V12

TM

MAINSAIL®

Tutorial, Volume II

MAINSAIL® Tutorial, Part II:

Efficiency and Implementation Tips for the Advanced MAINSAIL Programmer

with Answers to the Exercises of Part I

24 March 1989

Table of Contents

Appendices

List of Examples

List of Tables

1. Program Efficiency

This chapter describes a variety of techniques for writing efficient MAINSAIL code. Some of the techniques may make a considerable difference in a program's execution speed, and others have relatively minor effects. Of course, if a program is to execute well, its algorithm must be properly chosen to begin with; most "efficient programming techniques", including those described in this chapter, can be used to fine-tune a program but cannot rescue one that is grossly miswritten.

1.1. Detecting Sources of Inefficiency

The first step in improving a program's execution speed, once a satisfactory overall algorithm has been chosen, is to determine where the program is spending most of its time. Improvements to infrequently executed code are usually not worth a programmer's time, since they result in a negligible improvement in the performance of the program as a whole.

The MAINSAIL performance monitor, MAINPM, is an excellent tool for tracking the execution of programs in order to determine where they need improvement. MAINPM can give timings for various parts of a program as well as counts of individual statement executions. MAINPM can also be made to keep track of any user-specified quantity during program execution instead of CPU time. Furthermore, it can keep track of use of chunks (records, arrays, and data sections) and string space. MAINPM is described in detail in the "MAINPM User's Guide".

1.2. Use of Low-Level Data Types and Primitives

As described in Chapter 18 of part I of the "MAINSAIL Tutorial", the use of charadrs and addresses can be used to avoid some of the overhead associated with MAINSAIL high-level data types. Caution must be used when substituting low-level operations for high-level ones, for several reasons:

- Low-level operations are less convenient and more difficult to code than high-level ones. The programmer is more likely to err in writing low-level code.

- Low-level errors may be subtle and difficult to trace, since the program can inadvertently overwrite MAINSAIL data structures. Particular care must be used if an address or charadr points into a MAINSAIL data structure or into string space, since a garbage collection may move data structures or characters in string space.

- High-level primitives often provide built-in explicit checks for errors (e.g., array subscript checking). Such automatic checking is unavailable with low-level primitives.

- If garbage is accidentally generated in a static storage area, it cannot be reclaimed by MAINSAIL's garbage collector.

1.3. Explicit Disposal

Avoiding unnecessary garbage collections is the purpose of many MAINSAIL efficiency techniques. A garbage collection brings MAINSAIL execution to a halt until the collection finishes; this can be annoying in an interactive program, especially if the collection is long.

On systems that provide virtual memory (most modern operating systems do), garbage collection can take much longer than usual if a process's virtual address space has exceeded the amount of physical memory on the machine. MAINSAIL memory management accesses the pages of MAINSAIL's address space in an almost random fashion, which causes a virtual memory system to page heavily. The slowdown after a process has "gone virtual" can dramatic; garbage collections can take literally orders of magnitude longer.

Section 18.2 of part I of the "MAINSAIL Tutorial" describes the use of the system procedure "dispose". dispose recycles memory without requiring a garbage collection; in some cases, sufficiently frequent use of dispose may avoid collections altogether. The use of dispose is therefore recommended for efficiency reasons, although, as described in Section 18.2 of part I of the "MAINSAIL Tutorial", considerable caution is required.

MAINPM facilities may be useful in determining where garbage is generated; see the "MAINPM User's Guide".

1.4. Free Lists; $newRecords

Some programs allocate and dispose records of some class with great frequency. In such cases, the overhead of the system procedures "new" and "dispose" may be considered too great. A program may keep its own free list of a class, making allocation and disposal of a record of the class very fast. Example 1.4-1 shows the allocation and disposal procedures for a class fooCls.

It is not worthwhile for a program to maintain its own free list unless the allocation and disposal of records is quite frequent. The built-in new and dispose already use a free-list-like mechanism; their overhead is often not much greater than that of the procedures shown in Example 1.4-1. A program maintaining its own free list also gives up some flexibility. A free list maintained as in Example 1.4-1 grows to the maximum number of records needed at one time and does not shrink thereafter, even if the number of records needed becomes smaller again. Unlike records recycled with the built-in dispose, the space occupied by the free list

```
CLASS fooCls (
    POINTER(fooCls) next; # free list link
    ... other fields ...
);

POINTER(fooCls) fooClsFreeList;

INLINE POINTER(fooCls) PROCEDURE newFooCls;
# Use built-in new only when free list is exhausted.
BEGIN
POINTER(fooCls) p;
IF p := fooClsFreeList THENB
    fooClsFreeList := p.next; RETURN(p) END;
RETURN(new(fooCls));
END;



INLINE PROCEDURE disposeFooCls
    (MODIFIES POINTER(fooCls) p);
BEGIN
p.next := fooClsFreeList; fooClsFreeList := p;
p := NULLPOINTER;
END;
```

Example 1.4-1. Special-Purpose Allocation and Disposal Procedures

cannot be used for data structures other than records of fooCls. Records allocated with the special-purpose mechanism are not initially cleared.

An effective way of reducing the overhead of record allocation is to call $newRecords, which allocates many records at once. Example 1.4-2 shows the procedure newFooCls of Example 1.4-1 rewritten to allocate 50 new records whenever the free list is close to exhaustion. $newRecords also exists in a form that allocates an array of records; consult the "MAINSAIL Language Manual" for details.

1.5. Areas

Areas are portions of memory that can contain an entire data structure: records, arrays, data sections, and string text. An entire area can be disposed at once, which is a very efficient

```
POINTER(fooCls) PROCEDURE newFooCls;
# The free list is assumed initialized to contain at
# least one record by the initial procedure.
BEGIN
POINTER(fooCls) p;
IF (p := fooClsFreeList).next THEN
    fooClsFreeList := fooClsFreeList.next
EF NOT fooClsFreeList :=
    $newRecords(p,DSP(fooCls.next),50L) THEN
    errMsg("Couldn't allocate records","",fatal);
RETURN(p);
END;
```

Example 1.4-2. Use of $newRecords

operation. In programs where large amounts of data become obsolete at some well-defined point, allocating the data in an area and disposing the entire area at once can greatly reduce memory managment overhead. Areas are also appropriate for data structures in which no garbage is being generated, since garbage collection may be disabled for just those areas.

When using areas, considerable care must be taken to avoid errors that are difficult to track. See the "MAINSAIL Language Manual" for details.

1.6. Use of String Space

Garbage collections can be triggered by use of string space. Many string operations use string space, and it is difficult to avoid using string space entirely. Section 2.3.3 describes how a number of system procedures affect string space, and includes examples of code that makes good use of string space.

1.7. String Comparison

As desribed in Section 10.6 of part I of the "MAINSAIL Tutorial", string comparison may be most efficiently performed with the standard comparison operators ("=", "NEQ", ">", etc.) or with the special-purpose procedures "compare" and "equ", depending on the circumstances.

1.8. Concatenation of Strings in Calls to write

A common source of unnecessary inefficiency is the improper use of concatenation in a string written to a file. The inefficiency comes from two sources:

1. The failure to allow the compiler to perform concatenation at compiletime when it can, requiring that "write" be called more often than necessary.

2. The unnecessary concatenation of strings constructed at runtime, rather than making multiple calls to "write". This can eat up quite a bit of string space, and is the more serious source of inefficiency.

Assuming that i and j are integer variables, consider the following call to "write":

```
write(logFile,"First number:",tab,i,eol,
    "Second number:",tab,j,eol);
```

It makes eight calls to the string and integer instances for write to a textFile. Fewer calls could be made if adjacent string constants were concatenated to produce the recommended way of writing these strings to logFile:

```
write(logFile,"First number:" & tab,i,eol
    & "Second number:" & tab,j,eol);
```

which makes only five calls to write. The concatenations (between constant strings only!) are performed at compiletime; e.g.:

```
eol & "Second number:" & eol
```

is evaluated by the compiler to produce a single string.

It would be a mistake (the second source of inefficiency mentioned above) to concatenate everything to be written:

```
write(logFile,"First number:" & tab & cvs(i) & eol &
    "Second number:" & tab & cvs(j) & eol);
```

In this case, some of the strings concatenated together are not constants, since "cvs(i)" and "cvs(j)" cannot be evaluated at compiletime (although they could be if i and j were constants instead of variables). The compiler, recognizing that the concatenated expression is not composed entirely of constants, may fail to perform any of the possible concatenations at compiletime. Worse, since the compiler may evaluate the operands of "&" in any order, it may start by placing the last operand into string space, then concatenating it with the previous operand (requiring the copying of the last operand), then with the previous (requiring the copying of the last two operands), and so on. Enormous quantities of string space can be used if enough strings are concatenated together.

1.9. Inline Procedures

Example 1.4-1 shows two inline procedures: procedures that are expanded inline, somewhat like macros. That is, a separate body for the procedure is not produced in the objmod; the body of the procedure is expanded inside the calling procedure, and the normal procedure call overhead is avoided. The semantics of an inline procedure call are exactly the same as a regular ("closed") call.

An inline procedure declaration is preceded by the qualifier "INLINE". An individual procedure call may also be preceded by the keyword "INLINE", in which case that call is also made inline (unless it is an interface procedure of another module). The keyword "$ALWAYS" must precede "INLINE" if a procedure or call is to be made inline when compiled debuggable, since the "DEBUG" option normally forces calls to be closed (so that the debugger can jump into them).

An inline procedure is often a better way to package a small amount of code than a macro. Inline procedures are more flexible with respect to parameters; they can take advantage of the generic mechanism, and can use the "REPEATABLE" and "OPTIONAL" qualifiers, with the usual effect. Furthermore, inline procedures can have local variables. Macros are best used when their strong point, the substitution of arbitrary text into source code, is really needed.

Although inline calls are faster, they usually consume more space than closed calls. It is inadvisable to make a frequently called procedure inline unless it is quite small; otherwise, the objmod may become quite large, or the compiler may run out of memory trying to compile it.

1.10. Compiletime Evaluation

Compiletime evaluation may be used for data types other than strings; it performs the operations only once, at compiletime, instead of (potentially) many times, at runtime. Attention must be paid to the order of evaluation of operands; constant operands should be grouped together, either using "DEFINE" to create a new constant identifier or using parentheses to force operators with constant operands to be evaluated first. Otherwise, the compiler may not recognize the constant expression, and unnecessarily evaluate it at runtime. For example, if i is an integer variable, then:

```
i + 2 + 4
```

may be evaluated by the compiler as:

```
(i + 2) + 4
```

instead of:

```
i + (2 + 4)
```

The former expression entails two additions at runtime, the second only one. An alternative that makes it clearer to the reader that compiletime evaluation is being performed is:

```
DEFINE six = 2 + 4;
... i + six...
```

since macro constants can be defined only as constant expressions.

(Long) real expressions are never evaluated at compiletime, so this strategy does not work for them.

1.11. Substitution of Short for Long Operations

Avoid unnecessary long integer multiplication and division, since some processors on which MAINSAIL runs do not provide these operations as instructions; the operations are performed in software, which is slow. For example, if it is known that the product of two integers i and j is within the guaranteed integer range, use:

```
cvli(i * j)
```

rather than:

```
cvli(i) * cvli(j)
```

1.12. Comparisons with Zero

Comparison with Zero is more efficient on many processors than comparison with non-Zero values. For example, if it is known that an integer variable i must have either the value -1 or the value 0, then the code:

```
IF i THEN...
```

often compiles to more efficient object code than:

```
IF i = -1 THEN...
```

1.13. Intermodule vs. Local Calls

A local (within the same module) call is faster than an intermodule call, though usually not so much faster as to make a great difference in execution speed. In some cases, though, it may be better to compile a small, frequently called procedure into every module that uses it than to put it in a single module as an interface procedure.

1.14. Using exceptionPointerArg to Determine the Current Exception

Determining the current exception can be inefficient if many different exceptions are possible. For example, code to handle each of the following exceptions:

```
Exception 1
Exception 2
Exception 3
Exception A
Exception B
Exception C
```

would look like:

```
IF $exceptionName = "Exception 1" THENB ... END
EF $exceptionName = "Exception 2" THENB ... END
EF $exceptionName = "Exception 3" THENB ... END
EF $exceptionName = "Exception A" THENB ... END
EF $exceptionName = "Exception B" THENB ... END
EF $exceptionName = "Exception C" THENB ... END
EL $raise;
```

A string compare is not a very fast operation. It would be more desirable to use a Case Statement in this instance, but it is not possible to use a string as a Case Statement index. The solution is to pass a known record containing an exception code in the exceptionPointerArg parameter of $raise. See Example 1.14-1. This strategy requires an integer assignment (the exception code) at the time the exception is raised, a pointer comparison in the handler, and a Case Statement in order to get to the code to handle the exception properly. This is typically much faster than a series of string comparisons, especially if the list of possibilities is long.

1.15. Input and Output of Large Quantities of Data

Section 18.3 of part I of the "MAINSAIL Tutorial" discusses $storageUnitRead, $storageUnitWrite, $characterRead, $characterWrite, $pageRead, and $pageWrite, which may be considerably faster than reading or writing with "read" and "write" when large quantities of

```
CLASS excptCodeCls (
    INTEGER exceptionCode;
);

POINTER(excptCodeCls) excptCode;

                 .
                 .
                 .

# Code to raise an exception:
excptCode.exceptionCode := ...;
$raise("Exception ...","","",excptCode);

                 .
                 .
                 .

# Code to handle an exception:
$HANDLE ...
$WITH
   IF $exceptionPointerArg NEQ excptCode THEN $raise
   EL CASE excptCode.exceptionCode OFB
        # Assume codes range from 1 to n
        [1] ...
                 .
                 .
                 .
        [n] ...
        END;

                 .
                 .
                 .

INITIAL PROCEDURE;
... allocate excptCode...
```

Example 1.14-1. Using exceptionPointerArg to Recognize an Exception Quickly

data are involved. Opening high-volume I/O files with the $unbuffered bit may also result in improved efficiency, but beware of the restrictions on unbuffered files listed in the description of "open" in the "MAINSAIL Language Manual".

1.16. The Structure Blaster

The Structure Blaster (see the "MAINSAIL Structure Blaster User's Guide") provides a way to read and write MAINSAIL data structures. The Structure Blaster is designed to be especially fast on input; indeed, for large structures, it may be faster to read a structure with the Structure Blaster than with "hand-crafted" routines written especially for the classes in the data structure. The Structure Blaster is typically slower on output than special-purpose routines, however. If a data structure is small, or if it is written almost as frequently as it is read, then it may be more efficient (although certainly less convenient) to write special code to store and retrieve a given data structure.

The $unbuffered bit may be used for Structure Blaster files. Its use is advisable unless only small, non-page-aligned structures are being read or written.

2. More on Memory Management and String Space

This chapter continues the discussion of memory management begun in Chapter 18 of part I of the "MAINSAIL Tutorial". It discusses the implementation of strings and string space in some detail.

The contents of this chapter are current as of the date of this writing, but some details are subject to change in future releases. Only those features described in the "MAINSAIL Language Manual" are actually guaranteed to be available in the future.

2.1. Module Swapping

MAINSAIL does not need to keep in memory all control sections for which data sections exist. Some control sections may be "swapped out"; in fact, only the currently executing control section (and the MAINSAIL kernel module's control section) need be in memory at any given time. Module swapping takes place entirely automatically; most programs are completely unaware when swapping takes place.

A swapped-out control section is written to a file called the "swap file", unless the module was taken from an open module library, in which case MAINSAIL assumes that the control section may be re-read from the library when necessary. One side effect of closeLibrary is to read into memory (and perhaps then copy to the swap file) any modules from the library of which the control sections may be needed in the future.

MAINSAIL attempts to choose a unique name for the swap file, i.e., a name that does not match that of any other file on the current directory; otherwise, problems would arise if two MAINSAIL processes connected to the same directory were both swapping.

Modules are normally swapped only when the memory available to MAINSAIL is nearly full. Module swapping can be time-consuming, so a MAINSAIL program that starts swapping intensively can appear to come to a halt (a program low on memory may be executing slowly even before swapping because of frequent garbage collection). The solution to this problem is to increase the amount of memory allowed to MAINSAIL by the bootstrap parameters or to rewrite the program so that fewer data structures are in memory (MAINSAIL swaps only code, never data). If neither of these is possible, you need to move to a computer with more memory.

The MAINEX subcommmand "SWAPINFO" causes MAINSAIL to display a message whenever module swapping occurs; consult the "MAINSAIL Utilities User's Guide" for details.

2.2. Selecting Memory Management Parameters

The memory management parameters provided in the MAINSAIL bootstrap for all operating systems are "COLLECTMEMORYPERCENT", "MAXMEMORYSIZE", and "INITIALSTATICPOOLSIZE". Some additional parameters may be provided on some operating systems, including the maximum address to be used by MAINSAIL and the size of MAINSAIL's stack; consult the appropriate system-specific MAINSAIL user's guide for details. In addition, MAINSAIL provides a variety of procedures to control the caching of random access files (see the "MAINSAIL Language Manual" for details).

It is not possible to determine a priori the best combination of bootstrap (and runtime, in the case of the file cache) settings of memory management parameters for a given application; this must be determined by experimentation. Users who are having problems with performance of large MAINSAIL programs should experiment with different combinations of "COLLECTMEMORYPERCENT", "MAXMEMORYSIZE", "INITIALSTATICPOOLSIZE", and other relevant parameters. Some considerations in the choice of parameters are:

- On many operating systems, it is better to allow MAINSAIL to garbage collect or swap modules than to allow the operating system's virtual memory facility to swap pages of the MAINSAIL process. On such systems the limit on MAINSAIL's memory should not exceed the physical memory available. On other systems, the tradeoff may be reversed, and a large virtual memory should be specified.

- Applications that produce garbage at a more or less steady rate may benefit more from frequent but short garbage collections than from infrequent, long collections. Long collections can be particularly annoying in interactive applications.

XIDAK regrets that it is not possible to offer more concrete advice on memory management performance, but it is our experience that such performance varies considerably from system to system and application to application. Users of MAINSAIL are invited to contact XIDAK with results of their own memory management experiments, if any interesting phenomena turn up.

2.3. Implementation of Strings

2.3.1. Introduction

The MAINSAIL string data type differs from the strings provided in most programming languages. In the typical Pascal-like language, a string is just a character array, and thus inherits the limitations of Pascal-like arrays, such as fixed bounds. The user must declare the length of the string as a constant, and the individual characters are altered when the string is altered, just like the elements of an array. Often, strings of different length cannot be compared or assigned to one another.

In contrast, a MAINSAIL string is a variable-length sequence of characters that can dynamically grow and shrink. The programmer does not give an upper bound to the length of the string (though no string can exceed 32,766 characters). A MAINSAIL string is implemented as a "string descriptor", which is an integer length and the charadr (character address) of the first character:

```
+------------------------+
| length                 |
+------------------------+
| charadr of first char  |
+------------------------+
```

"length(s)" returns the length part, while "cvc(s)" returns the charadr part. Both these procedures are very efficient. The null string has both parts equal to Zero.

The length is actually allocated as a long integer so that the length and charadr parts are the same size (on all machines for which MAINSAIL is implemented, long integers and charadrs are the same size). On eight-bit-byte-addressable machines, the format of a charadr is the same as that of an address.

The characters themselves reside in an area of memory called "string space". As strings are built up from new characters, string space fills up. When it becomes full, a garbage collection is performed that compresses string space to one end, squeezing out characters no longer referenced by any accessible string descriptor.

2.3.2. Operations That Do Not Put Characters into String Space

Certain string operations, such as assignment and substring access, involve just string descriptors, and hence are more memory-efficient than those that put characters into string space.

2.3.2.1. newString

"newString(c,t)" builds a string descriptor with length equal to t and charadr equal to c. This is a very efficient inline operation.

It is possible for a string descriptor to reference characters that are not in string space. For example, if the programmer has n characters on a static page at an address that he or she wishes to treat as a string, then the assignment:

```
s := newString(cvc(a),n)
```

creates a string descriptor that references those characters and assigns it to s. It is now possible to use s in subsequent string operations like any other string, since MAINSAIL has been designed to properly handle such "foreign" strings.

2.3.2.2. Assignment, Parameter Passing

String assignment and parameter passing assign or pass the string descriptor, not the characters themselves. For example, if s is "abcdef", and the assignment "r := s" is executed, s's descriptor is copied to r, so that r and s are identical and reference the same characters:

```
+----------------------------+
| 6                          |
+----------------------------+
| charadr of 'a' in "abcdef" |
+----------------------------+
```

Such sharing of characters does not lead to any confusion since there are no string operations that alter the characters of an existing string. String characters are "immutable objects" that can be created, but not altered. This is quite different from Pascal-like strings of which the individual characters can be altered like array elements. Since MAINSAIL string characters cannot be altered, there is no danger that after "r := s", the characters referenced by s could somehow be altered, thereby implicitly altering r.

