRING PROCEDURE readInput;
    PROCEDURE writeOutput (REPEATABLE STRING s);
    PROCEDURE poll;
);

INITIAL PROCEDURE;
BEGIN
REAL squareSum,r;
squareSum := r := 0.0;
WHILE r < 9.5 DOB
    r .+ 1.0;
    poll;
    squareSum .+ r * r;
    writeOutput("Square root of sum of squares (up to ",
        cvs(r),") is ",cvs(sqrt(squareSum)),eol) END;
END;

END "smSqrt"
```

Example 19.3-4. SMSQRT Scheduler Application

```
Application to run (<eol> to end list): addnum<eol>
Application to run (<eol> to end list): bkwrd2<eol>
Application to run (<eol> to end list): smsqrt<eol>
Application to run (<eol> to end list): <eol>
Type in a string (<eol> to stop): Type a number (<eol> to
quit): Square root of sum of squares (up to Hello!<eol>
43<eol>
1Backwards string is Sum so far = ) is !olleH431


Type in a string (<eol> to stop): Type a number (<eol> to
quit): Time is money.<eol>
62<eol>
Square root of sum of squares (up to Backwards string is S
um so far = 2.yenom si emiT105) is

2.23607Type in a string (<eol> to stop): Type a number (<e
ol> to quit): <eol>
<eol>

Square root of sum of squares (up to 3) is 3.74166
Square root of sum of squares (up to 4) is 5.47723
Square root of sum of squares (up to 5) is 7.4162
Square root of sum of squares (up to 6) is 9.53939
Square root of sum of squares (up to 7) is 11.8322
Square root of sum of squares (up to 8) is 14.2829
Square root of sum of squares (up to 9) is 16.8819
Square root of sum of squares (up to 10) is 19.6214
```

Example 19.3-5. Sample Execution of SCHED

19.4. Ecological Simulation Example

In this section, a special-purpose scheduler is used to alternate among a variety of coroutines, each of which represents a "critter" (an organism in a simple ecological simulation). The program is designed so that the scheduling and most utility procedures are provided by a single executive module, DARWIN. DARWIN does as much work as possible, so that the individual critter modules are as simple to write as possible. Each critter is a module that is run in its own coroutine. Scheduling is done whenever a critter calls into DARWIN; the critters themselves do not contain any explicit calls to $resumeCoroutine or other coroutine procedures.

Coroutines greatly simplify the writing of critters; without coroutines, each critter would have to be written as an interface procedure that is called by the executive and returns some sort of code indicating the operation it wanted the executive to perform. To pass back the result of an operation, the executive would have to use the same data structure for every possible operation and pass it as a parameter to the critter's interface procedure. Context in the critter would have to be maintained with own variables instead of local variables, since the values of local variables would be lost between calls. Altogether, critters would look far less like "normal programs" than they do in this coroutine-based example.

Example 19.4-1 shows the CRTHDR module that makes the intmod (see Section 20.3) shared by the executive and all critter modules. The DARWIN interface procedures are the utilities to be used by all critters; they are described by comments in the CRTHDR and DARWIN source files. The DARWIN source file is shown in Example 19.4-2. Sample critters are shown in Examples 19.4-3 and 19.4-4; they are not very sophisticated, and better critters could certainly be designed.

DARWIN is the module to run initially. It prompts for the name of a debugging file ("NUL>" unless you want to see a log of all the operations undertaken by DARWIN), the size of the display, a display module, and a list of critter modules to include in the simulation (give the name of the same module more than once if you want to start off with more than one instance of it). It then uses the MAINSAIL display module to blank the screen. Each critter is display as a rectangle of four characters. The upper left character is the name of the critter; it is a single letter, uppercase if the critter has lots of spare energy, lowercase if it has only a little spare energy. The upper right character is usually ".", but it is "<" if the critter is attacking (think of it as an open mouth), and ">" if the critter is under attack. The lower left and lower right characters are the number of teeth and shell units the creature has, respectively.

```
BEGIN "crtHdr" # Header file for Darwinian critters

$DIRECTIVE "NOOUTPUT";
SAVEON;

DEFINE # possible what.status values
    emptyCell           =       1,
    cellOutsideWorld    =       2,
    critterCell         =       3,
```

Example 19.4-1. CRTHDR, Ecological Simulation Intmod (continued)

```
# game parameters (costs are in energy units and are
# deducted from the critter's spareEnergy field)
    energyForLegs        =        12,
        # Energy required to grow legs
        # (No more than one set of legs allowed)
    energyForLeaves      =        9,
        # Energy required to grow leaves
        # (No more than one set of leaves allowed)
    energyPerTooth       =        10,
        # Energy required to grow teeth
    maxTeeth             =        9,
        # No more than maxTeeth teeth allowed
    energyPerShell       =        10,
        # Energy required to grow a unit of shell
    maxShell             =        9,
        # No more than maxShell units of shell allowed
    birthCost            =        2,
        # Cost to reproduce
    halfBirthEnergy      =        20,
        # Energy which, when given to an offspring,
        # gives it a 50-50 chance of survival
    costs1ToMove         =        10,
        # Costs 1 unit to move a critter with this many
        # units of totalEnergy + teethAmount + shellAmount
    attackCost           =        1,
        # Cost per attack on an adjacent critter
    solarEnergyPerTurn   =        1,
        # Solar energy received each turn if has leaves
    maxEnergy            =        255,
        # Can never accumulate more energy than this
    approxTurnsPerUnitOfEnergyWhileResting
                         =        150,
        # Approximately this many turns, lose one unit
        # of energy if not doing anything else

    attackExcpt          =        "Attack by critter";
        # Exception raised if under attack by another
        # critter to give chance to defend itself;
        # offset to critter in
        # $exceptionStringArg1 = cvs(x offset),
        # $exceptionStringArg2 = cvs(y offset)
```

Example 19.4-1. CRTHDR, Ecological Simulation Intmod (continued)

```
CLASS what ( # Describes what is contained in a cell

    INTEGER status;