Actually, MAINSAIL strings can be altered, but only by low-level use of charadrs, not by using the standard string manipulation facilities. For example, the following procedure sets the charNum'th character of s to charVal:

```
PROCEDURE changeChar (STRING s; INTEGER charNum,charVal);
IF 1 LEQ charNum LEQ length(s) THEN
    store(cvc(s),charVal,charNum - 1);
```

This is dangerous if the string is in MAINSAIL string space, because if any other string descriptors reference the altered character, the strings they represent will also be altered. For example:

```
r := s; changeChar(s,1,'x');
```

changes both r and s. If the programmer feels the need to alter individual characters of a string (for example, due to familiarity with Pascal), then the use of an integer array, or at least a string in scratch space, is recommended instead of a string in MAINSAIL string space. This is especially true if a string is passed to a MAINSAIL system procedure, since such procedures may "hold on" to strings when the programmer does not expect them to. The MAINSAIL system procedures are not designed for strings that change their text.

2.3.2.3. first(s), last(s), cRead(s), rcRead(s)

The procedures first, last, cRead, and rcRead generate efficient inline code. They are equivalent to the procedures listed in Example 2.3.2.3-1.

```
INLINE INTEGER PROCEDURE firstEquivalent (STRING s);
RETURN(IF s THEN cLoad(cvc(s)) EL -1);



INLINE INTEGER PROCEDURE lastEquivalent (STRING s);
RETURN(IF s THEN cLoad(cvc(s),length(s) - 1) EL -1);



INLINE INTEGER PROCEDURE cReadEquivalent
    (MODIFIES STRING s);
BEGIN
INTEGER i,t;
CHARADR c;
IF NOT s THEN RETURN(-1);
c := cvc(s); t := length(s);
i := cRead(c); s := IF t .- 1 THEN newString(c,t) EL "";
RETURN(i) END;



INLINE INTEGER PROCEDURE rcReadEquivalent
    (MODIFIES STRING s);
BEGIN
INTEGER i,t;
CHARADR c;
IF NOT s THEN RETURN(-1);
c := cvc(s); t := length(s);
i := cLoad(c,t .- 1); s := IF t THEN newString(c,t) EL "";
RETURN(i) END;
```

Example 2.3.2.3-1. Equivalent Procedures to Certain String Manipulation Procedures

The typical way to examine successive characters of a MAINSAIL string is as follows:

```
        WHILE t := cRead(s) GEQ 0 DOB ... examine t... END;
```

To examine from last to first use:

```
        WHILE t := rcRead(s) GEQ 0 DOB ... examine t... END;
```

In both cases the loop terminates when s becomes the null string, since t is then assigned -1. If the original value of s is needed after such a loop, then just assign s to r, and use r in the loop:

```
          r := s; WHILE t := cRead(r) GEQ 0 DOB ... END;
          <s still references the original string>
```

Since the assignment copies just the descriptors rather than the characters, it takes a negligible amount of time.

2.3.2.4. Substring: s[i TO j], s[i FOR j]

The substring operator builds a new string descriptor; no characters are accessed or put into string space. The two forms (using "TO" and "FOR") are equivalent to the procedures in Example 2.3.2.4-1.

```
INLINE STRING PROCEDURE toSubstring
    (STRING s; INTEGER i,j);
BEGIN
i .MAX 1; j .MIN length(s);
IF i > j THEN RETURN("");
i .- 1; RETURN(newString(displace(cvc(s),i),j - i)) END;


INLINE STRING PROCEDURE forSubstring
    (STRING s; INTEGER i,j);
RETURN(toSubstring(s,i,i + j - 1));
```

Example 2.3.2.4-1. Equivalent Procedures to the Two Forms of Substring

If read-only, random access to the characters of a string is required, the macro ith can be defined to access the ith character of a string s:

```
          DEFINE ith(s,i) = [first((s)[(i) FOR 1])];
```

If s is Zero or i < 1, or i > length(s), the result is -1. However, if the characters are being accessed in order, it is more efficient to use cRead (forwards) or rcRead (backwards) than ith.

2.3.3. String-Producing Operations

There are many operations that cause characters to be put into string space. In each case, characters are put into the "top" of string space. Consider the following picture of string space, where the x's represent used-up string space, and the blank space is available. The top of string space is just after the last x:

```
+------------------------+
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxx              |
|                        |
|                        |
|                        |
+------------------------+
```

If the user types "abcdef<eol>" to "s := ttyRead", string space looks like the following:

```
+------------------------+
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxabcdef        |
|                        |
|                        |
|                        |
+------------------------+
```

The string descriptor for s has length = 6 and a charadr that references the 'a' of "abcdef".

A description of the most common operations that put characters into string space should help the user to avoid inefficient string space usage when possible.

2.3.3.1. cvs(x)

x is one of the following data types: i, li, r, lr, b, lb. The string representation for x is put into the top of string space.

2.3.3.2. cvcs(i)

The character with code i is put into the top of string space.

2.3.3.3. cvl(s), cvu(s)

cvl copies s to the top of string space only if s contains an uppercase character. Similarly, cvu copies s only if s contains a lowercase character. Otherwise, the original string descriptor is returned, and no string space is used.

2.3.3.4. scan(s,...,upperCase)

If any of the scanned chars are lowercase, the result is copied to the top of string space; otherwise, a substring of s is returned, and no string space is used. If the omit option is used, no characters are put into string space (and the result is the null string).

2.3.3.5. "abc" (any string constant)

The first time a data section is created for a given module, an implicit array of string descriptors is allocated, with one element for each unique string constant in the module. All subsequent data sections of the module share the same array. If the same string constant occurs many times in a module, a single element represents all of them. The first time a string constant is referenced in (any instance of) a module, it is copied to the top of string space, and its element is initialized to reference the copied characters. Subsequent use of the same string constant detects that the element is already initialized, and hence does not copy the characters again.

2.3.3.6. ttyRead, scan(f,...), read(f,s), fldRead(f,...)

The string is read from the file into the top of string space. If the omit option is used for scan, no characters are put into string space (and the result is the null string).

2.3.3.7. r & s

In the general case, r is copied to the top of string space, and then s is copied to the new top of string space (immediately after r). A string descriptor referencing the concatenated characters is returned.

Special cases are detected to avoid copying characters:

- If r is Zero, return s.
- If s is Zero, return r.
- If r and s are already concatenated (i.e., s occurs immediately after r in string space), then do not copy any characters.
- If r is already at the top of string space, do not copy it.

2.3.3.8. write(s,x)

x is one of the following data types: i, li, r, lr, b, lb, s. If x is a string, "write(s,x)" is equivalent to "s .& x"; otherwise, it is equivalent to "s .& cvs(x)".

Note that "write(s,a,b,c)" usually uses less string space than the equivalent "s .& (cvs(a) & cvs(b) & cvs(c))" since the former is really "write(s,a); write(s,b); write(s,c)", which first copies s to the top of string space, then cvs(a) (s is already at top), then cvs(b) (s is already at top), and finally cvs(c) (s is already at top). The latter operation is evaluated as if it were "r := cvs(a) & cvs(b) & cvs(c); s .& r". This causes r to be formed at the top of string space. Unless s was already at the top, it will have to copied on top of r, and then r copied once more on top of s, an inefficient operation.

2.3.3.9. cWrite(s,i)

Copy s to the top of string space if not Zero and not already at top, then put i immediately after s. Repeated use of cWrite, such as:

```
s := ""; FOR i := 1 UPTO n DO cWrite(s,char[i]);
```

efficiently builds s at the top of string space, one character at a time, since after the first cWrite, s remains at the top. However, a loop that alternately calls cWrite to append to two different strings, such as:

```
        r := ""; s := "";
        FOR i := 1 UPTO n DOB
            cWrite(r,char1[i]); cWrite(s,char2[i]) END;
```

is inefficient in its use of string space. For example, for n = 5, char1 initialized to 'a','b','c','d','e', and char2 initialized to '1','2','3','4','5', string space will be used as follows:

```
                    +------------------------+
                    |xxxxxxxxxxxxxxxxxxxxxxxx|
                    |xxxxxxxxxxxxxxxxxxxxxxxx|
                    |xxxxxxxxxxxxxxxxxxxxxxxx|
                    |xxxa1ab12abc123abcd1234|
                    |abcde12345              |
                    |                        |
                    |                        |
                    +------------------------+
```

Each iteration had to recopy r or s to the top of string space before adding the next character. It would be far better to use separate loops:

```
        r := ""; FOR i := 1 UPTO n DO cWrite(r,char1[i]);
        s := ""; FOR i := 1 UPTO n DO cWrite(s,char2[i]);
```

This would use only 10 characters of string space, rather than 30:

```
                    +------------------------+
                    |xxxxxxxxxxxxxxxxxxxxxxxx|
                    |xxxxxxxxxxxxxxxxxxxxxxxx|
                    |xxxxxxxxxxxxxxxxxxxxxxxx|
                    |xxxabcde12345           |
                    |                        |
                    |                        |
                    |                        |
                    +------------------------+
```

2.3.3.10. rcWrite(s,i)

Put the character i at the top of string space, then copy s to the new top of string space, immediately after i.

rcWrite is inefficient in its use of string space, since it must copy s after i (unless s is zero). Consider the following:

```
s := ""; FOR i := 1 UPTO n DO cWrite(s,char[i]);
r := ""; FOR i := n DOWNTO 1 DO rcWrite(r,char[i]);
```

Though both lines build the same string, the first line uses 5 characters of string space, while the second uses 15:

```
+----------------------+
|xxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxx|
|xxxxxxxxxxxxxxxxxxxxxx|
|xxxedecdebcdeabcde    |
|                      |
|                      |
|                      |
+----------------------+
```

2.3.3.11. Areas

Most of the above procedures take an optional area parameter. String space is really maintained on a per-area basis, meaning each area has its own string space (which has its own top). When an area is specified to these procedures, string text is put at the top of that area's string space, if string text is generated.

2.3.4. String Space and String Garbage Collection

Actually, there can be any number of string spaces in an area. An area starts out with just one. At any one time there is a "current" string space that is in use in that area. When it fills up, MAINSAIL checks to see whether there is another string space in the area with enough room; if not, a new string space is created, or a garbage collection occurs to attempt to reclaim space in the existing string spaces. Whether or not to garbage collect is based on various parameters that monitor how much garbage has already been collected, and on the predicted success of a garbage collection in reclaiming string space.

Conceptually, string garbage collection is straightforward. For the algorithm, consider all string spaces in an area to be concatenated to form one large string space. There are three phases:

1. Mark accessible characters: for each string descriptor, mark the characters that it references. This can be visualized as turning on a "mark bit" in each referenced character, though in practice the mark bits are stored elsewhere.

2. Update string descriptors: once again find each string descriptor, as in step 1. Count how many characters preceding the string have a cleared mark bit, say n. Displace

the charadr part of the string by -n. This adjusts each string descriptor to reference the charadr to which the string will be moved after step 3.

3. Compress string space: compress all marked characters towards the base of string space, squeezing out unmarked characters. Clear the mark bits.

It is possible to find each string descriptor because the MAINSAIL runtime data structures are self-descriptive; each data structure provides a way to locate each of its string descriptors. Each record has an implicit pointer to a class descriptor that describes the type of each field; each array has an implicit pointer to an array descriptor that gives the type of the array; each procedure frame has a descriptor that indicates the location of the strings, and so forth.

In practice, several "tricks" are used to avoid actually examining all previous characters to check the mark bits. Also, there is a complication in that a string must not be moved so that it spans two string spaces.

No matter what tricks are used in the above algorithm, it is time-consuming since every accessible string descriptor and character is manipulated several times. Furthermore, the memory accesses tend to touch randomly a large number of memory pages, which can cause a virtual memory system to "thrash" as real pages are brought into memory.

2.3.5. String Space Control

Using the string space facilities of areas, the programmer can indicate which area's string space is to be used by a string-producing operation. All an area's string space can be discarded at once when the programmer knows that its text is no longer needed. For example, the MAINSAIL compiler can read the source files into a particular area, then clear that area's string space (or dispose the area) when the compilation is finished. This immediately reclaims the occupied memory without the need for garbage collection.

The ability to suppress garbage collection of specified areas is provided. This is useful if the programmer knows that (almost) all characters in a particular area's string space are accessible, since the garbage collector can avoid marking and examining characters in the string space.

MAINPM's "SPACE" command can help keep track of the use of string space during a program's execution; see the "MAINPM User's Guide" for details.

3. Implementation Note on Exceptions and Stack Frames

An understanding of how a language or runtime feature is implemented can facilitate its use. It must be understood that the description given here is not guaranteed to remain valid; the user must not write code that depends upon exceptions being implemented as described here. Even if this description is not rigorous, or becomes invalid with time, it should still shed light on the mechanisms behind the scenes, and thus show the way towards the proper use of exceptions.

3.1. Handle Statement

Consider the following Handle Statement, where the si are statements:

```
s0;
$HANDLE s1 $WITH s2;
s3
```

The assembly code generated for a typical machine looks something like Example 3.1-1.

```
    s0                          # code for s0
    s1                          # code for s1
    BRANCH   x                  # branch to s3
    s2                          # code for s2
    PUSH     #1                 # number of handlers to exit:
                                # 1
    KERCALL  $handlerExit       # MAINSAIL kernel interface
                                # procedure call
x:  s3                          # code for s3
```

Example 3.1-1. Typical Assembly Code Generated for a Handle Statement

A Handle Statement causes just one extra instruction to be executed when no exception occurs, namely the BRANCH around the handler statement s2 and the handler exit code.

s2 can be given control only if an exception occurs in s1 (or a procedure called from s1, to any level). If execution "falls through" the handler statement s2, then it falls into the instructions (PUSH, KERCALL) that invoke "$handlerExit(1)". The argument, 1, means to exit one handler. Every "exit" from s2 (including the "fall through" exit shown here) first calls

"$handlerExit(n)", where n is the appropriate number of handlers to exit. An example of n > 1, i.e., exiting more than one handler, is depicted in Example 3.1-2.

```
DOB ...
    $HANDLEB
        ...
        s0;
        $HANDLE s1 $WITH s2;
        s3;
        ...
        END
    $WITH s4;
    ...
    END
```

Example 3.1-2. Exiting More Than One Handler

If the statement s2 is a Done or Return Statement, then two Handle Statements are exited. $handlerExit needs to know how many handlers to exit so it can properly "unwind" the stack as described below.

3.2. Stack Frames

An understanding of what happens when an exception occurs requires some understanding of MAINSAIL stack frames. Every procedure allocates (pushes) a stack frame on top of the stack upon entry, and deallocates (pops) it upon exit. Each stack frame contains parameters, local variables, and a frame header. The frame header for a frame X contains the caller's return address (X.rtnAdr), the address of the caller's frame header (X.rtnFp), a pointer to the current data section (X.db), and a frame descriptor (X.dscr) that gives some format information about the frame.

Assuming that the stack grows towards low memory (upwards towards top of page), Example 3.2-1 shows what a snapshot of the stack might look like after a procedure A has called a procedure B and B has called a procedure C.

SP is the stack pointer register, which points to the top of the stack. FP is the frame pointer register, which points to the frame header of the currently executing procedure. As described below, FP does not always point to the top stack frame.

```
            low memory

SP --> +-----------+
FP --> |   C       |
       +-----------+
       |   B       |
       +-----------+
       |   A       |
       +-----------+
       |           |

            high memory
```

Example 3.2-1. Snapshot of Stack Growing towards Low Memory

3.3. An Exception Occurs

A procedure A has called a procedure B, which has in turn has called a procedure C. Suppose an exception occurs in C. This exception can be either an explicit call to $raise, a hardware exception such as overflow or divde-by-zero, or a software exception such as a subscript error or nullPointer access. In all cases, MAINSAIL makes it appear that a call has been made from the point of the exception to the procedure $raise (implemented as a kernel interface procedure). The stack looks as in Example 3.3-1 upon entry to $raise:

```
SP --> +-----------+
FP --> |   $raise  |
       +-----------+
       |   C       |
       +-----------+
       |   B       |
       +-----------+
       |   A       |
       +-----------+
       |           |
```

Example 3.3-1. Stack's Appearance upon Entry to $raise

$raise searches the stack frames, in the order C, B, A, etc., looking for an active Handle Statement. A Handle Statement is active if it encloses code that would be executed upon return from $raise, or C, or B, or A, etc. That is, if the flow of control has entered s1 in "$HANDLE s1 $WITH s2", and has not yet exited, then the Handle Statement is active.

3.4. Finding an Active Handler

How does $raise find an an active handler? The frame descriptor contains a bit, $hasHandlerBit, which is set if and only if the corresponding procedure contains a Handle Statement. $raise skips over frames until it finds one with $hasHandlerBit set, thereby quickly ignoring frames that have no Handle Statement.

Once $raise finds such a procedure frame, it must then determine whether there is an active Handle Statement in the procedure. A procedure may contain many Handle Statements, but it could be the case that none, some, or all of them are currently active. Two or more Handle Statements can be active for a given exception only if they are nested. For example, consider the example in Example 3.4-1.

The procedure foo has five Handle Statements. If an exception occurs in s0, s2, s4, or s9 there are no active handlers in foo; if in s1, s3, or s8 there is one; if in s5 or s7 there are two; and if in s6 there are three.

To determine whether there is an active handler, $raise consults the handler table that the code generator puts into each object module. For each Handle Statement "$HANDLE s1 $WITH s2" in the module, the handler table gives the displacements (from the base of the control section) to s1 and s2. $raise converts the return address (in the previous frame) into a return displacement (from the start of the module). It then searches the handler table to see if the return displacement falls within s1. The handler table is ordered in ascending value of the s2 displacements, so that nested handlers are examined from the inside out.

For example, consider the frame for B. The return address to B is in the frame for C. B.db points to the data section, which in turn points to the module descriptor (a record allocated for each control section), which in turn points to the control section itself. $raise subtracts the address of B's control section from the return address to B to get the return address displacement. It then searches the handler table contained in B's control section to see if there is an active handler in B.

If there is an active handler, for example in B, the frame pointer register FP is pointed at B's frame header and control is transferred to the handler at:

```
    (address of control section) +
        (handler displacement obtained from the handler table)
```

At this point the stack looks like Example 3.4-2.

```
PROCEDURE foo;
BEGIN
...
s0;
...
$HANDLE s1 $WITH s2;
...
$HANDLE s3 $WITH s4;
...
$HANDLEB
    ...
    $HANDLE s5
    $WITHB
        ...
        $HANDLE s6 $WITH s7;
        ...
        END
    s8;
    END
$WITH s9;
...
END
```

Example 3.4-1. Nested Handle Statements

```
SP --> +-----------+
       |   $raise  |
       +-----------+
       |   C       |
       +-----------+
FP --> |   B       |
       +-----------+
       |   A       |
       +-----------+
       |           |
```

Example 3.4-2. After Control Is Transferred to the Handler

Note that the frame B for the currently executing procedure is not the top stack frame. (Unfortunately, some computer architectures assume that the currently executing frame is always the top stack frame, causing XIDAK's code generator writers to resort to a certain amount of fakery to get the effect shown above.) If B calls a procedure D, the stack looks like Example 3.4-3.

```
SP --> +-----------+
FP --> |  D        |
       +-----------+
       |  $raise   |
       +-----------+
       |  C        |
       +-----------+
       |  B        |
       +-----------+
       |  A        |
       +-----------+
       |           |
```

Example 3.4-3. After B Calls D

In this case, both D.rtnFp and $raise.rtnFp point at B. Also, D.rtnAdr and $raise.rtnAdr both return to B, though to different places.

3.5. Termination of the Handler

The handler statement s2 can be "terminated" in several ways:

- s2 can "handle" the exception by either "falling through", or by means of a Done, Continue, or Return Statement.

- s2 can "propagate" the exception by calling $raise (no arguments).

- s2 can call $raiseReturn to return from the exception.

Another exception can occur while s2 is active, and be handled by a procedure at or before the frame for A.

If the exception is handled (first case), the stack is "unwound" to make B the top stack frame before execution is continued (see Example 3.5-1).

```
SP --> +-----------+
FP --> |     B     |
       +-----------+
       |     A     |
       +-----------+
       |           |
```

Example 3.5-1. After the Stack Is Unwound

If the exception is propagated (second case) by calling $raise (no arguments), then rather than pushing a new frame for $raise onto the stack, MAINSAIL realizes that an exception is being propagated and continues execution in the current invocation of $raise, and the search for another handler resumes. This search continues looking in the same handler table, since there may be another Handle Statement that encloses the one that just propagated the exception. If one is not found, it goes to the next frame, and so forth.