# Remaining fields valid only if status = critterCell:

    INTEGER
        critterName,     # Critter's letter
        spareEnergy,     # How much spare energy critter
                         # has
        totalEnergy,     # How much energy will provide if
                         # eaten
        teethAmount,     # How many teeth
        shellAmount;     # How much shell

    BOOLEAN
        hasLeaves,       # If has leaves
        hasLegs;         # If has legs

);

MODULE darwin (

    # Interfaces are utilities to be used by critters;
    # most procedures cause rescheduling.  See also
    # comments in DARWIN module itself

    BOOLEAN PROCEDURE addLegs;
        # Legs are required if critter is to move

    BOOLEAN PROCEDURE addLeaves;
        # Leaves allow critter to get energy from sun

    BOOLEAN PROCEDURE addTeeth;
        # Teeth provide advantage in attacking

    BOOLEAN PROCEDURE addShell;
        # Shell provides protection from attack
```

Example 19.4-1. CRTHDR, Ecological Simulation Intmod (continued)

```
    BOOLEAN PROCEDURE reproduce
        (INTEGER xOffset,yOffset,initEnergy;
         OPTIONAL STRING geneticString;
         PRODUCES OPTIONAL BOOLEAN childSurvived);
        # Reproduce at xOffset,yOffset, giving the
        # child an initial value of initEnergy and
        # a "genetic string".  The more initEnergy,
        # the greater the chance the child will
        # survive

    STRING PROCEDURE myGeneticString;
        # Can be used for simulating evolution

    BOOLEAN PROCEDURE move (INTEGER xOffset,yOffset);
        # Returns true iff move legal; offsets must be
        # in range -1 to 1 (i.e., adjacent)

    PROCEDURE lookAround
        (MODIFIES POINTER(what) ARRAY(-2 TO 2,-2 TO 2)
            whatIsAround);
        # Can see the 5 x 5 area immediately surrounding

    BOOLEAN PROCEDURE attack (INTEGER xOffset,yOffset);
        # Can attack any adjacent critter

    POINTER($ranCls) ranPtr;
        # For use in ran

    POINTER(textFile) dbgf;
        # Debugging output file, if you want to see what
        # is going on

    STRING PROCEDURE cName (OPTIONAL POINTER(eWhat) p);
        # Returns the name of current critter for use
        # in debugging output.  p should not be specified
        # when called from a critter.

);
```

Example 19.4-1. CRTHDR, Ecological Simulation Intmod (continued)

```
INTEGER PROCEDURE ran (INTEGER loBound,hiBound);
# To be used if critter wants a random number to help it
# make decisions
RETURN(cvi(ranPtr.$rand MOD cvli(hiBound - loBound + 1))
        + loBound);

END "crtHdr"
```

Example 19.4-1. CRTHDR, Ecological Simulation Intmod (end)

```
BEGIN "darwin"

RESTOREFROM "crtHdr";

REDEFINE $scanName = "dpyHdr"; SOURCEFILE "msl:syslib";

INTEGER xMax,yMax;

CLASS(what) eWhat ( # Extended what class

    INTEGER atX,atY; # Current coordinates

    STRING modName,geneticString;

    POINTER modPtr;

    POINTER($coroutine) coroutine;

    POINTER(eWhat) prevCritter,nextCritter,
        # Circularly linked list
        attackee;

    BOOLEAN attacking,attacked;

);

LONG INTEGER critNum;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
POINTER(eWhat) curCritter;

POINTER(eWhat) ARRAY (0 TO *,0 TO *) world;

POINTER($coroutine) mainCo;

MODULE(dpyCls) dpy; # required by display module

# Debugging stuff

STRING PROCEDURE cName (OPTIONAL POINTER(eWhat) p);
BEGIN
STRING s;
IF NOT p THEN p := curCritter;
s := "";
cWrite(s,IF p.spareEnergy > 10 THEN p.critterName
         EL cvl(p.critterName),
         IF p.attacking THEN '<'
         EF p.attacked THEN '>'
         EL '.',
         p.teethAmount + '0',
         p.shellAmount + '0');
RETURN(s & " = " & (IF p.coroutine THEN p.coroutine.$name
                    EL "<no coroutine>"));
END;



# Display procedures

FORWARD PROCEDURE showAll;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
PROCEDURE displayMessage (STRING s);
BEGIN
INTEGER ro,ch;
STRING ss;
clearScreen;
ro := 0;
WHILE s DOB
    setCursorOnScreen(ro,0); read(s,ss);
    overStrikeChars(ss); ro .+ 1 END;
setCursorOnScreen(lastRowOfScreen,0);
overStrikeChars(
    "--- Hit D to debug, any other key to continue ---");
IF ch := dpycRead = 'd' OR ch = 'D' THEN $debugExec;
showAll;
END;



PROCEDURE showAt (INTEGER x,y);
BEGIN
POINTER(eWhat) p;
p := world[x,y];
setCursorOnScreen(2 * y,2 * x);
IF p.status = emptyCell THEN overstrikeChars("  ")
EB  overstrikeChar(
        IF p.spareEnergy > 10 THEN p.critterName
        EL cvl(p.critterName));
    overstrikeChar(IF p.attacking THEN '<'
                           EF p.attacked THEN '>'
                           EL '.') END;
setCursorOnScreen(2 * y + 1,2 * x);
IF p.status = emptyCell THEN overstrikeChars("  ")
EB  overstrikeChar(p.teethAmount + '0');
    overstrikeChar(p.shellAmount + '0') END;
dpyInfo(dumpScreenBuffer);
END;



PROCEDURE showCur;
showAt(curCritter.atX,curCritter.atY);
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
PROCEDURE showAll;
BEGIN
INTEGER x,y;
clearScreen;
FOR x := 0 UPTO xMax DO FOR y := 0 UPTO yMax DO
        showAt(x,y);
END;



# Utility procedures

INLINE PROCEDURE killCritter (POINTER(eWhat) p);
p.spareEnergy := -10000; # Mark as moribund



PROCEDURE checkCo;
IF $thisCoroutine NEQ curCritter.coroutine THENB
    displayMessage
        ("$thisCoroutine NEQ curCritter.coroutine!");
    killCritter(curCritter) END;