If s2 calls $raiseReturn (third case), control returns from the call to $raise that caused the exception, and the stack looks just as it did prior to the call to $raise (see Example 3.5-2).

```
SP --> +-----------+
FP --> |     C     |
       +-----------+
       |     B     |
       +-----------+
       |     A     |
       +-----------+
       |           |
```

Example 3.5-2. After $raiseReturn Is Called

Such continuation is allowed only for exceptions caused by an explicit call to $raise, since some computer architectures do not provide enough information to proceed from a hardware exception such as arithmetic overflow.

If another exception occurs in s2 or in a procedure invoked from s2, to any depth (fourth case), then the entire mechanism described here occurs recursively. For example, suppose the handler s2 in B calls D, which calls E, and then another exception occurs in E. The stacks looks as in Example 3.5-3.

```
SP --> +-----------+
FP --> |   $raise  |
       +-----------+
       |   E       |
       +-----------+
       |   D       |
       +-----------+
       |   $raise  |
       +-----------+
       |   C       |
       +-----------+
       |   B       |
       +-----------+
       |   A       |
       +-----------+
       |           |
```

Example 3.5-3. After an Exception Occurs in E

In Example 3.5-3, E.rtnFp points to D, and D.rtnFp points to B. Thus the rtnFp's do not always point to the physically previous frame. $raise uses the rtnFp field to search E, D, B, A, etc., looking for a handler. If the exception is handled by a handler in A, the stack is unwound to be that shown in Example 3.5-4, and the previous exception is simultaneously handled. If instead the exception is handled within D, then the stack is unwound to D, and eventually D returns to B as if no other exception had occurred, and execution of the handler in B continues.

```
SP --> +-----------+
FP --> |   A       |
       +-----------+
       |           |
```

Example 3.5-4. Unwinding the Stack All the Way to A's Frame

If the active handler search falls off the end of the stack, then either an error occurs ("No handler for exception ..."), or a $raiseReturn is simulated (this occurs if the $returnIfNoHandler bit is set in the ctrlBits argument to $raise; in this case, the $noHandler bit is set in the resultBits produces argument of $raise to let the $raise caller know this happened).

3.6. Exception Information

All information about an active exception is contained in the stack frame for the invocation of $raise that raised the exception. The kernel interface variable $raiseFrame points to the frame for the most recent invocation of $raise (corresponding to the current exception). For example, the value of $exceptionName is obtained from a variable local to the frame referenced by $raiseFrame ("" if $raiseFrame = Zero).

The $raise frames are linked together from most recent to least recent. MAINSAIL can determine whether there is an active exception by checking whether or not $raiseFrame is Zero.

It is important that the exception information is stored in a stack frame rather than in a dynamically allocated record since exceptions can occur at times (such as during memory management "critical sections") when record allocation is not possible.

3.7. Coroutines

All of the above descriptions assume there is just one coroutine. Since each coroutine has its own stack, the mechanism must be generalized to cope with situations such as exceptions being raised in one coroutine, but handled in another. The details are not described here; let us know if you would like to know more about how exceptions interact with coroutines.

3.8. Summary

In summary, the exception implementation involves a lot of stack traversal, plus some supporting information in the control sections. The implementation has been careful to support recursive exceptions properly (an exception during the handling of another exception). If there are not many stack frames, the overhead to raise and handle an exception is small. There is almost no overhead for Handle Statements in the absence of exceptions. The size of an object module is slightly increased by the handler table, which contains two long integers for each Handle Statement.

The implementation is dynamic in the sense that the decision as to whether a handler will handle a particular exception is determined at runtime rather than at compiletime as with some languages; this is because MAINSAIL exceptions are dynamically determined strings rather than entities defined at compiletime.

Each active handler is given control in turn, and decides whether it wants to handle the exception by examining values such as $exceptionName. Since the pending stack frames are not discarded when a handler is given control (and hence the handler may be executing out of a frame that is deep on the stack), a handler may decide to continue execution (of some types of exception) at the point of the exception by means of $raiseReturn, a feature also lacking in many other exception implementations.

4. Debugging Hints

This chapter contains some suggestions on the use of MAINDEBUG. Everything described herein works under the current version of MAINSAIL, but is subject to change in future releases.

4.1. Solving Debugger Confusion with Source Text

4.1.1. The Problem

Many MAINSAIL programmers have encountered a situation where the debugger seems to be confused about the position of statements in the source text. The symptoms are usually as follows:

- An error occurs in a module and the programmer edits and recompiles the module.
- The program that references the newly fixed module is re-executed, but although the debugger can find the modified source text, the position of the cursor is incorrect.

Such problems are most often due to the inadvertent use of an outdated intmod or objmod. This typically occurs under the following conditions:

- The source code for a module M is compiled to produce an intmod IM and an objmod OM.
- A program that references M is executed.
- During the course of program execution, MAINSAIL brings the objmod OM and/or the intmod IM into memory.
- The program terminates execution.
- The module M is edited and recompiled to produce a new objmod OM* and a new intmod IM*.
- A program that references M is executed.

- MAINSAIL determines that M's objmod OM is already in memory, or that IM is already open, so it uses the old version rather than bringing OM* or IM* into memory.

- The debugger is invoked during program execution and context is set to module M.

- The debugger finds the modified source text, but since it is using OM instead of OM* or IM instead of IM*, and since the debug information recorded in OM or IM is out of date, the cursor placement is wrong.

4.1.2. The Solution

The problem is that MAINSAIL has no way of knowing that OM is no longer the current object module. In general, objmods stay in memory and intmods stay open and are reused, unless they are explicitly gotten rid of. For example, if the compiler is invoked, and some other programs are run, and then the compiler is used again, it is likely that the compiler modules are still in memory and hence do not need to be loaded again. It is the user's responsibility to ensure that outdated modules are disposed of. The debugger's "-M" command can be used to dispose of old objmods and close old intmods when modules have to be recompiled. Exiting MAINSAIL and then running MAINSAIL again has the effect of disposing of all modules and closing all intmods, and hence gets rid of outdated modules.

4.2. Solving Non-Deterministic Errors and Garbage Collector Errors

4.2.1. The Problem

Many MAINSAIL programmers have encountered bugs with these symptoms:

- An error occurs: the "garbage collector has detected an error" message, or some sort of processor-dependent memory error, unexpected arithmetic error, or perhaps some other exception.

- When the program is re-executed, the error disappears, or appears at a different point in the program, or manifests itself differently, making it very difficult to track.

Frequently, such problems are due to improper use of the system procedure "dispose". Use of misclassified pointers and of the low-level (charadr and address) primitives in such a way as to overwrite MAINSAIL data structures can have the same effect. Another possibility for some types of error is an uninitialized local variable. More rarely, the problems are due to bugs in the MAINSAIL code generator or runtime system, or even in the operating system on which MAINSAIL is running.

Problems with dispose occur in the following way:

1. The program has two or more pointers to the same data structure (array, record, or data section);

2. The data structure is disposed, using one of the pointers;

3. Then a component (field or element) of the disposed data structure is accessed using another pointer to it. Since the memory formerly occupied by the component of the disposed data structure may have been reallocated, this access has undefined effects, possibly including:

   - Getting or assigning an invalid value for an arithmetic component. On some operating systems, using an invalid floating point value can cause an arithmetic exception, so this is one possible symptom of the bug; alternatively, an unexpected value for a (long) integer could cause integer overflow or division by zero.

   - Storing into a memory location used by the memory manager to keep track of the structure of memory. This can cause the collector to report an error at the time of the next collection (or perhaps earlier, at the time of allocation or disposal of some other data structure). If the next collection does not occur for a long time, the context in which the bug occurred may have disappeared, and the bug will appear to come from "out of nowhere".

   - Storing into a pointer, address, or charadr component of some other data structure. If the component so altered does not represent a valid memory address, a processor-dependent memory error may occur the next time it is used. If the component does represent a valid address, it in turn can be used to load from or store into some unpredictable location in memory, which may damage the structure of memory or overwrite some component of yet another data structure. In this way, the manifestation of the bug can become far removed from its source.

The bug may not be repeatable because the input has changed. Try to run the program with EXACTLY the same input as the run that elicited the bug.

Since the bug may occur a long time before the garbage collector detects the problem you would like a way to "run the program backwards" from the point where the garbage collector issues its message back to the point where the problem is found. Needless to say, MAINSAIL doesn't allow you to run a program backwards, but you can use the MAINDEBUG's count break facility (see the "MAINDEBUG User's Guide") to provide a similar capability. The module CONCHK (see the "MAINSAIL Utilities User's Guide") checks for some of the same error conditions the garbage collector checks for, but without actually doing a garbage collection.

```
BEGIN "dspBug"

CLASS c (INTEGER i);

POINTER(c) p1,p2;

INITIAL PROCEDURE;
BEGIN
p1 := p2 := new(c);        # Both pointers point to the same
                           # record
dispose(p1);               # The record is now gone
p2.i := 1;                 # This has undefined effects; p2
                           # is pointing to a disposed record
END;

END "dspBug"
```

Example 4.2.1-1. An Improper Access to a Disposed Record

Assuming the error has been made reproducible (otherwise, see Section 4.2.2), the following steps should help you track it down:

- Compile the relevant modules of the program with the "DEBUG" compiler option.
- Run the program using the new bootstrap (the bug may "go away" when the debuggable version of the module is executed; see Section 4.2.2).
- If the error occurs and gives an "Error response" prompt, enter MAINDEBUG (using the response "DEBUG").
- Note the count break where the error occurred (use the MAINDEBUG command "V #CNT").
- Do a binary search for the procedure in which the bug occurs:

  1. Run the debugger (although running the debugger first may cause the bug to "go away"; see Section 4.2.2).
  2. Set the new count break for some value that is approximately half of the value of the count break where the error occurred (e.g., if the original error occurred with a count break of 500, set the new count break to 250).

3. Execute the program.

4. Invoke CONCHK ("E CONCHK") when the count break is reached.

5. If CONCHK doesn't report an error then set the count break to a value that is halfway between the current count and the count at which the error occurred, continue program execution, and go to step 4.

6. If CONCHK reports an error then go to step 1, but at step 2 use a count break that is halfway between the highest count break at which there was no error and the lowest count break at which there was an error.

- Once the procedure in which the bug occurs is isolated, re-run the program, setting the count break to one less than that which gives the error. Then single step, invoking CONCHK at each statement to determine which one is causing the problem.

This type of binary search is a useful debugging tool for tracking many other kinds of bugs.

4.2.2. If the Bug Is Still Non-Deterministic or Goes Away When You Try to Use the Debugger

It may be that you cannot get the bug to reproduce consistently from execution to execution. In this case, the problem is usually either:

- An uninitialized local variable (on a system where the procedure stack is not cleared before each program execution), or

- A change outside MAINSAIL's direct control in its execution environment, but affecting the run of the program, e.g., a change in the layout of memory into which MAINSAIL is loaded, a change in the file system, affecting the sequence in which MAINSAIL performs file I/O calls, different behavior in another process with which the MAINSAIL process is communicating, or perhaps even a change in the time of day (if time-dependent code is executed by the program), or

- A bug in the operating system.

If you can, eliminate any changes in the execution environment; e.g., delete any files that might be opened by the program and have been created since the bug first appeared, or if the state of memory is dependent on the state of your login session, log out and log in again each time you try to reproduce the bug.

If the bug continues to occur in a non-deterministic fashion, or if the bug goes away when you compile with the "DEBUG" option or attempt to use the debugger, it may still be beneficial to run the program with CONCHK if the bug manifests itself as an address error or garbage collection error. Execute the program with the debugger, running CONCHK at arbitrary intervals. If it does detect an error, it may provide valuable clues on where to look, even if you can't zero in on the bug immediately.

You may wish to instrument your program with calls to "write", or add procedures to verify the consistency of the state of your program. These methods may sometimes disturb the state of execution less than compiling your modules with the "DEBUG" option or attempting to use MAINDEBUG, and may give clues as to the location of the bug.

If a bug in the operating system is suspected, it is best to try to reproduce the problem in a language other than MAINSAIL, and then (if the bug occurs in the other language), report the problem to the manufacturer. XIDAK personnel may be of assistance in this endeavor.

If all else has failed, it may be necessary to use your system's instruction-level debugger. If you are familiar with this debugger, you may be able to use the kernel's $modTimer field in a fashion analagous to the use of the debugger's count break (contact XIDAK for directions on how to do this). Otherwise, report the problem to XIDAK (including as many details as possible, and reducing the offending program to as small an example as possible), and we will try to track it.

4.3. Determining Where MAINSAIL Is in an Infinite Loop

On some operating systems, there is a command to send an operating system exception to a program. For example, on UNIX, you can issue the shell command:

```
kill -SEGV <process id>
```

to simulate a segmentation violation (one of the UNIX signals caught by MAINSAIL) in the process with process number <process id> (use the UNIX "ps" command to determine the process number of your MAINSAIL program). This can be used (sometimes) to stop a MAINSAIL program in an infinite loop and determine what the program was doing when the segmentation violation was simulated. If all goes well, MAINSAIL traps the segmentation violation and raises the appropriate exception; if the program permits, an "error response" prompt is written, and the "CALLS" response may be used to examine the execution stack. This is an unreliable technique, however; there are many critical sections in a MAINSAIL program where the raising of an excpetion is impossible. If you simulate the operating system exception in a critical section, you may well not reach the "error response" prompt, but instead do further damage to the MAINSAIL process (perhaps causing the process to abort or to hang in such a way that it is difficult to kill).

On UNIX, you can actually use the "K" response to the prompt given when SIGQUIT (usually CTRL-\) is caught instead of actually issuing a "kill" command; this is a little more convenient.

The same technique may be used on non-UNIX systems if the operating system provides a command to simulate one of the operating system exceptions caught by MAINSAIL. Check the operating system command guide and the appropriate system-specific MAINSAIL documentation for details.

4.4. Displaying the Contents of a Structure

The debugger's "A" and ".V" commands are useful for displaying the contents of an array or record, but it is sometimes useful to display an entire structure with a single command and then browse through the output. No such command exists in the debugger itself; however, those users licensed to run the Structure Blaster may display a structure by calling $structureWrite. For example, if a structure has its root at a pointer p, the debugger command:

```
XS $structureWrite(logFile,p)
```

writes the entire structure as a text form to logFile. The usual $structureWrite caveats apply. Be aware that writing a large structure to a terminal may take a long time; a temporary disk file "struct.txt" can be used instead with the command sequence:

```
.D POINTER(textFile) f;
XS open(f,"struct.txt",create!output);
XS $structureWrite(f,p); close(f);
```

The contents of "struct.txt" then describe the structure pointed to by p.

5. Common Pitfalls and Portability Problems

This chapter describes some common errors in writing MAINSAIL programs. Portability errors are an important category of error; a program that compiles and executes correctly on one machine can fail to compile for or execute properly on another machine if it has been written based on non-portable assumptions.

5.1. Character Set Assumptions

MAINSAIL does not assume that the ASCII character set is used; indeed, MAINSAIL implementations exist for EBCDIC machines. The guaranteed MAINSAIL character set does include all the printing characters in the ASCII character set, but there are printing characters in the EBCDIC character set that do not correspond to ASCII characters. These characters should not be used in MAINSAIL source programs (since the source programs cannot then be moved from the EBCDIC machine), nor should a portable MAINSAIL program assume that these characters can be generated or interpreted by peripherals.

A common pitfall on programs that are to run on both ASCII and EBCDIC machines is to assume that the alphabetic characters are contiguous within the character set. This is true of ASCII, but not of EBCDIC, which has gaps in the middle of the alphabet. MAINSAIL guarantees that the character codes for the digits '0' through '9' are contiguous (e.g., '0' + 1 = '1'), but not the letters. This test to see whether ch is an alphabetic character works on an ASCII machine:

```
'A' LEQ ch LEQ 'Z'
```

but not on an EBCDIC one, since non-alphabetic characters may lie in the range 'A' to 'Z'. The correct test for an alphabetic character is:

```
isAlpha(ch)
```

which works on both ASCII and EBCDIC machines. The correct way to find the next alphabetic character after ch is not:

```
ch + 1
```

but rather:

```
nextAlpha(ch)
```

Other procedures that allow character-set portability between ASCII and EBCDIC machines (and any other character set, in the unlikely event that such a character set is encountered) are cvu, cvl, isLowerCase, isUpperCase, prevAlpha, and isNul. Always use these procedures when appropriate rather than hardwiring assumptions about the ordering of character codes within the character set.

5.2. End-of-File, eof, and $gotValue

On some operating systems, programs cannot detect the exact end-of-file position, because the operating system does not record sufficient information about the file. On other operating systems, the exact end-of-file position is detectable; however, "eof(f)" may become true either immediately before or immediately after an attempt is made to read a datum beyond the end of the file, depending on the form of the operating system's I/O calls, and perhaps on the format of the file itself.

The use of the system procedure "eof" for detecting the end of a file is no longer encouraged. The system procedure $gotValue, available in Version 11 and subsequent versions of MAINSAIL, provides a more uniform behavior on all files on which the exact end-of-file position can be detected.

"$gotValue(f)" returns false after some form of the system procedure "read" has been issued on f, and that read failed because end-of-file had been reached. $gotValue is not set by procedures other than read, since other input procedures (e.g., fldRead, cRead, scan, etc.) return a distinctive value when end-of-file is reached. The moment at which $gotValue becomes true is better defined than the moment at which eof becomes true.

Whether $gotValue or eof is used, it is still the case on some operating systems that the end-of-file position can be detected only imprecisely; e.g., some operating systems store the end-of-file position rounded up to the nearest page. On such systems, $gotValue will not become false until after the entire last page of the file has been read. For this reason, it is best for a program, if possible, to structure a file so that its end position is apparent from the data in it, rather than depending on $gotValue or eof.

5.3. File Name Syntax

All that MAINSAIL guarantees about file names is that the operating systems on which it runs will support file names that correspond to an identifier (case may or may not be distinguished) of six or fewer characters. In practice, most operating systems permit longer identifiers (up to nine characters), and also permit an extension of up to three characters, separated from the base file name by a period. Assumptions that longer file names, multiple extensions, and non-identifier characters are permitted can lead to trouble when an application is moved from one machine to another. In particular, directory, volume, or device syntax in file names varies considerably from operating system to operating system.

MAINSAIL provides logical file names and searchpaths in order to make it possible to write system-independent applications that use files. A recommended use of searchpaths is to prefix the names of related files in a program with a common prefix, then (either in the bootstrap or at runtime) issue MAINEX subcommands to establish the correspondence between the prefixes used by the program and the actual location in the local file system of the files used. For example, if a program uses two initialization files and three data files, it might use the names:

```
init#init1
init#init2
data#data1
data#data2
data#data3
```

In the subcommands in the bootstrap, "SEARCHPATH" commands could define the "init#" files to reside on a directory "/foo/init" (in a UNIX-like file system), and the "data#" files on "/bar/data":

```
SEARCHPATH init#* /foo/init/*
SEARCHPATH data#* /bar/data/*
```

For a strategy like this one to work on all machines, the names prefixed with "init#" and "data#" must still be identifiers of no more than six characters, since this part of the file name is being used as the operating system's "leaf" file name.

5.4. Case Sensitivity in File Names

On some operating systems, file names are case-sensitive; i.e., "file1" names a different file from "File1" or "FILE1". On other systems, all three of these names designate the same file. Programs should be consistent about file name case for portability between case-sensitive and case-insensitive file systems; e.g., don't use "FOO" as a file name in one place in a program (or collection of related programs) and "foo" in another place referring to the same file. If necessary, a program can determine whether it is running on an operating system with case-sensitive file names by:

```
$attributes TST $fileNamesAreCaseSensitive
```

5.5. Random Access to Files

It is not guaranteed that a file is in a format that can be accessed for random output unless it was created by a MAINSAIL program with the "random" open bit set. On some operating systems (e.g., VAX/VMS), the default text file format is a record-oriented file that cannot be opened for random output (this is for compatibility with other VAX/VMS programs). It is a common error to create a file by writing it sequentially (without opening it for random access), then to close it and reopen it, expecting to be able to perform random output to the file.

Even when a random file is opened input-only (which is possible on the VAX/VMS-style record-oriented file), it may be a good idea to create the file with the random bit set. The reason is that the mechanism for random input to a record-oriented file is slow and consumes a great deal of memory. In sum, the rule of thumb is:

```
If a file is ever to be opened for random access,
create it with the random bit set in the call to "open".
```

5.6. Simultaneously Opening the Same File Several Times

The same operating system file may be reliably opened by more than one process, or several times by the same process, only if the file is opened input-only each time. Some operating systems do not generate an error when one program is writing and another reading or writing the same file, but unless special (operating-system-dependent) synchronization methods are used among the simultaneous readers and writers, the values read from or written to the file may be invalid or logically inconsistent.

If two MAINSAIL programs are reading from and writing to the same file at the same time (not possible on many operating systems if the files are opened with the system procedure "open"), it is not sufficient for one program to write a datum in order to make it show up in the file the other program is reading. The MAINSAIL runtime system buffers files; modifications to a file may be delayed until the file is closed.

There is, at present, no portable way in MAINSAIL to provide synchronization for simultaneous read and write accesses to a large body of information. XIDAK is developing database software for this purpose and expects this software to be available in the near future.

5.7. Number of File Handles

Some operating systems place tight restrictions on the number of files a process may have open simultaneously. Programmers should not write programs that require large numbers of simultaneously open files, because such programs may fail to open some of the files on some operating systems.

The LIB device module, described in the "MAINSAIL Utilities User's Guide", can be used to reduce the number of operating system file handles needed to access a group of files.

5.8. Redirection of Input and Output

cmdFile and logFile are redirectable from within MAINSAIL. Some operating systems also permit the primary input and output files of a MAINSAIL process (i.e., "TTY") to be redirected, but this facility is not available on all systems. A portable program of which the input is to be redirected should avoid ttyRead and ttyWrite, using instead "read(cmdFile,...)" and "write(logFile,...)".

5.9. Assignment to cmdFile and logFile

The redirection of cmdFile and logFile is a perfectly standard thing to do; i.e., a program may assign to the system variables cmdFile and logFile without fear of "messing up" MAINSAIL. However, if a program does not thereafter reassign the old cmdFile and/or logFile pointers back to logFile and/or cmdFile, it needs to be sure to close the old cmdFile and/or logFile.

5.10. Reading Non-String Values from a Text File

"read(f,x)", where x is a non-string variable, leaves all characters after the string representation of x in the file, including a terminating eol, if any. Example 8.4-1 of part I of the "MAINSAIL Tutorial" demonstrates the problems that may arise if a user is prompted for a value which is then read using a non-string form of "read". The usual way to read from an interactive file is to read entire lines with the string form of read, then parse the resulting strings.

5.11. Sizes of Data Types

A programmer who works consistently on one machine may find it easy to forget that the sizes of MAINSAIL data types vary from machine to machine. This can lead to several different kinds of portability errors:

- Hardwiring data type sizes, particularly in file positions. For example, to position past the third long integer in a data file, use "setPos(f,cvli(3 * $ioSize(f,longIntegerCode)))", never "setPos(f,3L)" or "setPos(f,12L)".

- Assuming that the sign bit in a number is some particular bit. For example, on two's-complement machines where a long integer occupies 32 bits, the bit pattern for -1L is 'HFFFFFFFFL; however, it is incorrect to assume this bit pattern represents -1L on every machine (on a 64-bit machine, this number would be 2 ^ 32 - 1). The portable way to create a bit pattern in which all bits are set is:

  ```
  lbMask(0,$bitsPerStorageUnit * size(longBitsCode) - 1)
  ```

- Assuming that a set bit will be shifted out of a bits or long bits value if it is shifted left by some hardwired number of bits. On a machine with 16-bit bits values, "'HF000 SHL 4" is '0, but on a machine with 32-bit bits values, it is 'HF0000.

5.12. Logical File Name Formats

Logical file names that redirect one file to another may be established with the "ENTER" subcommand or the procedure enterLogicalName (the searchpath facility provides an alternative mechanism of redirecting file I/O). This facility can be used to suppress or redirect the reading or writing of default files (like the editor's "eparms" file; see Example 9.4.1-7 of part I of the "MAINSAIL Tutorial").

Many XIDAK programs make use of logical names of the form:

```
(the logical name)
```

i.e., some text enclosed in parentheses. Such names are valid to the system procedure open:

```
open(f,"(the logical name)",...)
```

but may confuse many utility programs, which interpret spaces as file name separators; e.g., the MODLIB command:

```
add (the logical name) xxx yyy
```

interprets "(the" as the name of the library to which to add the modules "logical", "name)", "xxx", and "yyy". To avoid such problems, use logical names without spaces, or use the actual file name to which the logical name maps (provided that the actual file name does not contain spaces! XIDAK programs may be confused by such operating system file names as well).

XIDAK recommends that users avoid the parenthesized logical name convention, as it may conflict with future XIDAK logical names. Conventions using other punctuation characters are less likely to run into trouble; e.g., a company could decide to prefix its logical names with underscores:

```
_the_logical_name
```

5.13. Valid Addresses

As described in Section 18.1.6 of part I of the "MAINSAIL Tutorial", some addresses are not valid for storing or retrieving data because of machine alignment requirements. 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.

Section 18.1.6 of part I of the "MAINSAIL Tutorial" also points out that alignment considerations may cause "cvc(cva(c))", where c is a charadr, to be different from c.

5.14. Garbage Collections during $storageUnitRead and $storageUnitWrite

Garbage collections may occur during a call to $storageUnitRead or $storageUnitWrite. Therefore, converting an array to an address before reading data into it or writing data from it by means of these procedures has undefined effects, since the array may be moved during the operation, and the elements loaded into the wrong part of memory. The proper way to do a $storageUnitRead of n elements of type t from a file f into an array ary (starting at the first element) is to convert it to pointer:

```
    $storageUnitRead(f,n * cvli(size(tCode)),cvp(ary),
        lDisplacement(cva(cvp(ary)),$adrOfFirstElement(ary)))
```

where tCode is the type code for type t, i.e., one of integerCode, longIntegerCode, bitsCode, etc.

5.15. Date and Time Addition and Subtraction

The procedures $addToDateAndTime and $dateAndTimeDifference perform arithmetic on dates and times where both date and time are present. Such procedures are not necessary if only a date or only a time is involved; all MAINSAIL date formats are a number of days, and all MAINSAIL time formats are a number of seconds. The long integer "+" and "-" operators can therefore be used, provided that:

- In the case of "+", one operand must be a valid MAINSAIL date or date difference, and the other operand must be a difference.
- In the case of "-", both operands must have the same format (GMT times, local times, or time differences).

For example, if d is a date, "d + 3L" is the date that is three days later.

5.16. Searching for " in the Line-Oriented Debugger Interface

To search for a double quote character in the line-oriented debugger interface, the double quote must appear twice, as in a MAINSAIL string constant; e.g., use:

```
                    "He said, ""What?"""
```

to search for:

```
He said, "What?"
```

5.17. Multiline CONF Commands

Some CONF commands allow arguments of more than one line. To specify a multiline argument, type the command name on a line by itself and argument lines on subsequent lines. Multiline arguments are terminated with a single blank line (blank lines are not valid in the arguments to any multiline CONF command).

To add new lines to an existing multiline value, type "=" on the first line (see the "MAINSAIL Utilities User's Guide" for examples).

5.18. Including Brackets in Bracketed Text

It is often desirable to include brackets within a macro body. For example, you may want to define a macro body like:

```
p.s := "]"
```

However, if you try to use this macro definition:

```
DEFINE xyz = [p.s := "]"];
```

you run into the problem that the compiler thinks the definition is terminated by the first right square bracket, i.e., the one in quotes (which you wanted to appear within the macro body). The way around this is to concatenate the bracketed text with a string containing the bracket. It is legal, on the right side of a macro equate, to concatentate bracketed text with string constants to produce bracketed text (a piece of bracketed text must be the first thing on the right side of the macro equate, but it may be followed by any number of concatenated strings and pieces of bracketed text). A macro definition with the desired effect is:

```
DEFINE xyz = [p.s := "] & "]" & ["];
```

Section 13.10 of part I of the "MAINSAIL Tutorial" discusses some other common problems with macros.

5.19. Reallocating a Local Own Array

The programmer must be careful not to reallocate inadvertently an own array declared local to a procedure. The proper approach is shown in Example 5.19-1.

```
PROCEDURE p;
BEGIN
OWN STRING ARRAY(1 TO 10) sAry; # OWN so it does not go
                                # away upon procedure exit
...
IF NOT sAry THEN new(sAry);     # allocate it the first
...                             # time
END
```

Example 5.19-1. Allocating an Own Array Only Once

5.20. Final Procedure Order

When a MAINSAIL execution terminates, all data sections are disposed, causing their final procedures to be executed. The order in which the final procedures execute is unspecified. This can lead to problems if, for example, a module A accesses an interface field of a module B in its final procedure. If the interface field is a data field, and B is disposed before A, then A will get an unbound data access when MAINSAIL tries to exit. On the other hand, during another execution, A may be disposed before B, and no problem will occur. If A's final procedure calls an interface procedure of B, and B's final procedure calls an interface procedure of A, an infinite loop may result when MAINSAIL tries to exit, since A's final procedure will bind B, which means that B needs to be disposed, so its final procedure will be called, which will bind A, etc. Problems can also arise if a module disposes a data structure in its final procedure if the data structure may be accessed by the final procedures of other modules.

It is best to avoid the use of final procedures altogether if the logic of a program permits cleanup to be done at the end of the root module's initial procedure. Final procedures are really useful only in modules which, once bound, remain in memory indefinitely, which are potentially used by more than one application (if they are used by only one application, then that application should dispose them when it is finished with them), and which have allocated resources they must free before MAINSAIL exits. When final procedures are used, they must be carefully written.

5.21. Use of $next in Coroutine Scheduling

$thisCoroutine.$next is usually the coroutine that resumed the current one. However, if an application consists of several coroutines, it should not depend on $next to determine which coroutine to schedule next. The reasons are:

- The search for an exception handler rummages through the coroutine list and changes the value of $next. If an unexpected exception occurs, it can set $thisCoroutine.$next to a value that would not be expected from reading the text of the program. Note that when a coroutine calls $resumeCoroutine with the delete bit set, $descendantKilledExcpt is raised before resuming the target coroutine.

- The MAINSAIL runtime system may make use of coroutines. For example, if opening a special kind of file requires the creation and resumption of a coroutine, then $thisCoroutine.$next may be altered after the file is opened and every time I/O is performed to the file.

Programs using several coroutines should always remember which coroutines belong to them, and choose the coroutine to resume based on the logic of the application.

6. Miscellaneous

6.1. Compiletime Libraries

Compiletime libraries (also called "source libraries") are a technique for creating a repository of procedures used by a number of modules. The introduction of intmods has made this technique more or less obsolete, since intmods are more flexible in that they can contain common variables as well as common procedures and they are more efficiently compiled. Nonetheless, users may need to examine or maintain existing compiletime libraries, so this section has been retained.

The compiler directives "NEEDBODY(p)", "NEEDANYBODIES", "NEEDANYBODIES(c)", and "FORWARD(c)" are all provided to promote the use of compiletime libraries. A compiletime library is a file that contains procedure bodies that can be "compiled into" a number of different modules.

The diagrams and text in Example 6.1-1 shows the use of a compiletime library. Each procedure declaration in the file "lib" is surrounded by the directives shown in Example 6.1-2.

```
<eop>
BEGINSCAN "";
IFC NOT NEEDBODY(p) THENC SKIPSCAN ""; ENDC

PROCEDURE p;
BEGIN
... <procedure body> ...
END;

IFC NOT NEEDANYBODIES("lib") THENC DONESCAN; ENDC
<eop>
```

Example 6.1-2. Declaration of Procedures in "lib"

Any procedure declared forward in "hdr", and called in mi (but not given a body there), causes an automatic sourcefile of "lib". In "lib", the conditional compilation and compiler directives result in the compilation of only those procedures which were used (called) in mi. The use of "NEEDANYBODIES" together with "DONESCAN" is an optimization to stop compilation of "lib" when all needed procedure bodies have been obtained from it.

```
header file "hdr"
+--------------------------------+
|macro definitions               |
|class declarations              |
|module declarations             |
|forward procedure declarations  |
|  (each procedure header is     |
|   preceded by FORWARD("lib"))  |
+--------------------------------+

module m1        module m2                module mn
+--------+       +--------+               +--------+
|        |       |        |     . . .     |        |
|        |       |        |               |        |
+--------+       +--------+               +--------+

compiletime library "lib"
+--------------------------------+
|"body" declarations of procs    |
| declared with FORWARD("lib")   |
| in "hdr"                       |
+--------------------------------+

The forward declarations in hdr are qualified
with FORWARD("lib").

Each module mi has the form:

module mi
+--------------------------------+
|BEGIN "mi"                      |
|SOURCEFILE "hdr";               |
|...                             |
|END "mi"                        |
+--------------------------------+
```

Example 6.1-1. Use of a Compiletime Library

To convert a compiletime library to an intmod, go through the file(s) containing the procedure bodies (in this case, "lib") and remove all the directives bracketing each procedure declaration. Add the directives:

```
SAVEON;
$DIRECTIVE "NOGENCODE";
```

to the file "hdr", and add:

```
BEGIN "hdr"
```

and:

```
END "hdr"
```

at the beginning and end of "hdr", respectively. "hdr" is now the source file for an intmod, which can be used (once it has been compiled) by modules that contain:

```
RESTOREFROM "hdr";
```

If any of the procedures requires "private" procedure or outer variable or macro declarations, these may be hidden from the using module with the symbol visibility directives. For example, to hide a symbol "xxx", include the line:

```
$DIRECTIVE "MAKENOTVISIBLE xxx";
```

before the end of "hdr".

Most compiletime libraries can be converted to intmods. Sometimes, however, the procedures in a compiletime library may use identifiers that have been defined before their bodies are picked up from the compiletime library; e.g., a procedure might use an identifier "xxx" that it expects the calling module to have defined. Such procedures with source dependencies cannot be converted to procedures in intmods, since all identifiers in them must be defined when the intmod is compiled.

6.2. MAINED Editing Techniques

6.2.1. Deleting in Insert Mode

If you want to delete some stuff at the end of a line, and replace it with some other stuff, just go into insert mode and press the delete key. This deletes old text to the left.

6.2.2. Search and Replace Macro

To define a search and replace macro:

```
t<search string><eol>
/<macro id>
<do the replacement>t<search string><eol>
/
```

6.2.3. Defining Mode-Independent Macros with "QRM"

A macro execution can leave MAINED in command mode (or some other mode explicitly specified in the macro), or it can leave it in the mode which was in effect when the macro was invoked. The "QRM" command restores the saved mode (remembered at the start of macro execution). For example, to define a macro to search for "foo" and retain the editor mode, issue the following commands:

```
/<macroId>tfoo<eol>qrm<ecm>/
```

The editor mode remains unchanged by the invocation of <macroId>. The "QRM" command works for named as well as for unnamed macros.

6.3. Rationale for MAINSAIL's Binding Rules

Often, interface variables do not really "belong" to any module (they are equally shared by a number of modules) so that it is natural to put them in a single interface. In those cases when some interface variables are naturally associated with a module which would not necessarily bound when the variables are accessed, the programmer must choose between moving those variables to a "root" module that will be bound when the variables are accessed and explicitly binding the module with the interface variables in all modules that access the variables. There are good reasons why MAINSAIL imposes this minor inconvenience instead of automatically binding every module accessed by m when m is allocated or binding a module whenever one of its interface variables is accessed:

- Automatically binding every module accessible from m when m was allocated would lead to an "explosive" allocation of all potentially accessible modules in the entire program, since the allocation of each module would in turn cause the allocation of all its referenced modules. This defeats MAINSAIL's concept of a program as an open-ended collection of modules brought into execution only as necessary. For example, the runtime system contains many modules, perhaps only a few of which are used during a particular program execution. Also, it is common for an initial procedure to allocate data structures such as arrays; it would be potentially fatal to program execution if all such arrays were allocated at once.

- The check required to allocate a module on intermodule data access would make intermodule data access substantially less efficient.

6.4. Selective Compilation of Modules in a File

The techniques described here are most useful for large programs with many modules, where it is convenient to keep the source code for the modules in a single file. The programmer may want to put modules which are somehow "related" in the same file. Or if a program consists of a large number of small modules, rather than having a large number of files (one for each module), the programmer may wish to have a small number of files, each containing a number of the modules.

There are several ways to carry out compilation of a file containing more than one module. Four are discussed here; the programmer can doubtless come up with other approaches using the scanning directives combined with interactive "REDEFINE" directives.

1. If there are no scanning directives in the file, then the modules are compiled one after another; i.e., the compiler compiles all the modules (one after the other, as if each were in the file alone) in a file if it encounters no "DONESCAN" directive.

2. The use of interactive "REDEFINE" and the scanning directives make it possible to compile just one of the modules in the file. This is useful when, for example, all modules but one have been compiled correctly, so that just one needs to be recompiled. An example of this is described in Section 6.4.1.

3. The use of interactive "REDEFINE" and the "SKIPSCAN" and "BEGINSCAN" directives make it possible to ask the user, before each module is encountered (by the compiler) in the file, whether or not that module should be compiled. An example of this is described in Section 6.4.2.

4. A combination of the previous two methods make it possible for the user either to specify just one module to be compiled or to have the compiler ask whether or not each module should be compiled.

6.4.1. Asking Which Module Should Be Compiled

This example shows the use of an interactive "REDEFINE" and the scanning directives to allow the user to specify just one module in a file as the module to be compiled.

If a file "f1" contains a number of modules, two of which are MTHMOD and STRMOD, then "f1" might look as shown in Example 6.4.1-1.

When MAINSAIL starts to compile f1, it encounters the "REDEFINE", and so it outputs to the user's terminal (assuming logFile is associated with tty):

```
Desired module (MTHMOD STRMOD ...):
```

```
REDEFINE scanName "Desired module (MTHMOD STRMOD ...): ";
SKIPSCAN scanName;
<eop>
BEGINSCAN "mthMod";          # first line of a new page
BEGIN "mthMod"
...
END "mthMod"
DONESCAN;
<eop>
BEGINSCAN "strMod";          # first line of new page
BEGIN "strMod"
...
END "strMod"
DONESCAN;

...
```

Example 6.4.1-1. A File That Asks Which Module Should Be Compiled

If the user types "strmod", the effect is the same as if:

```
REDEFINE scanName = "strmod";
```

had occurred in the file instead of the interactive "REDEFINE". Then a "SKIPSCAN "strMod";" is done; the compiler examines the first line of each page until it finds:

```
BEGINSCAN "strMod"
```

Then it compiles the STRMOD module. The "DONESCAN" after the end of STRMOD prevents the compiler from looking any further in the file.

6.4.2. Asking Whether Each Module Should Be Compiled

This sample shows how to allow the user to specify whether or not each module in a file should be compiled.

Before each module in the file is encountered, the user is asked with an interactive "REDEFINE" whether or not that module should be compiled. If the answer is "TRUE", the module is compiled; if not, a "SKIPSCAN """ is done; that is, the compiler skips to the next "BEGINSCAN". In either case, further scanning directives direct the compiler to ask the same question about the next module.

In this case, the file "f1" looks like Example 6.4.2-1, where the last two lines are given to terminate the "SKIPSCAN """ that would be done if the user answered that "xyzMod" was not to be compiled.

```
REDEFINE xxx "Compile mthMod (TRUE or FALSE): ";
IFC NOT xxx THENC SKIPSCAN ""; ENDC

BEGIN "mthMod"
...
END "mthMod"
<eop>
BEGINSCAN "strMod";          # first line of new page

REDEFINE xxx "Compile strMod (TRUE or FALSE): ";
IFC NOT xxx THENC SKIPSCAN ""; ENDC

BEGIN "strMod"
...
END "strMod"
<eop>
BEGINSCAN "xyzMod";          # first line of new page

REDEFINE xxx "Compile xyzMod (TRUE or FALSE): ";
IFC NOT xxx THENC SKIPSCAN ""; ENDC

BEGIN "xyzMod"
...
END "xyzMod"
<eop>
BEGINSCAN "";                # first line of last page
DONESCAN;
```

Example 6.4.2-1. A File That Asks Whether Each Module Should Be Compiled

This method can be combined with the previous method to enable the user to state at the start of compilation whether all modules should be compiled, or whether a prompt should be given for each.

Appendix A. Solutions to Most Exercises

There are no "right" answers to most of the programming problems; any program that works and is easy for the human reader to understand is a good solution. The programs that appear here are therefore just suggestions.

Exercise 2-1 of part I of the "MAINSAIL Tutorial":

String	Module Name	Identifier
ABCDEFG	Illegal, too many characters (6 max).	Legal
1X	Illegal, must start with letter.	Illegal, must start with letter.
A_B_C	Illegal, may not contain underscore.	Illegal, may not contain underscore.
abc	Legal, same as "ABC".	Legal, same as "ABC".