BOOLEAN PROCEDURE spendOk
     (INTEGER energy;
      OPTIONAL BOOLEAN addHalfToTotal);
BEGIN
IF curCritter.spareEnergy < energy THEN RETURN(FALSE);
curCritter.spareEnergy .- energy;
curCritter.totalEnergy .- energy;
showCur;
IF addHalfToTotal THEN # some expenditures are nutritive
    curCritter.totalEnergy .+ energy DIV 2;
RETURN(TRUE);
END;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
BOOLEAN PROCEDURE addLegs;
# Add legs to the creature, if it has enough spare energy
# and doesn't already have legs.  Otherwise, don't change
# its energy and return false; return true iff legs added.
BEGIN
BOOLEAN bo;
checkCo;
bo := FALSE;
IF NOT curCritter.hasLegs AND spendOk(energyForLegs,TRUE)
    THENB
    write(dbgf,"Doing addLegs for ",cName,eol);
    curCritter.hasLegs := bo := TRUE END;
showCur; $resumeCoroutine(mainCo);
RETURN(bo);
END;



BOOLEAN PROCEDURE addLeaves;
# Like addLegs, true iff successful.
BEGIN
BOOLEAN bo;
checkCo;
bo := FALSE;
IF NOT curCritter.hasLeaves AND
    spendOk(energyForLeaves,TRUE) THEN
    write(dbgf,"Doing addLeaves for ",cName,eol);
    curCritter.hasLeaves := bo := TRUE;
showCur; $resumeCoroutine(mainCo);
RETURN(bo);
END;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
BOOLEAN PROCEDURE addTeeth;
# Like addLegs, true iff successful.
BEGIN
BOOLEAN bo;
checkCo;
bo := FALSE;
IF curCritter.teethAmount < maxTeeth AND
    spendOk(energyPerTooth) THENB
    write(dbgf,"Doing addTeeth for ",cName,eol);
    curCritter.teethAmount .+ 1; bo := TRUE END;
showCur; $resumeCoroutine(mainCo);
RETURN(bo);
END;



BOOLEAN PROCEDURE addShell;
# Like addLegs, true iff successful.
BEGIN
BOOLEAN bo;
checkCo;
bo := FALSE;
IF curCritter.shellAmount < maxShell AND
    spendOk(energyPerShell) THENB
    write(dbgf,"Doing addShell for ",cName,eol);
    curCritter.shellAmount .+ 1; bo := TRUE END;
showCur; $resumeCoroutine(mainCo);
RETURN(bo);
END;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
BOOLEAN PROCEDURE reproduce
    (INTEGER xOffset,yOffset,initEnergy;
     OPTIONAL STRING geneticString;
     PRODUCES OPTIONAL BOOLEAN childSurvived);
# Return true if spent the energy for reproduction; the
# critter must have spare energy at least equal to
# initEnergy + birthEnergy.  The child is created with
# initEnergy.  The child may or may not survive its first
# turn:  child survival is determined at random; the more
# initial energy, the better chance of suriving.  Survival
# rate is 1/2 at initEnergy = halfBirthEnergy.
# The genetic string is a special message passed from
# parent to child and which the child may examine at any
# time by calling myGeneticString.
# childSurvived is true iff child survived.
# Child must be created adjacent to parent.
BEGIN
BOOLEAN bo;
INTEGER x,y;
REAL r;
POINTER(eWhat) p;
checkCo;
bo := FALSE;
IF initEnergy > 0 AND
    0 LEQ x := curCritter.atX + xOffset LEQ xMax AND
    0 LEQ y := curCritter.atY + yOffset LEQ yMax AND
    world[x,y].status = emptyCell AND
    spendOk(initEnergy + birthCost) THENB
    write(dbgf,"Doing reproduce for ",cName," to ",
        xOffset,",",yOffset," with ",initEnergy,
        "; gene string = """,geneticString,"""",eol);
    r := (cvr(initEnergy) / cvr(halfBirthEnergy)) MIN 10.;
    r := 2. ^ r; # r is big for high initial energies
    IF ran(1,cvi(r)) = 1 AND
        # Even if small energy, give 1/100 chance
        ran(0,99) THENB
        write(dbgf,"Child did not survive" & eol);
        childSurvived := FALSE END
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
    EB  childSurvived := TRUE;
        write(dbgf,"Child survived!" & eol);
        p := world[x,y]; p.status := critterCell;
        p.critterName := curCritter.critterName;
        p.spareEnergy := p.totalEnergy :=
            initEnergy MIN maxEnergy;
        p.teethAmount := p.shellAmount := 0;
        p.hasLeaves := p.hasLegs := FALSE;
        p.atX := x; p.atY := y;
        p.modName := curCritter.modName;
        p.geneticString := geneticString;
        p.modPtr := NULLPOINTER;
        p.coroutine := NULLPOINTER;
        p.attacking := p.attacked := FALSE;
        p.prevCritter := curCritter;
        p.nextCritter := curCritter.nextCritter;
        curCritter.nextCritter.prevCritter := p;
        curCritter.nextCritter := p; bo := TRUE;
        showAt(x,y) END END;
$resumeCoroutine(mainCo);
RETURN(bo);
END;



STRING PROCEDURE myGeneticString;
# Return initial message from parent.
BEGIN
STRING s;
checkCo;
s := curCritter.geneticString;
$resumeCoroutine(mainCo);
RETURN(s);
END;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
BOOLEAN PROCEDURE move (INTEGER xOffset,yOffset);
# Must move to an adjacent square; i.e.,
# abs(xOffset) LEQ 1 and abs(yOffset) LEQ 1.
# Cost is (totalEnergy + teeth + shell) DIV costs1ToMove
# plus one (the creature's weight takes energy to move
# around).
BEGIN
BOOLEAN bo;
INTEGER oX,oY,x,y;
POINTER(eWhat) p;
checkCo;
bo := FALSE;
IF (xOffset OR yOffset) AND abs(xOffset) LEQ 1 AND
    abs(yOffset) LEQ 1 AND
    0 LEQ x := (oX := curCritter.atX) + xOffset LEQ xMax
        AND
    0 LEQ y := (oY := curCritter.atY) + yOffset LEQ yMax
    AND world[x,y].status = emptyCell AND
    spendOk(
        (curCritter.totalEnergy + curCritter.teethAmount +
            curCritter.shellAmount) DIV costs1ToMove + 1)
    THENB
    write(dbgf,"Doing move for ",cName," to ",
        xOffset,",",yOffset,eol);
    # Exchange empty cell and critter's cell
    p := world[x,y];
    world[x,y] := curCritter;
    curCritter.atX := x; curCritter.atY := y;
    world[oX,oY] := p;
    p.atX := oX; p.atY := oY;
    bo := TRUE;
    showAt(x,y); showAt(oX,oY) END;
$resumeCoroutine(mainCo);
RETURN(bo);
END;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
PROCEDURE lookAround
    (MODIFIES POINTER(what) ARRAY(-2 TO 2,-2 TO 2)
        whatIsAround);
# Allocate the array and its elements only if necessary,
# and fill in with description of the 5 x 5 area
# surrounding.  what.status is cellOutsideWorld for cells
# off the board.  Note that records returned in the array
# are what records, note the extended eWhat records used
# internally by this module to keep track of additional
# information about the cells.  Thus the critter does
# not have access to the internal form of the state info.
BEGIN
INTEGER i,j,x,y;
OWN POINTER(what) outside;
checkCo;
$resumeCoroutine(mainCo);
IF NOT outside THENB
    outside := new(what);
    outside.status := cellOutsideWorld END;
IF NOT whatIsAround THEN new(whatIsAround);
FOR i := -2 UPTO 2 DO FOR j := -2 UPTO 2 DOB
        IF NOT whatIsAround[i,j] THEN
            whatIsAround[i,j] := new(what);
        IF 0 LEQ x := curCritter.atX + i LEQ xMax AND
            0 LEQ y := curCritter.atY + j LEQ yMax THEN
            copy(world[x,y],whatIsAround[i,j])
        EL  whatIsAround[i,j] := outside END;
END;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
BOOLEAN PROCEDURE attack (INTEGER xOffset,yOffset);
# Return true iff managed to eat the critter at the
# specified offset from current critter.  If eaten,
# the critter disappears and its cell becomes empty,
# and the eater acquires its totalEnergy.  Note that the
# attack may consume the attacker's energy even if
# unsuccessful.  The attack never has effect against
# a shelled creature unless the attacker has teeth, but
# is always successful against an unshelled creature.
# An attack against a shelled creature serves only to
# weaken the shell; enough attacks must be made to
# eliminate the shell before the creature can be eaten.
BEGIN
BOOLEAN bo,bo2;
INTEGER x,y,oX,oY;
BITS xx;
POINTER(eWhat) p;
checkCo;
bo := bo2 := FALSE;
IF (xOffset OR yOffset) AND abs(xOffset) LEQ 1 AND
    abs(yOffset) LEQ 1 AND
    0 LEQ x := (oX := curCritter.atX) + xOffset LEQ xMax
        AND
    0 LEQ y := (oY := curCritter.atY) + yOffset LEQ yMax
    AND (p := world[x,y]).status = critterCell AND
    spendOK(attackCost) THENB
    write(dbgf,"Doing attack for ",cName," to ",
        xOffset,",",yOffset," (",cName(p),")",eol,
        "  Teeth = ",curCritter.teethAmount,
           "; shell = ",p.shellAmount,": ");
    bo2 := TRUE;
    curCritter.attacking := p.attacked := TRUE;
    curCritter.attackee := p;
    showCur; showAt(x,y);
    $raise(attackExcpt,cvs(oX - x),cvs(oY - y),
        NULLPOINTER,$cannotFallOut!$returnIfNoHandler,xx,
        p.coroutine);
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
    IF p.shellAmount AND
        curCritter.teethAmount GEQ
        ran(1,2 * p.shellAmount) THENB
        p.shellAmount .- 1; # managed to injure
        write(dbgf,"Injured, shell reduced to ",
            p.shellAmount,eol) END
    EF NOT p.shellAmount THENB
        # Defenseless creature.  Yum yum!
        write(dbgf,"Eaten!" & eol);
        curCritter.spareEnergy .+ p.totalEnergy;
        curCritter.totalEnergy .+ p.totalEnergy;
        curCritter.spareEnergy .MIN maxEnergy;
        curCritter.totalEnergy .MIN maxEnergy;
        killCritter(p); bo := TRUE END
    EL  write(dbgf,"Uninjured." & eol);
    showCur; showAt(x,y) END;
$resumeCoroutine(mainCo);
IF bo2 THENB
    curCritter.attacking := p.attacked := FALSE;
    curCritter.attackee := NULLPOINTER;
    showCur; showAt(x,y) END;
RETURN(bo);
END;



PROCEDURE startCurCritter;
BEGIN
curCritter.modPtr := new(curCritter.modName);
$resumeCoroutine(mainCo,delete);
END;



INITIAL PROCEDURE;
BEGIN
INTEGER x,y,critLetter;
STRING s;
POINTER(eWhat) p;

mainCo := $thisCoroutine;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
open(dbgf,"Debugging file: ",create!prompt!output);

write(logFile,
    "Number of columns (<eol> for full screen width): ");
read(cmdFile,s);
IF NOT xMax := cvi(s) THEN xMax := $maxInteger;
write(logFile,
    "Number of rows (<eol> for full screen height): ");
read(cmdFile,s);
IF NOT yMax := cvi(s) THEN yMax := $maxInteger;

write(logFile,"Display module: "); read(cmdFile,s);
setModName("dpy",s);

initializeTerminal;
xMax .MIN ((lastColOfScreen - 1) DIV 2);
yMax .MIN ((lastRowOfScreen -  1) DIV 2);
deInitializeTerminal;
new(world,0,xMax,0,yMax);

ranPtr := new($ranMod);
ranPtr.$initRand($date,$time);