Exercise 2-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "byeBye"

INITIAL PROCEDURE;
BEGIN
write(logFile,"Bye, folks!" & eol);
END;

END "byeBye"
```

Exercise 3-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "writeX"

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

write(logFile,str1 & str2);
END;

END "writeX"
```

Exercise 3-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "vars3"

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

write(logFile,"First number: "); read(cmdFile,s); read(s,i);
write(logFile,"Second number: "); read(cmdFile,s); read(s,j);
write(logFile,"Third number: "); read(cmdFile,s); read(s,k);
write(logFile,"First - 2 = ",i - 2,eol &
    "Second * Third = ",j * k,eol &
    "Third - First = ",k - i,eol);
END;

END "vars3"
```

Exercise 3-3 of part I of the "MAINSAIL Tutorial":

Expression	Answer
`(((5)))`	Legal, integer. Nested parentheses are OK to any depth as long as they are correctly paired ("((((5))" would be illegal). Means the same as 5 all by itself.
`* 62`	Illegal; "*" needs two operands, one before and one after.
`"Time " & " is " & " money."`	Legal, string.
`eol + 4`	Illegal; eol is a string, so is not a legal operand to "+", which operates only on numbers.
`tab & eop & eop`	Legal, string.

Exercise 4-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "vDelta" # Vertical Delta

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

write(logFile,"Height: "); read(cmdFile,s); read(s,count);
FOR i := count DOWNTO 1 DO
    BEGIN
    FOR j := 1 UPTO i - 1 DO write(logFile," ");
    FOR j := 1 UPTO 2 * (count - i) + 1 DO
        write(logFile,"*");
    write(logFile,eol);
    END;
END;

END "vDelta"
```

Exercise 4-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "yOrN" # Answer Yes or No

INITIAL PROCEDURE;
BEGIN
BOOLEAN answerOK;
STRING s;
DOB write(logFile,"YES or NO: "); read(cmdFile,s);
    answerOK :=
        (s = "YES" OR s = "yes" OR s = "NO" OR s = "no");
    IF NOT answerOK THEN write(logFile,
        "Only YES and NO (all uppercase or all lowercase)" &
            eol & "are acceptable answers." & eol);
    END UNTIL answerOK;
END;

END "yOrN"
```

A better method of testing for "YES" or "NO" would be:

```
        equ(s,"YES",upperCase) OR equ(s,"NO",upperCase)
```

but the procedure equ has not been introduced at the point of the exercise. Another strategy permitting prefixes of "YES" or "NO" would involve the use of the system procedure cmdMatch.

Exercise 5-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "colorP" # Colors with procedures

DEFINE red = 1;
DEFINE yellow = 2;
DEFINE blue = 4;

INTEGER PROCEDURE getColor (STRING promptStr);
BEGIN
INTEGER colorNum;
STRING s;
write(logFile,promptStr); read(cmdFile,s);
IF s = "red" THEN colorNum := red
EF s = "yellow" THEN colorNum := yellow
EF s = "blue" THEN colorNum := blue
EB  write(logFile,s,": unknown color, red assumed" & eol);
    colorNum := red END;
RETURN(colorNum);
END;



INITIAL PROCEDURE;
BEGIN
INTEGER color1,color2,totalColor;

color1 := getColor("First paint color: ");
color2 := getColor("Second paint color: ");

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 "colorP"
```

The long If Statement at the end of the initial procedure could be better written as a Case Statement, which is not yet introduced at the point of the exercise. It would look like:

```
        CASE totalColor OFB
            [red + red] write(logFile,"red");
            [red + yellow] write(logFile,"orange");
            [yellow + yellow] write(logFile,"yellow");
            [red + blue] write(logFile,"purple");
            [yellow + blue] write(logFile,"green");
            [blue + blue] write(logFile,"blue");
            END;
```

Exercise 5-2 of part I of the "MAINSAIL Tutorial":

The two code fragments don't necessarily do the same thing. In the first instance, the order of the two calls to getString is known. In the second, since MAINSAIL doesn't guarantee the order in which procedure arguments are evaluated, it is possible that the second call to getString may be performed first, with the result that the prompts appear in the order:

```
Second string:
First string:
```

which is hardly desirable.

Exercise 5-3 of part I of the "MAINSAIL Tutorial":

```
BEGIN "odPrSq" # Odds, primes, and squares

BOOLEAN PROCEDURE isOdd (INTEGER i);
BEGIN
RETURN((i DIV 2) * 2 NEQ i);
END;



BOOLEAN PROCEDURE isPrime (INTEGER i);
# Prime iff no integer between 2 and i - 1 divides i.
BEGIN
INTEGER j;
IF i LEQ 1 THEN RETURN(FALSE);
FOR j := 2 UPTO i - 1 DO
    IF (i DIV j) * j = i THEN RETURN(FALSE);
RETURN(TRUE);
END;
```

```
BOOLEAN PROCEDURE perfectSquare (INTEGER i);
BEGIN
INTEGER j;
IF i = 0 THEN RETURN(TRUE);
j := 1;
WHILE j * j LEQ i DOB
    IF j * j = i THEN RETURN(TRUE);
    j := j + 1 END;
RETURN(FALSE);
END;



INITIAL PROCEDURE;
BEGIN
INTEGER i;
STRING s;
write(logFile,"A nonnegative integer: "); read(cmdFile,s);
read(s,i);
write(logFile,i," is ");
IF isOdd(i) THEN write(logFile,"odd")
EL write(logFile,"even");
write(logFile,"." & eol,i," is ");
IF NOT isPrime(i) THEN write(logFile,"not ");
write(logFile,"prime." & eol,i," is ");
IF NOT perfectSquare(i) THEN write(logFile,"not ");
write(logFile,"a perfect square." & eol);
END;

END "odPrSq"
```

The operator "MOD", not yet introduced at the point of the exercise, allows a more succinct test for whether an integer a is divisible by an integer b:

```
a MOD b = 0
```

so that, e.g., the body of isOdd could be rewritten:

```
RETURN(i MOD 2 NEQ 0)
```

Exercise 6-1 of part I of the "MAINSAIL Tutorial" is easier to do with the addition of another procedure digitRead:

```
INTEGER PROCEDURE digitRead (MODIFIES STRING s);
BEGIN
IF s NEQ "" THEN RETURN(rcRead(s)) EL RETURN('0');
END;



STRING PROCEDURE stringAdd (STRING s,t);
# The strings s and t represent nonnegative integers.
# Return the string that represents their sum.
BEGIN
INTEGER carry,ch;
STRING u;
u := ""; carry := 0;
WHILE s OR t DOB
    carry :=
        digitRead(s) - '0' + digitRead(t) - '0' + carry;
    IF carry GEQ 10 THENB ch := carry - 10; carry := 1 END
    EB ch := carry; carry := 0 END;
    rcWrite(u,ch + '0') END;
IF carry NEQ 0 THEN rcWrite(u,'1'); RETURN(u);
END;
```

Exercise 6-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "rpn2"

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 = "-" OR t = "*" OR t = "/" OR t = "S" OR
    t = "C" OR t = "?" OR t = "" THEN RETURN(t);

# If it isn't "Q", "S", "+", or nothing, then it must be
# an integer or illegal.
u := t;
IF first(u) = '-' THEN cRead(u);
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;
```

```
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;



INTEGER PROCEDURE digitRead (MODIFIES STRING s);
BEGIN
IF s NEQ "" THEN RETURN(rcRead(s)) EL RETURN('0');
END;



STRING PROCEDURE nonnegativeAdd (STRING s,t);
# The strings s and t represent nonnegative integers.
# Return the string that represents their sum.
BEGIN
INTEGER carry,ch;
STRING u;
u := ""; carry := 0;
WHILE s OR t DOB
    carry :=
        digitRead(s) - '0' + digitRead(t) - '0' + carry;
    IF carry GEQ 10 THENB ch := carry - 10; carry := 1 END
    EB ch := carry; carry := 0 END;
    rcWrite(u,ch + '0') END;
IF carry NEQ 0 THEN rcWrite(u,'1'); RETURN(u);
END;
```

```
STRING PROCEDURE nonnegativeSub (STRING s,t);
# The strings s and t represent nonnegative integers,
# s > t.  Return the string that represents their
# difference.
BEGIN
INTEGER borrow,ch;
STRING u;
u := ""; borrow := 0;
WHILE s OR t DOB
    borrow := digitRead(s) - digitRead(t) - borrow;
    IF borrow < 0 THENB ch := borrow + 10; borrow := 1 END
    EB ch := borrow; borrow := 0 END;
    rcWrite(u,ch + '0') END;
WHILE first(u) = '0' DO cRead(u);
RETURN(u);
END;
```

```
INTEGER PROCEDURE nonnegativeCompare (STRING s,t);
# Compare two signless strings; return -1 if s < t,
# 0 if s = t, 1 if s > t
BEGIN
INTEGER chS,chT;
WHILE length(s) > length(t) DO
    IF cRead(s) NEQ '0' THEN RETURN(1);
WHILE length(s) < length(t) DO
    IF cRead(t) NEQ '0' THEN RETURN(-1);
# Now lengths are equal:
WHILE s DOB
    chS := cRead(s); chT := cRead(t);
    IF chS < chT THEN RETURN(-1)
    EF chS > chT THEN RETURN(1) END;
RETURN(0);
END;
```

```
BOOLEAN PROCEDURE wasNeg (MODIFIES STRING s);
BEGIN
IF first(s) = '-' THENB cRead(s); RETURN(TRUE) END;
RETURN(FALSE);
END;
```

```
STRING PROCEDURE stringAdd (STRING s,t);
BEGIN
BOOLEAN sNeg,tNeg;
INTEGER i;
STRING u;
sNeg := wasNeg(s); tNeg := wasNeg(t);
IF sNeg = tNeg THENB
    u := nonnegativeAdd(s,t); IF sNeg THEN rcWrite(u,'-');
    RETURN(u) END;
# Signs differ.  Make t the smaller one.
i := nonnegativeCompare(s,t); IF i = 0 THEN RETURN("0");
IF i = -1 THENB
    u := s; s := t; t := u; sNeg := NOT sNeg END;
u := nonnegativeSub(s,t); IF sNeg THEN rcWrite(u,'-');
RETURN(u);
END;
```

```
STRING PROCEDURE stringSub (STRING s,t);
BEGIN
# Change sign of t and add
IF NOT wasNeg(t) THEN rcWrite(t,'-');
RETURN(stringAdd(s,t));
END;
```

```
STRING PROCEDURE stringMul (STRING s,t);
BEGIN
BOOLEAN sNeg,tNeg;
INTEGER i,j,k;
STRING u,v,ss;
sNeg := wasNeg(s); tNeg := wasNeg(t); u := "0"; j := 0;
WHILE t DOB
    # Multiply s by each digit of t
    v := ""; FOR k := 1 UPTO j DO cWrite(v,'0');
    i := rcRead(t) - '0'; ss := s; k := 0;
    WHILE ss DOB
        k := k + i * (rcRead(ss) - '0');
        rcWrite(v,k - ((k DIV 10) * 10) + '0');
        k := k DIV 10 END;
    IF k NEQ 0 THEN rcWrite(v,k + '0');
    u := nonnegativeAdd(u,v); j := j + 1 END;
IF sNeg NEQ tNeg THEN rcWrite(u,'-');
RETURN(u);
END;
```

```
STRING PROCEDURE stringDiv (STRING s,t);
# s DIV t, except always truncates towards zero (direction
# of truncation of MAINSAIL DIV is unspecified for
# negative arguments).  Standard long division algorithm.
BEGIN
BOOLEAN sNeg,tNeg;
INTEGER shiftCount,i,numSubs;
STRING u;
sNeg := wasNeg(s); tNeg := wasNeg(t);
# Check for division by zero, and return zero if so:
WHILE t AND first(t) = '0' DO cRead(t);
IF t = "" THENB
    write(logFile,"Division by zero!" & eol);
    RETURN("0") END;
IF nonnegativeCompare(s,t) < 0 THEN RETURN("0");
    # Dividend less than divisor
# Shift divisor left until larger than dividend:
shiftCount := -1; # Account for later shift right
DOB shiftCount := shiftCount + 1;
    cWrite(t,'0') END UNTIL nonnegativeCompare(t,s) > 0;
# Since shift was too far, now shift back one:
rcRead(t); u := "";
# Now do repeated subtracts at each position:
FOR i := 0 UPTO shiftCount DOB
    numSubs := 0;
    WHILE nonnegativeCompare(t,s) LEQ 0 DOB
        numSubs := numSubs + 1;
        s := nonnegativeSub(s,t) END;
    cWrite(u,numSubs + '0'); # Accumulate a digit
    rcRead(t); # Shift divisor right one position
    END;
IF sNeg NEQ tNeg THEN rcWrite(u,'-');
RETURN(u);
END;
```

```
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 = "?" THEN write(logFile,
    "S                show stack" & eol &
    "C                clear stack" & eol &
    "<integer>        push integer" & eol &
    "+, -, *, /       add, subtract, multiply or" & eol &
    "                 divide top two stack elements" & eol)
EF s = "S" THEN write(logFile,"Stack:",stack,eol)
EF s = "C" THENB
    stack := ""; stackDepth := 0;
    write(logFile,"Stack cleared" & eol) END
EF s = "+" OR s = "-" OR s = "*" OR s = "/" THENB
    IF stackDepth < 2 THEN
        write(logFile,"Impossible; stack has only ",
            stackDepth," items." & eol)
    EB  u := stackTop; t := stackTop;
        IF s = "+" THEN t := stringAdd(t,u)
        EF s = "-" THEN t := stringSub(t,u)
        EF s = "*" THEN t := stringMul(t,u)
        EL              t := stringDiv(t,u);
        push(t) 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 "rpn2"
```

Less string manipulation would be done in the multiplication algorithm if arrays were used instead of strings, but arrays have not been introduced at the point of the exercise. Overall, efficiency has been sacrificed for simplicity of implementation in this program.

Nothing is done to prevent leading zeros or negative zeros; a real string calculator would probably do so. This would allow the test to see if s is a zero value to be:

```
first(s) = '0'
```

Exercise 7-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "pigLat"

BOOLEAN PROCEDURE isVowel (INTEGER ch);
# Disregard 'y'
BEGIN
RETURN(ch = 'a' OR ch = 'e' OR ch = 'i' OR ch = 'o' OR
       ch = 'u');
END;
```

```
STRING PROCEDURE pigify (STRING s);
# Return Pig Latin translation of s.
BEGIN
BOOLEAN wasCap;
INTEGER ch;
STRING ss,t,u;
ss := "";
WHILE s DOB
    # Remove non-letter characters and put into result:
    WHILE s AND NOT isAlpha(first(s)) DO
        cWrite(ss,cRead(s));
    IF NOT s THEN DONE;
    t := ""; # Gather the next word into t:
    WHILE isAlpha(first(s)) DO cWrite(t,cRead(s));
    # Preserve case of initial letter:
    wasCap := isUpperCase(first(t)); t := cvl(t);
    IF isVowel(first(t)) THEN u := t & "yay"
    EB  u := "";
        WHILE t AND NOT isVowel(first(t)) DO
            cWrite(u,cRead(t));
        u := t & u & "ay" END;
    IF wasCap THENB
        ch := cvu(cRead(u)); rcWrite(u,ch) END;
    ss := ss & u END;
RETURN(ss);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;
DOB write(logFile,"String to translate to Pig Latin " &
        "(<eol> to stop): "); read(cmdFile,s);
    IF NOT s THEN DONE;
    write(logFile,pigify(s),eol) END;
END;

END "pigLat"
```

Exercise 7-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "hexClc" # Hexadecimal calculator

STRING x,y,z;  # Values in the accumulators
```

```
BOOLEAN PROCEDURE isHexChar (INTEGER ch);
BEGIN
RETURN('0' LEQ ch LEQ '9' OR ch = 'A' OR ch = 'B' OR
    ch = 'C' OR ch = 'D' OR ch = 'E' OR ch = 'F');
END;



INTEGER PROCEDURE hexVal (INTEGER ch);
BEGIN
IF ch GEQ '0' AND ch LEQ '9' THEN RETURN(ch - '0')
EF ch = 'A' THEN RETURN(10)
EF ch = 'B' THEN RETURN(11)
EF ch = 'C' THEN RETURN(12)
EF ch = 'D' THEN RETURN(13)
EF ch = 'E' THEN RETURN(14)
EF ch = 'F' THEN RETURN(15)
EB  write(logFile,"Unexpected character '",cvcs(ch),"'" &
        eol);
    RETURN(0) END;
END;



INTEGER PROCEDURE hexChar (INTEGER val);
BEGIN
IF val GEQ 0 AND val LEQ 9 THEN RETURN(val + '0')
EF val = 10 THEN RETURN('A')
EF val = 11 THEN RETURN('B')
EF val = 12 THEN RETURN('C')
EF val = 13 THEN RETURN('D')
EF val = 14 THEN RETURN('E')
EF val = 15 THEN RETURN('F')
EB  write(logFile,"Unexpected value ",val,eol);
    RETURN('0') END;
END;



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; "-" may be
# an operator or the start of an integer:
IF (ch = '-' AND NOT isHexChar(first(s))) OR
    (ch NEQ '-' AND NOT isHexChar(ch)) THEN
    RETURN(cvcs(ch));
# Now it must be an integer:
t := cvcs(ch);
WHILE isHexChar(first(s)) 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 validHexInteger (STRING s);
# Returns true if s is a (possibly negative) hex integer.
BEGIN
IF NOT s THEN RETURN(FALSE);
IF first(s) = '-' THEN cRead(s);
WHILE s DO IF NOT isHexChar(cRead(s)) THEN RETURN(FALSE);
RETURN(TRUE); # If made it to here, must be OK
END;


INTEGER PROCEDURE digitRead (MODIFIES STRING s);
BEGIN
IF s THEN RETURN(hexVal(rcRead(s))) EL RETURN(0);
END;
```

```
STRING PROCEDURE nonnegativeAdd (STRING s,t);
# The strings s and t represent nonnegative hex integers.
# Return the string that represents their sum.
BEGIN
INTEGER carry,ch;
STRING u;
u := ""; carry := 0;
WHILE s OR t DOB
    carry := digitRead(s) + digitRead(t) + carry;
    IF carry GEQ 16 THENB ch := carry - 16; carry := 1 END
    EB ch := carry; carry := 0 END;
    rcWrite(u,hexChar(ch)) END;
IF carry THEN rcWrite(u,'1'); RETURN(u);
END;



STRING PROCEDURE nonnegativeSub (STRING s,t);
# The strings s and t represent nonnegative integers,
# s > t.  Return the string that represents their
# difference.
BEGIN
INTEGER borrow,ch;
STRING u;
u := ""; borrow := 0;
WHILE s OR t DOB
    borrow := digitRead(s) - digitRead(t) - borrow;
    IF borrow < 0 THENB ch := borrow + 16; borrow := 1 END
    EB ch := borrow; borrow := 0 END;
    rcWrite(u,hexChar(ch)) END;
WHILE first(u) = '0' DO cRead(u);
RETURN(u);
END;
```

```
INTEGER PROCEDURE nonnegativeCompare (STRING s,t);
# Compare two signless strings; return -1 if s < t,
# 0 if s = t, 1 if s > t
BEGIN
INTEGER chS,chT;
WHILE length(s) > length(t) DO
    IF cRead(s) NEQ '0' THEN RETURN(1);
WHILE length(s) < length(t) DO
    IF cRead(t) NEQ '0' THEN RETURN(-1);
# Now lengths are equal:
WHILE s DOB
    chS := cRead(s); chT := cRead(t);
    IF hexVal(chS) < hexVal(chT) THEN RETURN(-1)
    EF hexVal(chS) > hexVal(chT) THEN RETURN(1) END;
RETURN(0);
END;



BOOLEAN PROCEDURE wasNeg (MODIFIES STRING s);
BEGIN
IF first(s) = '-' THENB cRead(s); RETURN(TRUE) END;
RETURN(FALSE);
END;



STRING PROCEDURE stringAdd (STRING s,t);
BEGIN
BOOLEAN sNeg,tNeg;
INTEGER i;
STRING u;
sNeg := wasNeg(s); tNeg := wasNeg(t);
IF sNeg = tNeg THENB
    u := nonnegativeAdd(s,t); IF sNeg THEN rcWrite(u,'-');
    RETURN(u) END;
# Signs differ.  Make t the smaller one.
i := nonnegativeCompare(s,t); IF i = 0 THEN RETURN("0");
IF i = -1 THENB
    u := s; s := t; t := u; sNeg := NOT sNeg END;
u := nonnegativeSub(s,t); IF sNeg THEN rcWrite(u,'-');
RETURN(u);
END;
```

```
STRING PROCEDURE stringSub (STRING s,t);
BEGIN
# Change sign of t and add
IF NOT wasNeg(t) THEN rcWrite(t,'-');
RETURN(stringAdd(s,t));
END;
```

```
STRING PROCEDURE stringMul (STRING s,t);
BEGIN
BOOLEAN sNeg,tNeg;
INTEGER i,j,k;
STRING u,v,ss;
sNeg := wasNeg(s); tNeg := wasNeg(t); u := "0"; j := 0;
WHILE t DOB
    # Multiply s by each digit of t
    v := ""; FOR k := 1 UPTO j DO cWrite(v,'0');
    i := hexVal(rcRead(t)); ss := s; k := 0;
    WHILE ss DOB
        k := k + i * hexVal(rcRead(ss));
        rcWrite(v,hexChar(k - ((k DIV 16) * 16)));
        k := k DIV 16 END;
    IF k THEN rcWrite(v,hexChar(k));
    u := nonnegativeAdd(u,v); j := j + 1 END;
IF sNeg NEQ tNeg THEN rcWrite(u,'-');
RETURN(u);
END;
```