critLetter := 'A';
DOB write(logFile,
        "Next critter to add to world (<eol> to stop): ");
    read(cmdFile,s); IF NOT s := cvu(s) THEN DONE;
    IF NOT $canFindModule(s) THENB
        write(logFile,"Couldn't find module ",s,eol);
        CONTINUE END;
    DOB x := ran(0,xMax); y := ran(0,yMax) END
        UNTIL NOT world[x,y];

    world[x,y] := p := new(eWhat);
    p.status := critterCell;
    p.critterName := critLetter;

    write(logFile,"The creature in module ",s,
        " will be displayed with the letter ");
    cWrite(logFile,critLetter);
    write(logFile,eol);
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
    critLetter := nextAlpha(critLetter);
    p.spareEnergy := p.totalEnergy := 18;
    p.teethAmount := p.shellAmount := 0;
    p.hasLeaves := p.hasLegs := FALSE;
    p.atX := x; p.atY := y;
    p.modName := s; p.geneticString := "";
    # p.modPtr and .coroutine to be initialized when run
    p.attacking := p.attacked := FALSE;
    IF curCritter THENB
        p.prevCritter := curCritter.prevCritter;
        p.nextCritter := curCritter;
        curCritter.prevCritter.nextCritter := p;
        curCritter.prevCritter := p END
    EB  curCritter := p;
        curCritter.prevCritter :=
            curCritter.nextCritter := p END END;

# Finish initializing the world
FOR x := 0 UPTO xMax DO FOR y := 0 UPTO yMax DO
        IF NOT world[x,y] THEN
            (world[x,y] := new(eWhat)).status :=
                emptyCell;

initializeTerminal; showAll;

critNum := 0L;
# Now run the critters until none left
DOB $HANDLEB

        write(dbgf,"Running ",cName,
            "; spare energy = ",curCritter.spareEnergy,
            eol);

        IF ran(1,approxTurnsPerUnitOfEnergyWhileResting)
            = 1 THENB
            # Just sitting around requires some energy
            curCritter.spareEnergy .- 1;
            curCritter.totalEnergy .- 1;

            write(dbgf,"Diminishing energy" & eol);

            showCur END;
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
# The critter is dead if its energy is negative
IF curCritter.spareEnergy < 0 THENB

    write(dbgf,"Killing curCritter" & eol);

    IF NOT $killedCoroutine(curCritter.coroutine)
        THEN $killCoroutine(curCritter.coroutine);
    dispose(curCritter.modPtr);
    IF p := curCritter.attackee THEN
        p.attacked := FALSE;
    IF p := curCritter.nextCritter = curCritter
        THEN DONE;
    p.prevCritter := curCritter.prevCritter;
    p.prevCritter.nextCritter := p;
    (world[curCritter.atX,curCritter.atY] :=
        new(eWhat)).status := emptyCell;
    showCur;
    # Don't dispose curCritter, procedure attack
    # could still be using pointer to it
    curCritter := p; CONTINUE END;

IF curCritter.hasLeaves THENB # add solar energy

    write(dbgf,"Adding solar energy" & eol);

    curCritter.spareEnergy .+ solarEnergyPerTurn;
    curCritter.totalEnergy .+ solarEnergyPerTurn;
    curCritter.spareEnergy .MIN maxEnergy;
    curCritter.totalEnergy .MIN maxEnergy;
    showCur END;

IF NOT curCritter.coroutine THEN
    curCritter.coroutine :=
        $createCoroutine(thisDataSection,
            "startCurCritter",
            curCritter.modName & " " &
                cvcs(curCritter.critterName) &
                cvs(critNum .+ 1L),
            # Smallish stack, so don't run out
            # of memory allocating coroutines
            (4096 DIV $pageSize) MAX 1);
$resumeCoroutine(curCritter.coroutine) END
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (continued)

```
    $WITH
        IF $exceptionName = $systemExcpt OR
            $exceptionName NEQ $descendantKilledExcpt AND
                $exceptionBits TST $cannotReturn THENB
            # Don't show msg for $descendantKilledExcpt
            # here because it happens all the time
            displayMessage(
                "Critter " & cvcs(curCritter.critterName)
                    & " (" & curCritter.modName & "):" &
                    eol &
                "Exception = " & $exceptionName & eol &
                $exceptionStringArg1 & eol &
                $exceptionStringArg2);
            killCritter(curCritter);
            IF $exceptionBits TST $cannotFallOut THEN
                $raise END
        EL  $raise;

    curCritter := curCritter.nextCritter END;

displayMessage("All critters have died.  End of world.");

clearScreen; deInitializeTerminal;
unbind(dpy); relModName("dpy");

END;

END "darwin"
```

Example 19.4-2. DARWIN, Ecological Simulation Executive Module (end)

```
BEGIN "plant"

RESTOREFROM "crtHdr";

DEFINE
    bestBirthEnergy =
        halfBirthEnergy MAX (energyForLeaves + 1);

POINTER(what) ARRAY (-2 TO 2,-2 TO 2) w;

INITIAL PROCEDURE;
BEGIN
BOOLEAN clearSpaceClose;
INTEGER x,y;
addLeaves;
DOB lookAround(w);
    clearSpaceClose := FALSE;
    FOR x := -1 UPTO 1 DO FOR y := -1 UPTO 1 DO
        IF w[x,y].status = emptyCell THEN
            clearSpaceClose := TRUE;
    IF clearSpaceClose AND
        w[0,0].spareEnergy >
            birthCost + bestBirthEnergy THENB
        DOB x := ran(-1,1); y := ran(-1,1) END
            UNTIL (x OR y) AND w[x,y].status = emptyCell;
        reproduce(x,y,bestBirthEnergy) END END;
END;

END "plant"
```

Example 19.4-3. PLANT Critter

```
BEGIN "eater"

RESTOREFROM "crtHdr";
```

Example 19.4-4. EATER Critter (continued)

```
DEFINE
    bestBirthEnergy =
        halfBirthEnergy MAX (energyForLegs + 1),
    reserve =
        2 * attackCost;

POINTER(what) ARRAY (-2 TO 2,-2 TO 2) w;

INTEGER PROCEDURE sgn (INTEGER i);
RETURN(IF i > 0 THEN 1
       EF i < 0 THEN -1
       EL 0);



INITIAL PROCEDURE;
BEGIN
INTEGER x,y,tShell,tTotEn,tX,tY,eX,eY;
    # Target's shell, total energy, x, y, and empty x, y
POINTER(what) me,p;
addLegs;
DOB lookAround(w); me := w[0,0];
    tX := tY := eX := eY := 0;
    tShell := $maxInteger; tTotEn := - $maxInteger;

    FOR x := -1 UPTO 1 DO FOR y := -1 UPTO 1 DO
        # Search immediately adjacent spaces
        IF (p := w[x,y]).status = critterCell AND
            p.critterName NEQ me.critterName AND
            # Go for less shell first, then more energy
            (p.shellAmount < tShell
                OR p.shellAmount = tShell AND
                    p.totalEnergy > tTotEn) THENB
                    tX := x; tY := y;
                    tShell := p.shellAmount;
                    tTotEn := p.totalEnergy END
        EF p.status = emptyCell THENB
            eX := x; eY := y END;
```

Example 19.4-4. EATER Critter (continued)

```
    IF NOT (tX OR tY) THENB
        # Didn't find a creature adjacent, look farther
        FOR x := -2 UPTO 2 DO FOR y := -2 UPTO 2 DO
            IF (p := w[x,y]).status = critterCell AND
                p.critterName NEQ me.critterName AND
                (p.shellAmount < tShell
                    OR p.shellAmount = tShell AND
                        p.totalEnergy > tTotEn) THENB
                        tX := x; tY := y;
                        tShell := p.shellAmount;
                        tTotEn := p.totalEnergy END;
        IF tX OR tY THENB # Move towards target
            move(sgn(tX),sgn(tY)); CONTINUE END END;

    IF tX OR tY THEN attack(tX,tY);

    # Now, should we try to reproduce, grow shell, or
    # grow teeth?

    CASE ran(1,3) OFB
        [1] IF me.spareEnergy > reserve + energyPerShell
                AND me.shellAmount < maxShell THEN
                addShell;
        [2] IF me.spareEnergy > reserve + energyPerTooth
                AND me.teethAmount < maxTeeth THEN
                addTeeth;
        [3] IF me.spareEnergy >
                reserve + birthCost + bestBirthEnergy
                AND (eX OR eY) THEN
                reproduce(eX,eY,bestBirthEnergy);
        END END;
END;

END "eater"
```

Example 19.4-4. EATER Critter (end)

19.5. Coroutines and Exceptions

When an exception occurs in a child coroutine, the exception propagates within the coroutine in the usual fashion. If it is handled within the coroutine, then the coroutine's ancestors are not affected. However, if a child coroutine does not handle an exception, then each of the ancestors in turn is given an opportunity to handle it; the system macro $exceptionCoroutine returns the coroutine in which the exception was originally raised. The "MAINSAIL Language Manual" contains a more detailed description of exceptions in coroutines.

A program may run an application in a child coroutine, as in Example 19.3-1. Suppose that the program wants to handle its own exceptions, but wants its ancestors to handle exceptions generated by the application. The program's Handle Statements may look like:

```
$HANDLE
    <code that may be running while a child is running>
$WITHB
    IF $exceptionCoroutine NEQ $thisCoroutine
        OR <exception isn't one we want to handle> THEN
        $raise;
    <handling code> END;
```

If desired, a parameter to $raise permits one coroutine to raise an exception in another; see the "MAINSAIL Language Manual" for details. An example appears in the DARWIN executive module of Example 19.4-2, where an exception is raised in a critter coroutine when the critter is under attack.

19.6. Exercises

Exercise 19-1.

Write a scheduler module and some applications that remedy the following defects of Example 19.3-1 by using the MAINSAIL display modules:

- Intermingling of input and output. Create a special region of the screen for each application.
- Failure to handle exceptions in child coroutines. Abort only the child in which the exception occurs.

The MAINSAIL STREAMS package can be used to overcome the problem of blocking on terminal input, but you need not use STREAMS in your answer to this exercise.

Exercise 19-2.

Write a new critter that runs under the DARWIN executive module of Example 19.4-2. Critters that are superior to both exisiting modules should be easy enough to write; test them by running them against the PLANT and EATER example modules of Examples 19.4-3 and 19.4-4.

20. Bootstraps, Libraries, and Intmods

MAINSAIL starts execution as a host system executable file called a "bootstrap". In order to execute on an operating system, the bootstrap is constructed in accordance with the operating system's conventions for executable files. The bootstrap is the only piece of MAINSAIL so constructed; all MAINSAIL object modules are written in a format specific to MAINSAIL, which has no relationship with the operating system's linkable or executable object file format.

Many MAINSAIL object modules may be stored together in a file called an "objmod library" (or just a "library" when the context is clear). When a MAINSAIL execution starts, one of the first thing it does is open the library where it expects to find certain crucial modules of the runtime system (the "system library"). The system library is not the only objmod library that may be used; programmers may group their own modules into libraries as well.

Intmods are repositories for symbols used by MAINSAIL system porgrams or shared among a number of modules. For the latter purpose, they are often superior to sourcefiled header files or source libraries.

20.1. Bootstraps

The standard MAINSAIL bootstrap is adequate for many purposes. However, users may sometimes want to construct their own custom bootstraps for various reasons:

- To have MAINSAIL run an initial module rather than come up at the MAINEX asterisk prompt.
- To specify different locations for the system library and kernel files (the kernel file is loaded by the bootstrap and provides many MAINSAIL system functions; the kernel in turn opens and reads in the necessary modules from the system library).
- To specify a list of foreign object modules (linked with the bootstrap) to which MAINSAIL is to have access through the Foreign Language Interface.
- To specify that MAINSAIL is to be invoked from a foreign language program.
- To specify initial MAINEX subcommands that govern the execution of subsequent modules.
- To alter parameters governing memory usage and memory management; the most important of these parameters are those governing the maximum memory that

MAINSAIL may request from the operating system and the frequency of garbage collection. Some operating systems may also allow the user to specify the size of the initial coroutine's stack.

The MAINSAIL utility module CONF is used to build a bootstrap. The portable part of CONF is described in the "MAINSAIL Utilities User's Guide"; operating-system-specific commands, if any, are described in the guides for each operating system.