```
STRING PROCEDURE stringDiv (STRING s,t);
# s DIV t, except always truncates towards zero (direction
# of truncation of MAINSAIL DIV is unspecified for
# negative arguments).  Standard long division algorithm.
BEGIN
BOOLEAN sNeg,tNeg;
INTEGER shiftCount,i,numSubs;
STRING u;
sNeg := wasNeg(s); tNeg := wasNeg(t);
# Check for division by zero, and return zero if so:
WHILE t AND first(t) = '0' DO cRead(t);
IF NOT t THENB
    write(logFile,"Division by zero!" & eol);
    RETURN("0") END;
IF nonnegativeCompare(s,t) < 0 THEN RETURN("0");
    # Dividend less than divisor
# Shift divisor left until larger than dividend:
shiftCount := -1; # Account for later shift right
DOB shiftCount := shiftCount + 1;
    cWrite(t,'0') END UNTIL nonnegativeCompare(t,s) > 0;
# Since shift was too far, now shift back one:
rcRead(t); u := "";
# Now do repeated subtracts at each position:
FOR i := 0 UPTO shiftCount DOB
    numSubs := 0;
    WHILE nonnegativeCompare(t,s) LEQ 0 DOB
        numSubs := numSubs + 1;
        s := nonnegativeSub(s,t) END;
    cWrite(u,hexChar(numSubs)); # Accumulate a digit
    rcRead(t); # Shift divisor right one position
    END;
IF sNeg NEQ tNeg THEN rcWrite(u,'-');
RETURN(u);
END;



FORWARD STRING PROCEDURE value (MODIFIES STRING s);

STRING PROCEDURE factor (MODIFIES STRING s);
BEGIN
STRING t,i;
```

```
t := getToken(s);
IF t = "X" THEN RETURN(x)
EF t = "Y" THEN RETURN(y)
EF t = "Z" THEN RETURN(z)
EF t = "(" THENB
    i := value(s); t := getToken(s);
    IF t NEQ ")" THEN
        write(logFile,"Factor: missing ')' at ",t,eol);
    RETURN(i) END
EF validHexInteger(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;



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

product := factor(s);
DOB t := getToken(s);
    IF t = "*" THEN
        product := stringMul(product,factor(s))
    EF t = "/" THEN
        product := stringDiv(product,factor(s))
    EB  ungetToken(s,t); DONE END END;
RETURN(product);
END;



STRING PROCEDURE expression (MODIFIES STRING s);
BEGIN
STRING sum,t;

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

```
STRING PROCEDURE assignment (MODIFIES STRING s);
BEGIN
STRING val,regName;

regName := getToken(s); getToken(s); # Discard "="
val := value(s);
IF regName = "X" THEN x := val
EF regName = "Y" THEN y := val
EF regName = "Z" THEN z := val
EL write(logFile,"Assignment: illegal register name ",
         regName,eol);
RETURN(val);
END;


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

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 = "?" THEN write(logFile,
    "Q          quit" & eol &
    "S          show accumulators (named X, Y, Z)" & eol &
    "Else, type an infix arithmetic expression" & eol &
    "using the operators +, -, *, /, or" & eol &
    "= (assignment to register), and parentheses;" & eol &
    "constants are in hex and are negative if" & eol &
    "prefixed by ""-"".  Examples:" & eol &
    eol &
    "    3AB + E00" & eol &
    "    z = ((x = 100) - 1eab4004 * e31) / 2" & eol)
EF t = "S" THEN
    write(logFile,"X: ",x,eol & "Y: ",y,eol & "Z: ",z,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 leftover token ",t,eol);
RETURN(TRUE);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

# Set up the accumulators:
x := "0"; y := "0"; z := "0";

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

END "hexClc"
```

The names of the accumulators have been changed from A, B, and C in Example 7.2.2-2 of part I of the "MAINSAIL Tutorial" to X, Y, and Z, since A, B, and C are valid hexadecimal digits. The string math procedures from Exercise 6-2 of part I of the "MAINSAIL Tutorial" have been adapted to handle hexadecimal number strings. The syntax is a little more liberal than that of Example 7.2.2-2 of part I of the "MAINSAIL Tutorial" in that it allows an assignment embedded in an expression (e.g., "2 + (x = 100)", meaning assign 100 to X, then add 2); a call to the procedure "value" was substituted for the call to "expression" in the procedure "factor".

isHexChar,hexVal, and hexChar would be better implemented with arrays, but arrays have not been introduced at the point of the exercise.

Exercise 8-1 of part I of the "MAINSAIL Tutorial":

```
            (b1 SHR n TST '1) AND NOT (b2 SHR n TST '1)
```

Exercise 8-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "bitAdd"

BITS PROCEDURE bitByBitAdd (BITS b1,b2);
BEGIN
BOOLEAN carry,result,bo1,bo2;
INTEGER i;
BITS b;
b := '0; carry := FALSE;
FOR i := 0 UPTO 15 DOB
    bo1 := b1 TST ('1 SHL i); bo2 := b2 TST ('1 SHL i);
    IF result := ((bo1 NEQ bo2) NEQ carry) THEN
        b := b IOR ('1 SHL i);
    # If result NEQ carry, then carry is unchanged; if
    # result = carry, then bo1 = bo2 = new carry
    IF result = carry THEN carry := bo1 END;
IF carry THEN write(logFile,"Overflow!" & eol);
RETURN(b);
END;



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



INITIAL PROCEDURE;
BEGIN
BITS b1,b2;
STRING s;
write(logFile,"First bits value: "); read(cmdFile,s);
b1 := cvb(s); writeBits(b1);
write(logFile,"Second bits value: "); read(cmdFile,s);
b2 := cvb(s); writeBits(b2);
write(logFile,"Sum is "); writeBits(bitByBitAdd(b1,b2));
END;

END "bitAdd"
```

Exercise 9-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "calcFT" # CALC with F and T commands

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

POINTER(textFile) inFile,outFile;

STRING PROCEDURE getLine;
BEGIN
STRING s;
IF inFile THENB
    read(inFile,s);
    IF NOT $gotValue(inFile) THENB
        write(logFile,eol & "Reached end-of-file on ",
            inFile.name,", returning to cmdFile",eol);
        close(inFile); RETURN(getLine) END;
    write(logFile,s,eol) END
EL  read(cmdFile,s);
IF outFile THEN write(outFile,s,eol);
RETURN(s);
END;



PROCEDURE iPut (REPEATABLE INTEGER i);
BEGIN
IF outFile THEN write(outFile,i);
write(logFile,i);
END;



PROCEDURE sPut (REPEATABLE STRING s);
BEGIN
IF outFile THEN write(outFile,s);
write(logFile,s);
END;



GENERIC PROCEDURE put "iPut,sPut";
```

```
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;



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 put("Factor: missing ')'" & eol);
    RETURN(i) END
EF validInteger(t) THENB read(t,i); RETURN(i) END
EB  put("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  put("Term: division by zero" & eol);
            product := 0 END END # Give zero result
    EB ungetToken(s,t); DONE END END;
RETURN(product);
END;



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 put("Assignment: illegal register name ",regName,eol);
RETURN(val);
END;



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

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 := cvu(getToken(s));
IF t = "Q" THEN RETURN(FALSE);
IF t = "S" THEN put("A: ",a,"  B: ",b,"  C: ",c,eol)
EF t = "F" THENB
    IF inFile THEN
        put("""F"" command not allowed in command file" &
            eol)
    EB  IF inFile THEN close(inFile);
        WHILE first(s) = ' ' OR first(s) = first(tab) DO
            cRead(s);
        open(inFile,s,input) END;
    s := "" END
EF t = "T" THENB
    WHILE first(s) = ' ' OR first(s) = first(tab) DO
        cRead(s);
    IF outFile THEN close(outFile);
    IF s THENB open(outFile,s,create!output); s := "" END;
    END
EF t THENB
    ungetToken(s,t); s := cvu(s); put(value(s),eol) END;
IF t := getToken(s) THEN # Something was left over
    put("Illegal token ",t,eol);
RETURN(TRUE);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;

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

DOB put("CALC command ('Q' to quit): ");
    s := getLine END UNTIL NOT doCommand(s);

IF inFile THEN close(inFile);
IF outFile THEN close(outFile);
END;

END "calcFT"
```

The procedure doCommand is upgraded to handle the new "F" and "T" commands. Calls to "write(logFile,...)" and "read(cmdFile,...)" are now funneled through the new procedures put (a generic procedure, like write) and getLine, which take care of all the necessary echoing. The initial procedure closes its files at the end, as all good programs should.

Exercise 9-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "datClc" # Data file calculator

INITIAL PROCEDURE;
BEGIN
BITS b;
REAL acc,r;
POINTER(dataFile) d;
open(d,"Input data file: ",input!prompt);
acc := 0.0;
DOB read(d,r,b); IF b NTST '4 THEN DONE; b := b MSK '3;
    IF b = '0 THENB
        write(logFile,"Adding "); acc := acc + r END
    EF b = '1 THENB
        write(logFile,"Subtracting "); acc := acc - r END
    EF b = '2 THENB
        write(logFile,"Multiplying by ");
        acc := acc * r END
    EB  write(logFile,"Dividing by "); acc := acc / r END;
    write(logFile,r,"; new value = ",acc,eol);
    END;
write(logFile,"Done." & eol); close(d);
END;

END "datClc"
```

Exercise 10-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "symDel" # Symbol table with delete

# 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;

INTEGER PROCEDURE fSize (INTEGER typeCode);
RETURN($ioSize(f,typeCode));
```

```
LONG INTEGER PROCEDURE nullRecordPos;
RETURN(numBucketsPos + cvli(fSize(integerCode)));



LONG INTEGER PROCEDURE eofPosPos;
RETURN(nullRecordPos + cvli(fSize(longIntegerCode)));



LONG INTEGER PROCEDURE firstBucketPos;
RETURN(eofPosPos + cvli(fSize(longIntegerCode)));



# Now free list starts right after last bucket start pos
LONG INTEGER PROCEDURE freeListPos;
RETURN(firstBucketPos +
       cvli(numBuckets * fSize(longIntegerCode)));



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

```
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);
# + 1 to add free list:
FOR i := 1 UPTO numBuckets + 1 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;
```

```
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 * fSize(longIntegerCode)));



BOOLEAN PROCEDURE lookup
    (STRING recName;
     PRODUCES OPTIONAL STRING recVal;
     PRODUCES OPTIONAL LONG INTEGER atPos);
# Return true if record recName is found, or if recName
# is "" (illegal record name); atPos is record pos or -1L
# if recName ""
BEGIN
INTEGER nameLen,valLen;
LONG INTEGER nextPos,prevPos;

IF NOT recName THENB
    errMsg("Null record name"); recVal := "";
    atPos := -1L; RETURN(TRUE) END; # Act as if found it

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

```
LONG INTEGER PROCEDURE setUpRecordAtPos
    (LONG INTEGER pos,theBucketPos);
# Put in the record prefix (next and prev links) for a
# record at pos in the bucket starting at theBucketPos.
# Return the position just after the next and prev links.
BEGIN
LONG INTEGER listPos,endPos;
# Insert the record at the head of the hash list:
setPos(f,theBucketPos);
# Overwrite the head of the list position:
read(f,listPos); relPos(f,- fSize(longIntegerCode));
write(f,pos); setPos(f,pos);
# nullRecordPos means no back link
write(f,listPos,nullRecordPos);
endPos := getPos(f);
# Now update back link in former head record of bucket
IF listPos NEQ nullRecordPos THENB
    setPos(f,listPos + cvli(fSize(longIntegerCode)));
    write(f,pos) END;
RETURN(endPos);
END;



PROCEDURE writeRecord (STRING recName,recVal;
                       LONG INTEGER pos);
# Write the new record at pos.
BEGIN
LONG INTEGER endPos,eofPos;

setPos(f,setUpRecordAtPos(pos,bucketPos(hash(recName))));
write(f,length(recName));
WHILE recName DO write(f,cRead(recName));
write(f,length(recVal));
WHILE recVal DO write(f,cRead(recVal));
endPos := getPos(f);
# Now update eofPos if wrote record at end of file
setPos(f,eofPosPos); read(f,eofPos);
IF endPos > eofPos THENB
    setPos(f,eofPosPos); write(f,endPos) END;
END;
```

```
LONG INTEGER PROCEDURE findFreeSpaceFor
    (STRING recName,recVal);
# Currently looks for a record of exactly the same size on
# the free list.  Could be set up also to merge adjacent
# free records and to break up large free records when
# allocating a smaller record.  Return -1L if not found.
BEGIN
INTEGER len1,len2;
LONG INTEGER pos,nextPos,prevPos;
pos := freeListPos;
WHILE pos NEQ nullRecordPos DOB
    setPos(f,pos); read(f,nextPos,prevPos,len1);
    relPos(f,len1 * fSize(integerCode));
    read(f,len2);
    IF length(recName) + length(recVal) = len1 + len2 THEN
        RETURN(pos);
    pos := nextPos END;
RETURN(-1L);
END;



PROCEDURE removeFromList (LONG INTEGER atPos,listPos);
# Remove the record at atPos from the list at listPos.
BEGIN
LONG INTEGER nextRec,prevRec,ii;
setPos(f,atPos); read(f,nextRec,prevRec);
IF prevRec NEQ nullRecordPos THENB # link prev to next
    setPos(f,prevRec); write(f,nextRec) END;
IF nextRec NEQ nullRecordPos THENB # link next to prev
    setPos(f,nextRec + cvli(fSize(longIntegerCode)));
    write(f,prevRec) END;
setPos(f,listPos); read(f,ii);
IF ii = atPos THENB
    # removed head of list, make atPos's next new head
    setPos(f,listPos); write(f,nextRec) END;
END;



PROCEDURE createRecord (STRING s);
BEGIN
LONG INTEGER pos;
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;
# Now write it into the file
IF pos := findFreeSpaceFor(s,recVal) < 0L THENB
    setPos(f,eofPosPos); read(f,pos) END
EL  removeFromList(pos,freeListPos);
writeRecord(s,recVal,pos);
END;



PROCEDURE deleteRecord (STRING s);
BEGIN
LONG INTEGER atPos;
STRING recVal;

scan(s," " & tab,proceed!omit); # Remove leading blanks
IF NOT s OR NOT lookup(s,recVal,atPos) THENB
    errMsg("No such record:",s); RETURN END;
# Remove the record from its current bucket and add to
# the free list.
removeFromList(atPos,bucketPos(hash(s)));
setUpRecordAtPos(atPos,freeListPos);
END;



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,prevPos;

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,prevPos,nameLen);
        write(logFile,getString(nameLen),eol);
        setPos(f,nextPos) END END;
END;



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);
    ['D'] deleteRecord(s);
    ['L'] lookupRecord(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,"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

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

close(f);
END;

END "symDel"
```

The file format has been changed to have a free list after the last bucket position. The format of individual records has been changed to add a backwards link as well as a forwards link:

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

The backwards link facilitates the removing of a record from the list, at the cost of a little extra work when adding the record to the list.

$ioSize has been used instead of size to allow PDF I/O on the database file.

Exercise 10-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "symTxt"

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

```
DEFINE liSpace = 20L;     # Space allocated for long integer
                          # and eol (20 chars should be more
                          # than enough)

DEFINE numBucketsPos = 0L;
DEFINE nullRecordPos = numBucketsPos + liSpace;
DEFINE eofPosPos = nullRecordPos + liSpace;
DEFINE firstBucketPos = eofPosPos + liSpace;

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

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;
# Create null rec:
setPos(f,nullRecordPos); write(f,0L,eol);
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,eol);
# Now initialize all the buckets to be empty:
FOR i := 0 UPTO numBuckets - 1 DOB
    setPos(f,firstBucketPos + cvli(i) * liSpace);
    write(f,nullRecordPos,eol) END;
eofPos := firstBucketPos + cvli(numBuckets) * liSpace;
setPos(f,eofPosPos); write(f,eofPos,eol); 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
STRING s;

s := "";
WHILE (numChars .- 1) GEQ 0 DO cWrite(s,cRead(f));
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) * liSpace);



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;
STRING s;

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,s); nextPos := cvli(s);
    IF NOT nextPos THEN RETURN(FALSE); # End of this list
    read(f,s); nameLen := cvi(s);
    IF getString(nameLen) NEQ recName THENB
        setPos(f,nextPos); CONTINUE END;
    read(f,s); valLen := cvi(s);
    recVal := getString(valLen); RETURN(TRUE) END;
END;



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

setPos(f,eofPosPos); read(f,eofPos);
# Insert the record at the head of the hash list:
setPos(f,buckPos := bucketPos(hash(recName)));
# Overwrite the head of the list position:
read(f,listPos); setPos(f,buckPos);
write(f,eofPos,eol); setPos(f,eofPos);
write(f,listPos,eol,length(recName),eol,recName,
    length(recVal),eol,recVal);
eofPos := getPos(f); setPos(f,eofPosPos);
write(f,eofPos,eol);
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) END;
writeRecord(s,recVal); # Now write it into the file
END;



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;
STRING s;

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,s); nextPos := cvli(s);
        IF NOT nextPos THEN DONE;
        read(f,s); nameLen := cvi(s);
        write(logFile,getString(nameLen),eol);
        setPos(f,nextPos) END END;
END;
```

```
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

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

```
close(f);
END;

END "symTxt"
```

The format of the file is basically unchanged from Example 10.11-4 of part I of the "MAINSAIL Tutorial", except that integers and long integers take up an amount of space that varies with the value of the (long) integer. The long integer values that can be changed as new records are added to the database must therefore be padded in case a longer value overwrites a shorter one. Twenty characters' worth of space is alloted for the values and a terminating eol. Numbers in the records themselves are terminated with eol but not padded (note the use of the string form of read to remove the eol at the same time as the number). Since this version does not allow for changing or deleting a record, the numbers in a record are never changed, so padding is unnecessary.

The null character may appear in the file as part of the padding, but is never actually read by the program, so the keepNul but is not set in the call to open (the hint in the exercise suggested this might be necessary). Other solutions to the program might involve encoding (long) integers in a special way that sometimes writes the null character to represent a zero value.

Exercise 11-1 of part I of the "MAINSAIL Tutorial":

```
From p:
    REC's 1 (= p), 2 (= p.p1), 3 (= p.p1.p),
        4 (= p.p2), 5 (= p.p2.p2)
From q:
    REC's 4 (= q), 5 (= q.p2), 2 (= q.p2.p1),
        3 (= q.p2.p1.p)
```

Exercise 11-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "linLst"

CLASS lineLstCls (
    STRING line;
    POINTER(lineLstCls) prev,next;
);

POINTER(lineLstCls) lineLst;
```

```
PROCEDURE addToLineLst (STRING s);
BEGIN
INTEGER i;
POINTER(lineLstCls) p,pNext,q;
IF NOT p := lineLst OR
    compare(s,lineLst.line,upperCase) < 0 THENB
    lineLst := new(lineLstCls); lineLst.line := s;
    IF lineLst.next := p THEN p.prev := lineLst;
    RETURN END;
IF compare(s,lineLst.line,upperCase) = 0 THEN RETURN;
DOB pNext := p.next;
    IF NOT pNext OR
        i := compare(s,pNext.line,upperCase) LEQ 0 THEN
        DONE;
    p := pNext END;
IF NOT pNext OR i < 0 THENB
    # Otherwise i = 0 and line already added
    p.next := q := new(lineLstCls); q.line := s;
    q.prev := p;
    IF q.next := pNext THEN pNext.prev := q END;
END;



PROCEDURE printList;
BEGIN
POINTER(lineLstCls) p;
p := lineLst;
WHILE p DOB write(logFile,p.line,eol); p := p.next END;
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 addToLineLst(s) END;
close(f); printList;
END;

END "linLst"
```

The above linear list program produces the same output given the same input as the binary tree program in Example 11.5-2 of part I of the "MAINSAIL Tutorial". Run on unordered input files, the time required to add a new line to the binary tree data structure is proportional to the logarithm of the number of lines already seen; to the linear data structure, it is directly proportional to the number of lines seen. The binary tree is thus much faster for typical input. A quick benchmark, run on the same 2000-line file, shows the binary tree taking 4 seconds versus the linear list's 39 seconds. Procedure recursion, used in the binary tree program, is often thought of as an "expensive" technique compared to the Iterative Statement loop of the linear list program; however, the superior binary tree data structure more than makes up for the difference. Of course, it is also possible to construct a binary tree with an Iterative Statement (an exercise left to the reader).

Exercise 12-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "chars"

INITIAL PROCEDURE;
BEGIN
INTEGER ch;
LONG INTEGER ARRAY(0 TO $maxChar) ary;
POINTER(textFile) f;
open(f,"Input file: ",input!keepNul!prompt); new(ary);
WHILE ch := cRead(f) GEQ 0 DO ary[ch] .+ 1L; close(f);
FOR ch := 0 UPTO $maxChar DO IF ary[ch] THEN
        write(logFile,"Character code ",ch," ('",cvcs(ch),
            "'): ",ary[ch],eol);
END;

END "chars"
```

Exercise 12-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "maze"

DEFINE minX = 1;
DEFINE minY = 1;
DEFINE initialX = 10;
DEFINE initialY = 10;

INTEGER
    maxX,maxY, # Dimensions of maze within theMaze array
    sX,sY,     # Position of 'S'
    eX,eY;     # Position of 'E'

INTEGER ARRAY(minX TO *,minY TO *) theMaze;
```

```
PROCEDURE enlargeTheMazeIfNecessary;
# Make sure theMaze.ub1 GEQ maxX and theMaze.ub2 GEQ maxY
BEGIN
INTEGER i,j;
INTEGER ARRAY(minX TO *,minY TO *) ary;
IF theMaze.ub1 < maxX OR theMaze.ub2 < maxY THENB
    new(ary,minX,2 * (maxX MAX theMaze.ub1),
        minY,2 * (maxY MAX theMaze.ub2));
    FOR i := minX UPTO theMaze.ub1 DO
        FOR j := minY UPTO theMaze.ub2 DO
            ary[i,j] := theMaze[i,j];
    theMaze := ary END;
END;



BOOLEAN PROCEDURE checkForChar
    (INTEGER ch,targCh,x,y; MODIFIES INTEGER targX,targY);
# Return FALSE iff duplicate targCh seen.
BEGIN
IF ch NEQ targCh THEN RETURN(TRUE);
IF targX THENB
    write(logFile,"More than one ");
    cWrite(logFile,targCh); write(logFile,"!" & eol);
    RETURN(FALSE) END;
targX := x; targY := y; RETURN(TRUE);
END;
```

```
BOOLEAN PROCEDURE readFile;
BEGIN
INTEGER i,ch;
STRING s;
POINTER(textFile) f;
open(f,"Maze input file: ",input!prompt);
new(theMaze,minX,initialX,minY,initialY);
maxX := maxY := 0;
DOB read(f,s); IF NOT s := cvu(s) THEN DONE;
    maxY .MAX length(s); maxX .+ 1;
    enlargeTheMazeIfNecessary;
    FOR i := 1 UPTO length(s) DOB
        ch := theMaze[maxX,i] := cRead(s);
        IF NOT checkForChar(ch,'S',maxX,i,sX,sY) OR
           NOT checkForChar(ch,'E',maxX,i,eX,eY) THEN
           RETURN(FALSE) END END;
close(f);
IF NOT sX THEN write(logFile,"Didn't see an S!" & eol);
IF NOT eX THEN write(logFile,"Didn't see an E!" & eol);
RETURN(sX AND eX);
END;



PROCEDURE printMaze;
BEGIN
INTEGER i,j,ch;
FOR i := minX UPTO maxX DOB
    FOR j := minY UPTO maxY DO
        cWrite(logFile,
            IF ch := theMaze[i,j] THEN ch EL ' ');
        write(logFile,eol) END;
END;



FORWARD
BOOLEAN PROCEDURE solveMazeFrom (INTEGER x,y);
BOOLEAN PROCEDURE tryFrom (INTEGER x,y);
BEGIN
IF x < minX OR x > maxX OR y < minY OR y > maxY OR
    (theMaze[x,y] NEQ ' ' AND theMaze[x,y] NEQ 0) THEN
    RETURN(FALSE);
# Mark attempted trail so don't try this spot again while
# already trying a trail through it:
theMaze[x,y] := '.';
IF solveMazeFrom(x,y) THEN RETURN(TRUE);
theMaze[x,y] := ' '; # Didn't succeed, restore blank
RETURN(FALSE);
END;



BOOLEAN PROCEDURE solveMazeFrom (INTEGER x,y);
BEGIN
IF (abs(x - eX) = 1 AND abs(y - eY) = 0) OR
    (abs(x - eX) = 0 AND abs(y - eY) = 1) THEN
    RETURN(TRUE); # Right next to end point
IF tryFrom(x + 1,y) OR tryFrom(x,y + 1) OR
    tryFrom(x - 1,y) OR tryFrom(x,y - 1) THEN
    RETURN(TRUE);
RETURN(FALSE);
END;



BOOLEAN PROCEDURE solveMaze;
RETURN(solveMazeFrom(sX,sY));



INITIAL PROCEDURE;
BEGIN
IF NOT readFile THEN RETURN;
write(logFile,eol & "Unsolved maze:" & eol);
printMaze;
IF solveMaze THENB
    write(logFile,eol & "Solved maze:" & eol);
    printMaze END
EL  write(logFile,"Maze has no solution." & eol);
END;

END "maze"
```

The program shown relaxes the constraints of the exercise a little bit; it tolerates mazes in which not all the lines are the same length by treating 0 (an uninitialized array element) as a blank. The program still doesn't do anything to expand tabs into blanks, however.

Exercise 13-1 of part I of the "MAINSAIL Tutorial":

```
       DEFINE
           strDecls(n) =
               [REDEFINE strDeclStr = "";
                $FORC i = 1 UPTO n $DOC
                    REDEFINE strDeclStr = strDeclStr &
                        "STRING s" & cvs(i) & ";" & eol;
                    ENDC
                REDEFINE doStrDecls = [] & strDeclStr;
                doStrDecls];
```

Exercise 14-1 of part I of the "MAINSAIL Tutorial":

<u>File "arhdr.msl":</u>

```
   CLASS nodeCls (
       STRING op;
       POINTER(nodeCls) down,right,left;
   );

   MODULE arith (
       INTEGER ARRAY(0 TO $maxChar) ops;
       PROCEDURE evaluate (POINTER(nodeCls) p);
       STRING PROCEDURE getToken (MODIFIES STRING s);
       BOOLEAN PROCEDURE isNumber (STRING s);
   );

   # Nodes represent expressions and are represented by
   # records of nodeCls.  If op is an operator (e.g., "+",
   # "-"), then its list of operands is found on down; the
   # operands are connected through right.  If op is a
   # constant (e.g., "32") then down is nullPointer.

   # The left link may be useful to the input processor but
   # need not be set in a structure passed to ARITH.evaluate.

   # All operators are single characters.  The number of
   # operands of each operator is given by ARITH in the
   # array ops.  Invalid operators have ops[op] = -1.
```

```
# ARITH.evaluate takes a series of nodes connected through
# the right field and evaluates them, printing out the
# results.

# ARITH.getToken and ARITH.isNumber are utility procedures
# useful to all input processor modules.
```

File "arith.msl":

```
BEGIN "arith"

SOURCEFILE "arhdr.msl";

LONG INTEGER aReg,bReg,cReg;

STRING PROCEDURE getToken (MODIFIES STRING s);
# Return an integer or a (one-character) operator.
BEGIN
STRING ss;
$removeLeadingBlankSpace(s);
IF NOT ss := $removeInteger(s) THENB
    # If no valid integer, must be single character
    ss := s[1 FOR 1]; s := s[2 TO INF] END;
RETURN(ss);
END;



BOOLEAN PROCEDURE isNumber (STRING s);
RETURN('0' LEQ first(s) LEQ '9' OR length(s) > 1);
```

```
LONG INTEGER PROCEDURE eval (POINTER(nodeCls) p);
BEGIN
LONG INTEGER ii;
STRING op;
op := p.op;
IF isNumber(op) THEN RETURN(cvli(op));
CASE first(op) OFB
    ['A'] RETURN(aReg);
    ['B'] RETURN(bReg);
    ['C'] RETURN(cReg);
    ['M'] RETURN(- eval(p.down));
    ['+'] RETURN(eval(p.down) + eval(p.down.right));
    ['-'] RETURN(eval(p.down) - eval(p.down.right));
    ['*'] RETURN(eval(p.down) * eval(p.down.right));
    ['/'] BEGIN
        IF NOT ii := eval(p.down.right) THENB
            errMsg("Division by zero"); RETURN(0L) END;
        RETURN(eval(p.down) DIV ii) END;
    ['='] CASE first(p.down.op) OFB
            ['A'] RETURN(aReg := eval(p.down.right));
            ['B'] RETURN(bReg := eval(p.down.right));
            ['C'] RETURN(cReg := eval(p.down.right));
            [   ] BEGIN
                errMsg("Invalid left side of assignment:",
                    p.down.op);
                RETURN(0L) END;
            END;
    END;
END;


PROCEDURE evaluate (POINTER(nodeCls) p);
# Expressions are assumed to contain only legal
# operators/operands and operators to have the right
# number of operands.
BEGIN
WHILE p DOB write(logFile,eval(p)," "); p := p.right END;
write(logFile,eol);
END;
```

```
INITIAL PROCEDURE;
BEGIN
INTEGER i;
new(ops); FOR i := 0 UPTO $maxChar DO ops[i] := -1;
# Now fill in the implemented ones:
ops['A'] := ops['B'] := ops['C'] := 0; # Registers (0 arg)
ops['M'] := 1; # Unary minus
ops['+'] := ops['-'] := ops['*'] := ops['/'] :=
    ops['='] := 2; # Binary ops; = is assignment
END;

END "arith"
```

<u>File "rpn.msl":</u>

```
BEGIN "rpn"

SOURCEFILE "arhdr.msl";

POINTER(nodeCls) expr,exprEnd;

PROCEDURE addNode (STRING op; OPTIONAL POINTER down);
# Add to end of expr
BEGIN
POINTER(nodeCls) p;
p := new(nodeCls); p.op := op; p.down := down;
IF exprEnd THENB
    (exprEnd.right := p).left := exprEnd;
    exprEnd := p END
EL  expr := exprEnd := p;
END;
```

```
BOOLEAN PROCEDURE addToExpr (STRING s);
# Return false iff error
BEGIN
INTEGER i,j;
POINTER(nodeCls) p;
IF NOT s THEN RETURN(TRUE);
IF isNumber(s) THEN addNode(s)
EB  IF j := ops[first(s)] < 0 THENB
        errMsg("Not a valid operator/operand:",s);
        RETURN(FALSE) END;
    # Remove this many operands
    IF NOT j THEN p := NULLPOINTER
    EB  p := exprEnd;
        FOR i := 2 UPTO j WHILE p DO p := p.left;
        IF NOT p THENB
            errMsg("Insufficient operands for operator:",
                s);
            RETURN(FALSE) END;
        # Unlink operands from expr list
        IF NOT exprEnd := p.left THEN expr := NULLPOINTER
        EL p.left := exprEnd.right := NULLPOINTER END;
    # Add operator, with operands, if any, (back) to list
    addNode(s,p) END;
RETURN(TRUE);
END;


INITIAL PROCEDURE;
BEGIN
BOOLEAN bo;
STRING s;
bind(arith);
DOB write(logFile,"RPN expression(s) (<eol> to quit): ");
    read(cmdFile,s); IF NOT s := cvu(s) THEN DONE;
    expr := exprEnd := NULLPOINTER;
    WHILE bo := addToExpr(getToken(s)) DO UNTIL NOT s;
    IF bo THEN evaluate(expr) END;
END;

END "rpn"
```

File "std.msl":

```
BEGIN "std"

SOURCEFILE "arhdr.msl";
```

```
POINTER(nodeCls) PROCEDURE newNode
    (STRING op; OPTIONAL POINTER(nodeCls) down);
BEGIN
POINTER(nodeCls) p;
p := new(nodeCls); p.op := op; p.down := down; RETURN(p);
END;



FORWARD
POINTER(nodeCls) PROCEDURE expr (MODIFIES STRING s);

POINTER(nodeCls) PROCEDURE maybeBinaryOp
    (POINTER(nodeCls) p; MODIFIES STRING s);
# Just saw p; must be about to see end of string,
# binary op, or ")".  Return appropriate tree if binary
# op, else p.
BEGIN
STRING ss,sss;
sss := s;
IF NOT ss := getToken(s) THEN RETURN(p);
# must be binary op or ")"
IF ss = ")" THENB s := sss; RETURN(p) END;
IF isNumber(ss) OR ops[first(ss)] NEQ 2 THENB
    errMsg("Expected binary op or ) or end at:",sss);
    RETURN(NULLPOINTER) END;
IF NOT p.right := expr(s) THEN RETURN(NULLPOINTER);
RETURN(newNode(ss,p));
END;
```

```
POINTER(nodeCls) PROCEDURE expr (MODIFIES STRING s);
# Precedence information isn't given by ARITH, so
# some operator precedences here are non-standard;
# use parentheses when in doubt.
BEGIN
STRING ss;
POINTER(nodeCls) p;
IF NOT ss := getToken(s) THENB
    errMsg("Unexpected end of string");
    RETURN(NULLPOINTER) END;
IF ss = "(" THENB
    IF NOT p := expr(s) THEN RETURN(NULLPOINTER);
    ss := s;
    IF getToken(s) NEQ ")" THENB
        errMsg("Expected ) at:",ss);
        RETURN(NULLPOINTER) END;
    RETURN(maybeBinaryOp(p,s)) END;
# Handle 0-ary, unary, and binary operators only
IF isNumber(ss) OR ops[first(ss)] = 0 THENB
    IF NOT p := newNode(ss) THEN RETURN(NULLPOINTER);
    RETURN(maybeBinaryOp(p,s)) END;
IF ops[first(ss)] = 1 THENB
    IF NOT p := expr(s) THEN RETURN(NULLPOINTER);
    RETURN(newNode(ss,p)) END;
errMsg("Unexpected operator:",ss);
RETURN(NULLPOINTER);
END;



INITIAL PROCEDURE;
BEGIN
STRING s;
POINTER(nodeCls) p;
bind(arith);
DOB write(logFile,
        "Arithmetic expression (<eol> to quit): ");
    read(cmdFile,s); IF NOT s := cvu(s) THEN DONE;
    IF p := expr(s) AND s THEN errMsg("Leftover text:",s);
    IF p THEN evaluate(p) END;
END;

END "std"
```

Exercise 15-1 of part I of the "MAINSAIL Tutorial":

This exercise is too open-ended and would require too large a solution to be presented here. If you want additional ideas, examine also the ecological simulation of Section 19.4 of part I of the "MAINSAIL Tutorial", which uses coroutines to represent different creatures in a rectangular grid world. Coroutine-based creatures would probably be appropriate to a dungeon game as well.

Exercise 19-1 of part I of the "MAINSAIL Tutorial":

```
BEGIN "sched"

REDEFINE $scanName = "dpyHdr"; SOURCEFILE "msl:syslib";

MODULE(dpyCls) dpy; # required by display module

MODULE sched (
    STRING PROCEDURE readInput;
    PROCEDURE writeOutput (REPEATABLE STRING s);
    PROCEDURE poll;
);

CLASS coroutLstCls (
    POINTER($coroutine) co;
    STRING applicationName,inBuffer,inLineBuffer,
        outBuffer;
    POINTER dataSec;
    BOOLEAN wantsInput,hasInput;
    POINTER(coroutLstCls) next,prev; # circular list
    INTEGER firstScreenRow,lastScreenRow,
        outputEndRow,outputEndCol;
);

POINTER(coroutLstCls) coroutLst,curCorout,cursorCorout;
POINTER($coroutine) parent;



PROCEDURE screenMsg (STRING s);
# Show a message and wait for a keystroke
BEGIN
setCursorOnScreen(lastRowOfScreen,0);
clearToEndOfLine; overStrikeChars(s); dpycRead;
END;
```

```
STRING PROCEDURE readInput;
# This is called from the application.
BEGIN
STRING s;
curCorout.wantsInput := TRUE;
curCorout.co := $thisCoroutine; # In case switched
$resumeCoroutine(parent);
s := curCorout.inLineBuffer; curCorout.inLineBuffer := "";
curCorout.wantsInput := curCorout.hasInput := FALSE;
RETURN(s);
END;



PROCEDURE writeOutput (REPEATABLE STRING s);
# This is called from the application.
BEGIN
curCorout.outBuffer .& s;
curCorout.co := $thisCoroutine; # In case switched
$resumeCoroutine(parent);
END;



PROCEDURE poll;
# This is called from the application.
$resumeCoroutine(parent);



PROCEDURE runApplication;
# This is each application coroutine's initializing
# procedure.
BEGIN
IF NOT curCorout.dataSec :=
    new(curCorout.applicationName,errorOK) THEN
    screenMsg("Couldn't find module " &
        curCorout.applicationName &
        "; any char continues");
IF curCorout.dataSec THEN dispose(curCorout.dataSec);
$resumeCoroutine(parent,delete);
    # when done, coroutine dies
END;
```

```
PROCEDURE scroll (POINTER(coroutLstCls) c);
# scroll by one line
BEGIN
setCursorOnScreen(c.firstScreenRow,0);
deleteLines(1,c.lastScreenRow - c.firstScreenRow + 1);
c.outputEndRow .- 1;
END;



PROCEDURE doWriteOutput;
# If there is input waiting, move it over
BEGIN
INTEGER ch;
STRING s;
setCursorOnScreen(curCorout.outputEndRow,
                  curCorout.outputEndCol);
IF curCorout.inBuffer THEN clearToEndOfLine;
WHILE curCorout.outBuffer DOB
    s := scan(curCorout.outBuffer,eol,break!discard,ch);
    overstrikeChars(s);
    IF ch GEQ 0 THENB
        IF curCorout.outputEndRow .+ 1 >
            curCorout.lastScreenRow THEN
            scroll(curCorout);
        curCorout.outputEndCol := 0 END;
    curCorout.outputEndCol .+ length(s);
    IF ch < 0 THENB
        setCursorOnScreen(curCorout.outputEndRow,
                          curCorout.outputEndCol);
        overstrikeChars(curCorout.inBuffer) END END;
END;



PROCEDURE updateCursor; # current "correct" cursor pos.
BEGIN
setCursorOnScreen(cursorCorout.outputEndRow,
                  cursorCorout.outputEndCol +
                      length(cursorCorout.inBuffer));
dpyInfo(dumpScreenBuffer);
END;
```

```
PROCEDURE processChar (INTEGER ch);
# process input char; if see eol, move inBuffer to
# inLineBuffer
BEGIN
updateCursor;
IF ch = dpyEol THENB
    IF cursorCorout.outputEndRow .+ 1 >
        cursorCorout.lastScreenRow THEN
        scroll(cursorCorout);
    cursorCorout.outputEndCol := 0;
    cursorCorout.inLineBuffer := cursorCorout.inBuffer;
    cursorCorout.hasInput := TRUE;
    cursorCorout.inBuffer := "" END
EL  CASE ch OFB
        [dpyUp] cursorCorout := cursorCorout.prev;
        [dpyDown] cursorCorout := cursorCorout.next;
IFC $charSet = $ascii THENC
        [8] [127] # Backspace, delete
$EFC $charSet = $ebcdic THENC
        [22] [4] # Backspace, delete
ELSEC MESSAGE "Unknown character set","error";
ENDC
        IF cursorCorout.inBuffer THENB
            rcRead(cursorCorout.inBuffer);
            updateCursor; deleteChars(1) END
        EL  ringBell;
        [ ] BEGIN # Normal printing char
            overstrikeChar(ch);
            cWrite(cursorCorout.inBuffer,ch) END;
        END;
updateCursor;
END;



PROCEDURE findRunnableCoroutine;
# set curCorout to one that doesn't want input (or that
# wants it and already has it), if any; otherwise, don't
# change curCorout.
BEGIN
POINTER(coroutLstCls) p,q;
p := q := curCorout;
WHILE q.wantsInput AND NOT q.inLineBuffer DO q := q.next
    UNTIL q = p;
curCorout := q;
END;
```

```
INITIAL PROCEDURE;
BEGIN
INTEGER i,j,k;
STRING s;
POINTER(coroutLstCls) p,endP;
parent := $thisCoroutine; i := 0;
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;
    i .+ 1 END;

write(logFile,"Display module: "); read(cmdFile,s);
setModName("dpy",s); initializeTerminal; clearScreen;

# Now partition the screen among the applications
j := lastRowOfScreen DIV i; # num lines per appl.
p := coroutLst;
FOR k := 0 UPTO i - 1 DOB
    p.lastScreenRow :=
        (p.outputEndRow := p.firstScreenRow := k * j) +
        j - 1;
    p := p.next END;
```

```
cursorCorout := curCorout := coroutLst;
# Round-robin scheduling, except that gives preference to
# applications that don't want input.  Input is put into
# the window that has the cursor in it; arrow keys change
# windows.
# Ideally would get input "simultaneously" with output,
# but can't check for typeahead, so input always blocks.
updateCursor;
DOB findRunnableCoroutine;
    IF curCorout.wantsInput THENB
        # All living coroutines must want input, or would
        # not have run one that wants input
        updateCursor;
        DO  processChar(dpycRead)
            UNTIL cursorCorout.hasInput;
        curCorout := cursorCorout END;
    $HANDLE $resumeCoroutine(curCorout.co)
    $WITH # Kill only if fatal error
        IF $exceptionBits TST $cannotReturn AND
            $exceptionBits NTST $cannotFallOut THENB
            $killCoroutine(curCorout.co);
            screenMsg("Killed " &
                curCorout.applicationName &
                " because of " & $exceptionName &
                "; any key continues");
            curCorout.wantsInput := FALSE END
        EL  $raise;
    IF curCorout.outBuffer THEN doWriteOutput;
    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;
        IF curCorout = cursorCorout THEN
            cursorCorout := curCorout.next END;
    curCorout := curCorout.next END;
screenMsg("All applications done; hit any key to end");
clearScreen; deInitializeTerminal;
unbind(dpy); relModName("dpy");
END;

END "sched"
```

The STREAMS package provides a pre-built mechanism for automatically scheduling different applications on separate I/O channels (streams).