20.1.1. Bootstrap Caveats

On some systems, MAINSAIL requires certain parameters to be set correctly in the bootstrap in order to run. For example, on UNIX, the "UNIXBITS" parameter must be set to the value shipped in the system bootstrap configuration file and the "FOREIGNMODULES" parameter must include all the modules listed in the system configuration file. If you use CONF's "SAVE" command to create a UNIX parameters file, be sure not to use "RESTORE" to read the file under a later version of MAINSAIL without making sure that the "UNIXBITS" and "FOREIGNMODULES" parameters are compatible with the required values for the new version of MAINSAIL.

20.2. Objmod Libraries

Objmod libraries provide several advantages:

- When a module resides in an open library, MAINSAIL does not have to attempt to open a new object file; it can read it from an already open file, saving the operating system overhead for a file open (except when the "EXEFILE" MAINEX subcommand is in effect; consult the "MAINSAIL Utilities User's Guide" for details).

- Opening fewer files means consuming fewer operating system file handles. Some operating systems impose an annoyingly low limit on the number of simultaneously open files.

- When a module read from an open library is swapped (see Section 2.1 of part II of the "MAINSAIL Tutorial"), the MAINSAIL runtime does not write it out into a separate swap file, since it knows it can read it in again from the library.

Objmod libraries are typically opened towards the beginning of a MAINSAIL execution with the "OPENEXELIB" MAINEX subcommand (which may be installed in a bootstrap) or with the openLibrary system procedure. Once a library file is opened, it typically remains open until the end of the execution (although it may be closed, if desired, with the closeLibrary system procedure or the "CLOSEEXELIB" MAINEX subcommand).

Objmod libraries are managed with an interactive utility, MODLIB, which is described in the "MAINSAIL Utilities User's Guide".

The MAINSAIL compiler can compile directly into objmod libraries; consult the "MAINSAIL Compiler User's Guide" for details.

Some operating systems provide a facility for "mapping" an objmod library, making it resident in memory; consult the appropriate system-specific user's guide for details. Mapping libraries, if available, can result in considerably faster access to modules within the library, at the expense of making the memory occupied by the library unavailable for other purposes.

20.3. Intmods and Intmod Libraries

Intmods are files that may substitute for sourcefiled header files (files of definitions shared by several modules). Intmods are also required by various MAINSAIL utility programs, and are produced when certain compiler subcommands (e.g., "DEBUG", "ALIST", etc.) are in effect. Intmods that can substitute for header files are made when the "SAVEON" source directive or compiler subcommand is in use, and the symbols from an intmod may be made visible with the "RESTOREFROM" source directive.

Intmods are made from complete modules; i.e., they save all the symbols in a module. As an example, imagine that the lines reading:

```
SOURCEFILE "decls";
```

in Examples 14.6-1 and 14.6-2 were replaced with:

```
RESTOREFROM "decls";
```

An intmod made from a module DECLS could replace the sourcefiled file "decls" of Example 14.6-3. It would look like Example 20.3-1. The "SAVEON" directive at the end causes the compiler to produce an intmod for DECLS.

Intmods have several advantages over header files:

- The compiler has already processed the declarations in an intmod file and translated them into an internal form. It is faster to make an intmod visible than to read a source file.

- Intmods may be used implicitly; an identifier "foo" from an intmod BAR may be specified as "bar$foo" in a program. There is no need to issue a directive to open BAR's intmod explicitly; the compiler looks for the intmod in a search analogous to the search for executable objmods.

```
BEGIN "decls"

MODULE itf1 (
    PROCEDURE proc1 (MODIFIES INTEGER i2);
);

MODULE itf2 (
    INTEGER i2;
    PROCEDURE proc2 (STRING whereFrom);
);

SAVEON;

END "decls"
```

Example 20.3-1. A Module Compiled to Produce an Intmod

- When an intmod is made visible, some of the symbols it contains may still be invisible; directives exist to control visibility on a per-symbol basis. Sometimes a visible symbol in a package of symbols needs to use some supporting variables or procedures, but the supporting symbols would not be useful outside the package, and so should not be visible outside it. This sort of information hiding is unavailable in a sourcefiled header file.

- Intmods may be grouped into library files analagous to objmod library files. This prevents clutter in the file system and reduces the number of simultaneously open files. Intmod libraries are managed with a utility called INTLIB, which is very similar to MODLIB for objmod libraries.

More on intmods and intmod libraries may be found in the "MAINSAIL Language Manual" and in the chapter on INTLIB in the "MAINSAIL Utilities User's Guide". Examples 19.4-2, 19.4-3, and 19.4-4 show modules that restore from a common intmod.

20.4. Foreign Language Modules

MAINSAIL can call routines in certain foreign (non-MAINSAIL) languages, and certain foreign languages can call MAINSAIL procedures. A MAINSAIL module interface is used to describe the foreign routines; a MAINSAIL module with interface procedure headers that "simulate" the foreign procedure headers is compiled with special compiler subcommands, and the resulting file (which is not a regular MAINSAIL objmod file, but an operating-system-specific assembly or linker file) is linked with a MAINSAIL bootstrap to allow MAINSAIL to call the foreign language or vice versa (depending on which compiler subcommands were given). The MAINSAIL Foreign Language Interface (FLI) is described in the "MAINSAIL Compiler User's Guide" and in the appropriate operating-system-dependent user's guide.

The FLI is available only for certain language-operating system combinations, and in some cases may be available only for calls from MAINSAIL to a foreign language (a "Foreign Call Compiler", or FCC) or for calls from a foreign language to MAINSAIL (a "MAINSAIL Entry Compiler", or MEC). If you require a combination that is not presently available, contact XIDAK for information.

20.4.1. Special FLI Considerations

20.4.1.1. Strings

The method of passing MAINSAIL strings to a foreign language varies from language to language. Some languages do not have any data structure corresponding to a MAINSAIL string; in such a case, passing a MAINSAIL string to a foreign language is illegal.

In some cases, a string must be passed as a charadr (the location of the first character in the MAINSAIL string); sometimes a null (zero) character must be appended to the string before the charadr is computed.

In some cases, a single MAINSAIL string corresponds to two separate parameters to a foreign language: one parameter indicates the location of the characters, the other the length of the string.

A foreign language must not change characters of a string allocated in MAINSAIL string space.

A foreign language must not attempt to access characters beyond the end of the MAINSAIL string. The MAINSAIL string might be allocated near the end of the memory accessible to MAINSAIL; accessing data beyond the end of the string might cause a (system-dependent) memory violation.

When a foreign language allocates a string and passes it to MAINSAIL, it may be desirable to use $getInArea in order to move the string into MAINSAIL string space. This is necessary if the foreign language is likely to reuse the space in which it allocated the string, but the MAINSAIL program wishes to continue referring to the string after the foreign language has reused the space. By copying the string to MAINSAIL string space, the program may ensure that the characters of the string will remain the same as long as a MAINSAIL string variable points to them (the characters are collected when the string becomes inaccessible). The use of $getInArea is necessary only if the space where the foreign string was allocated is to be reused;

the MAINSAIL garbage collector is not confused by string variables that point outside MAINSAIL string space.

20.4.1.2. Arrays

Arrays are usually passed as the address of the first element of the array to languages without the concept of a dynamically sized array. Additional information must be passed if the foreign language needs to determine the size of the array.

In some languages, there is no way to refer to an "unallocated" array. In such a case, passing a MAINSAIL nullArray to the foreign language raises the exception $nullArrayExcpt.

A foreign language must not access array elements outside of the array; as in the case of characters beyond the end of a string, attempting to access such elements may cause a memory violation.

20.5. Bootstrap and Library Example

Suppose a bootstrap is to be constructed for a music synthesis program. An FLI module, MUSFLI, is required to access the sound generation software or hardware on the system; the MAINSAIL modules for the music synthesis program are:

```
MUSMOD
NOTE
TIMBRE
SYNTH
```

It is desired to put the modules into a single objmod library for XYZ operating system (where $systemNameAbbreviation is "xyz") and open this library in the MAINSAIL bootstrap.

Assuming the source files end in ".msl", the commands to compile the modules into a library, "music-xyz.olb", are shown in Example 20.5-1. Alternatively, the library may be built from the objmod files with the commands shown in Example 20.5-2.

The FLI module, MUSFLI, must be compiled (see Example 20.5-3). It must then (on most systems) be assembled with the system assembler; in addition, the foreign language code implementing MUSFLI must be compiled or assembled (not shown). It is assumed here that on the XYZ operating system, the linker file for MUSFLI is "musfli.link", and for the foreign code, "mus.link".

The bootstrap file must be made to open "music-xyz.olb" and and to specify MUSFLI as a foreign module (see Example 20.5-4). "=" is used to indicate that the multi-line parameters "FOREIGNMODULES" and "SUBCOMMANDS" should start with their original values. The

```
MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): musmod.msl,<eol>
> outobjlib music-xyz.olb<eol>
> <eol>
Opening intmod for $SYS...

musmod.msl 1 2 3 4 5 6 7 8 9 10 11
Created library music-xyz.olb
Adding musmod to music-xyz.olb
Intmod for MUSMOD not stored

compile (? for help): note.msl<eol>
Opening intmod for $SYS...

note.msl 1 2 3 4 5
Adding note to music-xyz.olb
Intmod for NOTE not stored

compile (? for help): timbre.msl<eol>
Opening intmod for $SYS...

timbre.msl 1 2 3 4
Adding timbre to music-xyz.olb
Intmod for TIMBRE not stored

compile (? for help): synth.msl<eol>
Opening intmod for $SYS...

synth.msl 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Adding synth to music-xyz.olb
Intmod for SYNTH not stored

compile (? for help): <eol>
```

Example 20.5-1. Compiling Modules into a Library

CONF output must (on most systems) be assembled with the system assembler (not shown); it is assumed the resuling linker file is "music.link".

```
MAINSAIL (R) Objmod Librarian (? for help)
Copyright (c) 1984, 1985, 1986, and 1987 by XIDAK, Inc.,
 Menlo Park, California, USA.

MODLIB: create music-xyz.olb<eol>
Created library music-xyz.olb
MODLIB: add music-xyz.olb musmod note timbre synth<eol>
Adding musmod to music-xyz.olb
Adding note to music-xyz.olb
Adding timbre to music-xyz.olb
Adding synth to music-xyz.olb
MODLIB: <eol>
```

Example 20.5-2. Building a Module Library

```
MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, and 1986 by XIDAK, Inc.,
 Menlo Park, California, USA.

compile (? for help): musfli.msl,
> fli t<abbreviation for foreign language><eol>
>

musfli.msl 1
Object module for MUSFLI stored on musfli.<extension>

compile (? for help): <eol>
```

Example 20.5-3. Compiling the FLI Module MUSFLI

Example 20.5-5 shows a hypothetical linker command for the music bootstrap. Of course, linkers vary a great deal from system to system; consult the appropriate system-dependent MAINSAIL user's guide and, if necessary, the system linker manual for details.

```
MAINSAIL (R) Bootstrap Configurator
Restoring configuration values from file
 <system configuration file>
CONF: bootfilename music.<extension><eol>
CONF: foreignmodules<eol>
FOREIGNMODULES is
SYSMD1
SYSMD2
SYSMD3
Should be:
=<eol>
MUSFLI<eol>
<eol>
CONF: subcommands<eol>
SUBCOMMANDS is
SUBCOMMANDS <system subcommand file>
Should be:
=<eol>
openexelib music-xyz.olb<eol>
<eol>
CONF: <eol>
Bootstrap written in file music.<extension>
```

Example 20.5-4. Making the Bootstrap for the Music Program

```
OS prompt:
link music.link musfli.link mus.link into music<eol>
Linkage complete, no errors.
OS prompt:
run music<eol>
<MAINSAIL executes>
```

Example 20.5-5. Linking and Running the Bootstrap

Index

XIDAK, Inc., 530 Oak Grove Avenue, M/S 101, Menlo Park, CA 94025, (415) 324-8745