Exercise 19-2 of part I of the "MAINSAIL Tutorial":

```
BEGIN "nCrt"

RESTOREFROM "crtHdr";
RESTOREFROM "srtMod";

BOOLEAN underAttack;

INTEGER
    attackerX,attackerY,
    childsBirthEnergy,reserve;

STRING origInheritance;

POINTER(what) ARRAY (-2 TO 2,-2 TO 2) w;

# Format of inheritance string is:
#
#       n mXXX
#
# n is the birth energy for children; m is the reserve;
# XXX is some sequence of L's, S's, and T's.  Before the
# creature does anything else, it tries to produce a leaf
# for each L, a shell unit for each S, and a tooth for
# each T, in the order given.

STRING PROCEDURE getInheritance;
BEGIN
POINTER(textFile) f;
IF NOT origInheritance := myGeneticString THENB
    origInheritance := "";
    # Arbitrary but reasonable initial default:
    write(origInheritance,halfBirthEnergy * 2," ",
        energyPerShell,"LST") END;
IF NOT open(f,"ncrt.dat",input!output!random!errorOK)
   THEN open(f,"ncrt.dat",create!input!output!random);
setPos(f,$getEofPos(f));
write(f,$thisCoroutine.$name,": ",origInheritance,eol);
close(f);
read(origInheritance,childsBirthEnergy,reserve);
RETURN(origInheritance);
END;
```

```
STRING PROCEDURE childsInheritance;
BEGIN
INTEGER i;
STRING s,ss;
i := childsBirthEnergy;
IF NOT ran(0,2) THEN i .+ ran(-2,2); # Perturb?
i .MAX energyForLeaves; # Reasonable minimum
s := cvs(i);
i := reserve;
IF NOT ran(0,2) THEN i .+ ran(-2,2); # Perturb?
i .MAX 0;
write(s," ",i);
# Now, should we make the LST list the same or different?
IF ran(0,5) THENB # remove old energy, ss is old LST list
    ss := origInheritance; read(ss,i) END
EB  ss := ""; # ss is new LST list, random length/content
    FOR i := 1 UPTO ran(0,10) DO
        CASE ran(1,3) OFB
            [1] cWrite(ss,'L');
            [2] cWrite(ss,'S');
            [3] cWrite(ss,'T');
            END END;
RETURN(s & ss);
END;
```

```
PROCEDURE checkCell
    (INTEGER x,y; MODIFIES INTEGER
        tShell,tTeethLo,tTeethHi,tTotEn,enemyX,enemyY,
        foodX,foodY;
     MODIFIES OPTIONAL INTEGER emptyX,emptyY);
# Find best enemy and food, based on shell, teeth, and
# energy.  Creature of same name is not candidate for
# either food or enemy.  Any empty cell will do for
# (emptyX,emptyY).
BEGIN
INTEGER pShell,pTeeth,pTotEn,meTeeth,meShell;
POINTER(what) p;
p := w[x,y];
CASE p.status OFB
    [emptyCell] BEGIN emptyX := x; emptyY := y END;
    [cellOutsideWorld] ;
    [critterCell]
        IF p.critterName NEQ w[0,0].critterName THENB
            pShell := p.shellAmount;
            pTeeth := p.teethAmount;
            pTotEn := p.totalEnergy;
            meTeeth := w[0,0].teethAmount;
            meShell := w[0,0].shellAmount;
            IF pShell < meTeeth AND pTeeth < meTeeth AND
                pTeeth < meShell AND pShell < tShell AND
                pTeeth < tTeethLo OR
                pShell = tShell AND pTeeth = tTeethLo AND
                pTotEn > tTotEn THENB # best food so far
                foodX := x; foodY := y;
                tShell := pShell; tTeethLo := pTeeth;
                tTotEn := pTotEn END;
            IF (pTeeth > meTeeth OR pTeeth > meShell) AND
                pTeeth > tTeethHi THENB # worst enemy
                enemyX := x; enemyY := y;
                tTeethHi := pTeeth END END;
    END;
END;
```

```
BOOLEAN PROCEDURE emptyCellAdjacentTo
    (INTEGER tX,tY; PRODUCES INTEGER x,y);
# If there is an empty cell adjacent to both (0,0) and
# (tX,tY), return it as (x,y) and return true.  (tX,tY)
# is on outer perimeter of square, so
# abs(x) MAX abs(y) = 2.
BEGIN
FOR x := -1 UPTO 1 DO FOR y := -1 UPTO 1 DO
        IF abs(tX - x) LEQ 1 AND abs(tY - y) LEQ 1 AND
            w[x,y].status = emptyCell THEN RETURN(TRUE);
RETURN(FALSE);
END;



PROCEDURE mainLoop;
BEGIN
INTEGER en,x,y,emptyX1,emptyY1,enemyX1,enemyY1,
    edibleX1,edibleY1,enemyX2,enemyY2,edibleX2,edibleY2,i,
    tShell1,tTeethLo1,tTeethHi1,tTotEn1,
    tShell2,tTeethLo2,tTeethHi2,tTotEn2,moveCost;
POINTER(what) me;
INTEGER ARRAY(0 TO 24) ary,ind;

new(ary); new(ind);

DOB lookAround(w); me := w[0,0];
    en := me.spareEnergy - reserve;
    moveCost :=
        (me.totalEnergy + me.teethAmount + me.shellAmount)
            DIV costs1ToMove + 1;

    # Always make sure has leaf if possible
    IF NOT me.hasLeaves AND en > energyForLeaves THEN
        addLeaves;

    # Use the SRTMOD indexed sort to search the cells in
    # visible part of the world in a random order.  A
    # deterministic order imparts a bias to the creature's
    # actions.
    FOR i := 0 UPTO 24 DO ary[i] := ran(1,10000);
    sort(ary,0,24,ind);
    # This leaves ind as a randomly ordered array of
    # numbers 0 to 24, representing the positions in the
    # visible square
# Now check for empty space, enemies, and edibles
emptyX1 := emptyY1 := enemyX1 := enemyY1 :=
    edibleX1 := edibleY1 := enemyX2 := enemyY2 :=
    edibleX2 := edibleY2 := 0;
tShell1 := tTeethLo1 := tShell2 := tTeethLo2 :=
    $maxInteger;
tTotEn1 := tTeethHi1 := tTotEn2 := tTeethHi2 :=
    $minInteger;
FOR i := 0 UPTO 24 DOB
    x := ind[i] MOD 5 - 2; y := ind[i] DIV 5 - 2;
    IF x = y = 0 THEN CONTINUE;
    IF abs(x) MAX abs(y) = 1 THEN
        checkCell(x,y,tShell1,tTeethLo1,tTeethHi1,
            tTotEn1,enemyX1,enemyY1,edibleX1,edibleY1,
            emptyX1,emptyY1)
    EL  checkCell(x,y,tShell2,tTeethLo2,tTeethHi2,
            tTotEn2,enemyX2,enemyY2,
            edibleX2,edibleY2) END;

IF underAttack THENB # Run away
    IF w[x := - attackerX,y := - attackerY].status =
        emptyCell THEN underAttack := FALSE
    EF emptyX1 OR emptyY1 THENB
        x := emptyX1; y := emptyY1;
        underAttack := FALSE END;
    IF NOT underAttack THENB
        move(x,y); underAttack := FALSE; CONTINUE END;
    underAttack := FALSE END;

# Reproduce if possible
IF (emptyX1 OR emptyY1) AND
    en > birthCost + childsBirthEnergy THENB
    reproduce(emptyX1,emptyY1,childsBirthEnergy,
        childsInheritance);
    CONTINUE END;

# Eat if possible
IF (edibleX1 OR edibleY1) AND en > attackCost THENB
    attack(edibleX1,edibleY1); CONTINUE END;

# Run away if possible
IF (enemyX1 OR enemyY1) AND (emptyX1 OR emptyY1) AND
    en > moveCost THENB
    move(emptyX1,emptyY1); CONTINUE END;
    # Move towards far food if possible
    IF (edibleX2 OR edibleY2) AND en > moveCost AND
        tTotEn2 > moveCost AND
        emptyCellAdjacentTo(edibleX2,edibleY2,x,y) THENB
        move(x,y); CONTINUE END;

    # Now move if there is far enemy
    IF (enemyX2 OR enemyY2) AND (emptyX1 OR emptyY1) AND
        en > moveCost THENB
        move(emptyX1,emptyY1); CONTINUE END;

    # Now add shell if need shell
    IF me.shellAmount < maxShell AND en > energyPerShell
        THENB addShell; CONTINUE END;

    # Now add teeth if need teeth
    IF me.teethAmount < maxTeeth AND en > energyPerTooth
        THENB addTeeth; CONTINUE END;

    # Now just move, but only if fully armed sometimes
    # at random
    IF (emptyX1 OR emptyY1) AND
        me.shellAmount GEQ maxShell AND
        me.teethAmount GEQ maxTeeth AND NOT ran(0,100) AND
        en > moveCost THENB
        move(emptyX1,emptyY1); CONTINUE END;

    END;
END;


INITIAL PROCEDURE;
BEGIN
STRING inheritance;

inheritance := getInheritance;
# First add the mandatory characteristics required by
# inheritance:
DO  CASE cRead(inheritance) OFB
        [-1] DONE;
        ['L'] addLeaves;
        ['S'] addShell;
        ['T'] addTeeth;
        END;
```

```
$HANDLE mainLoop
$WITHB
    IF $exceptionName = attackExcpt THENB
        underAttack := TRUE;
        attackerX := cvi($exceptionStringArg1);
        attackerY := cvi($exceptionStringArg2) END;
    $raise END;
END;

END "nCrt"
```

This creature makes use of the genetic string facility. Note also the use of the SRTMOD procedure "sort" to shuffle a sequence of integers 0 through 24.

Experimentation shows that this module usually performs pretty well against the creatures of Examples 19.4-3 and 19.4-4 of part I of the "MAINSAIL Tutorial", polishing them off in a few juicy bites.

Appendix B. Comparison of MAINSAIL and C

The most popular programming language today is C (we wish we could say it were MAINSAIL, but alas, that is not so). MAINSAIL and C are both general-purpose programming languages, suitable to a great variety of tasks.

MAINSAIL provides many facilities that C does not, including garbage collection, dynamic linkage, coroutines, and a well-developed exception mechanism. MAINSAIL's approach to programming tasks is, generally speaking, more high-level than C's. MAINSAIL is particularly well suited to writing large, complicated systems; on the other hand, its large size can be inconvenient in programs that must fit into a small address space or that cannot afford to spend any time on program initialization. Because of garbage collection considerations, MAINSAIL is unsuited to real-time applications unless great care is taken to avoid generating garbage data.

C is more suitable for small tasks than MAINSAIL. It is also closer to the UNIX operating system, so that it is less awkward to write UNIX-specific code. C's lack of garbage collection can be annoying when designing large programs that allocate and deallocate data freely, since much of the program design is concerned with memory management. On the other hand, a lack of garbage collections means that time-critical tasks are not interrupted, so C is more suitable for real-time applications. C has relatively powerful compiletime facilities, such as macros and the ability to include one source file in another; however, MAINSAIL's compiletime facilities are more extensive. C programs are linked with the operating system linker, which may impose non-portable restrictions (such as the maximum length of an external symbol) and also usually fails to provide interface consistency checking and information hiding among C modules (both are available in MAINSAIL).

B.1. Portability

MAINSAIL and C are both highly portable programming languages. In the case of C, portability is a consequence of the widespread popularity of the UNIX operating system, which is implemented in C. C's portability depends on the presence of a library that provides calls that emulate UNIX system calls (sometimes imperfectly, if the underlying operating system is not UNIX). In MAINSAIL's case, the language was designed from the beginning to be ported to a variety of operating systems. MAINSAIL therefore adapts somewhat better to C to non-UNIX operating systems, although it runs quite satisfactorily on UNIX as well.

B.2. Modules and Dynamic Linking

One of MAINSAIL's great strengths is its concept of a module. In C, all (non-static) procedures and variables from all files that are linked together to form a program are on an equal footing. All non-static identifiers are visible in every module in the program. In MAINSAIL, each module has a well-defined interface. Only those modules that declare external modules can see the external procedures and variables provided by those modules.

MAINSAIL modules are linked dynamically, i.e., at runtime. A MAINSAIL module is not brought into memory until it is needed (e.g., when one of its interface procedures is called). In many cases, the MAINSAIL module is brought into memory automatically, without the calling module's having to do any setup. When the code for a module is no longer needed, it may also be removed from memory automatically. MAINSAIL programs can be much larger than C programs, since they do not have to be linked together all at once. A MAINSAIL system can consist of hundreds or thousands of modules, which might be too large to fit in memory if all loaded at once; however, in a typical execution, only some of the modules may ever be actually loaded, and those that are loaded but then no longer needed may be swapped out.

MAINSAIL module interfaces are checked for consistency when the module is loaded. This allows the runtime system to catch interface inconsistency errors that may be difficult to detect in a C program.

A MAINSAIL module may have more than one instance; all the interface variables and module-specific ("outer") variables are maintained in a record-like structure called a "data section". Many data sections can share the same code. The replication of data sections with identical interfaces encourages an object-oriented programming style difficult to achieve in standard C.

MAINSAIL does pay a penalty for its dynamic linking; the file I/O entailed by module loading does require some time, and intermodule calls are slightly (although usually not noticeably) less efficient than local (within one module) calls. Furthermore, some MAINSAIL functions must be a part of the MAINSAIL kernel, a memory-resident module loaded when MAINSAIL execution starts. Because it is not known a priori which kernel features will be used during an execution, the entire (large) kernel must be loaded each time. A C program can be selectively linked to include only those procedures that are necessary to support a given program, which can result in a small object file if the program does not use many features of the C runtime system. In practice, MAINSAIL's runtime overhead is usually negligible, but the large kernel requirement means that MAINSAIL is not suited for small computers. MAINSAIL just does not fit on most personal computers with typical amounts of memory; it usually needs an engineering workstation or a mainframe.

B.3. Compiletime Facilities

MAINSAIL and C each provide a more extensive set of compiletime preprocessing than do most programming languages. Facilities provided by the two languages are summarized in Table B.3-1.

Both languages provide macro facilities that are usually sufficient. MAINSAIL's are more general, since they allow multiline macro text, and macro text can be built up by concatenating arbitrary text and string constants. This also allows individual identifiers to be built up using macro parameters. MAINSAIL does not provide the ability to forget a macro identifier analogous to C's "#undef".

MAINSAIL provides "inline procedures", which are procedures compiled inline in the calling procedure like a macro definition. Inline procedures look syntactically like regular procedures. They provide type checking as well as avoiding the common macro problems with multiple evaluations of macro arguments and unintended interactions between macro text and argument text or surrounding text.

MAINSAIL's conditionals are more flexible than C's. In addition to "IFC", a compiletime analogue of the If Statement and similar to C's "#if", MAINSAIL provides compiletime analogues of Iterative and Case (switch) Statements, "$DOC" and "$CASEC".

The MAINSAIL compiletime system procedure $compileTimeValue provides information on the current compiler subcommands in effect, the name of the file being compiled, the name of the current procedure (if any), and the date and time of the current compilation, among other information. Since this information is available at compiletime, it can be used to govern conditional compilation. It can also be used to automate the generation of runtime debugging or error messages.

"SKIPSCAN" and "BEGINSCAN" in MAINSAIL provide a way to cause compilation to skip forward long distances in the current sourcefile, and are more convenient than "IFC" for this purpose.

MAINSAIL intmods are repositories of precompiled symbols and procedure bodies. If many different modules share common information, it is better to create an intmod than to use the "SOURCEFILE" directive to recompile the same text into every module, as C must do. Intmods also provide information hiding; i.e., symbols that support an intmod's public interface may be hidden so that they do not conflict with identifiers in the programs that use the intmod. This is more reliable than just choosing a long name for such supporting symbols, as must be done if common code is picked up as source text.

MAINSAIL's flexible compiletime facilities simplify the writing of very large programs, which often contain numerous repetitions of nearly similar text. For such programs, the more powerful a programming language's macro facilities, the better.

Facility	MAINSAIL	C
macro definition	DEFINE, REDEFINE	#define
forgetting a macro definition	no	#undef
arbitrary macro text	yes, delimited by [and]	no, must fit on one line
repeatable macro arguments	REPEATABLE	no
argument substitution in string constants and identifiers	yes, identifiers can be constructed	no
inline procedures	INLINE	no
conditional	IFC	#if
conditional based on whether identifier defined	IFC DCL	#ifdef
compiletime iteration, case (switch)	$DOC, $CASEC	no
setting line and file	no	#line
including another source file	SOURCEFILE	#include
information about current compilation	$compileTimeValue	no
skipping large amounts of text	SKIPSCAN, BEGINSCAN	no
saved symbol tables	intmods	no

Table B.3-1. MAINSAIL and C Compiletime Facilities

B.4. Control Structures

MAINSAIL and C both have large and flexible sets of control structures. Approximate correspondences are shown in Table B.4-1. In addition, MAINSAIL provides coroutines (interleaved executions) and exceptions (something like an interprocedure "goto"), as described in Section B.10.

MAINSAIL	C
BEGIN, END	{, }
IF, THEN, ELSE	if, else
CASE	switch
WHILE, FOR, DO, UNTIL	while, for, do
DONE	break (in loop)
CONTINUE	continue
RETURN	return
named DONE or CONTINUE	goto

Table B.4-1. MAINSAIL and C Control Structures

B.5. Operators

MAINSAIL and C both provide a large set of operators for use in constructing expressions. Correspondences are shown in Table B.5-1.

Most of the operations that do not have exact correspondences in the other language are easy to construct equivalents for using an additional operator or two. The exceptions on the MAINSAIL side are the "&" (string concatenation) and "^" (power) operators:

- "&": C's strings are not at all equivalent to MAINSAIL's. Because MAINSAIL manages the allocation of storage occupied by string text automatically, it can provide a simple concatenation operator. In C, the programmer must be sure to allocate sufficient space and to copy string text explicitly, deallocating space if it becomes unused.

- "^": The integer power operator is easy to write as a procedure in C. The power function is provided by the UNIX library function "pow".

MAINSAIL	C
:=	=
NOT	!
OR, AND	||, &&
=, NEQ, <, LEQ, >, GEQ	==, !=, <, <=, >, >=
TST	&
+, -	+, -
IOR, XOR, MSK	|, ^, &
*, / (DIV), MOD	*, /, %
SHL, SHR	<<, >>
IF ... THEN ... ELSE ...	... ? ... : ...
.+, .-, etc.	+=, -=, etc.

There are no simple C equivalents for the following MAINSAIL operators:

NTST, TSTA, NTSTA, MIN, MAX, CLR, &, ^

There are no simple MAINSAIL equivalents for the following C operators:

~, ++, --

Table B.5-1. MAINSAIL and C Operators

On the C side, "++" and "--" are frequently used and convenient operators. However, when used for string or arbitrary storage management, the equally convenient MAINSAIL procedures read, write, cRead, and cWrite are often approximate equivalents.

B.6. Data Types and Data Structures

The approximate correspondence between MAINSAIL's and C's simple data types is shown in Table B.6-1.

C and MAINSAIL have different models of data structures. C's allow for more possibilities, but there is no garbage collection; MAINSAIL's are somewhat restricted by the requirement

MAINSAIL	C
BOOLEAN	int
INTEGER	short
LONG INTEGER	long
REAL	float
LONG REAL	double
BITS	short
LONG BITS	long
ADDRESS	short *
CHARADR	char *
POINTER	short *, but collectable

MAINSAIL has no equivalent of C's char (character codes are treated as integers). C has no equivalent of MAINSAIL strings; char pointers or arrays provide some of the functionality.

Table B.6-1. MAINSAIL and C Simple Data Types

that the runtime system be able to keep track of allocated data and determine when the data become inaccessible.

C structure tags correspond approximately to MAINSAIL classes, and pointers to C arrays approximately to MAINSAIL arrays. However, all MAINSAIL records and arrays must be explicitly allocated at runtime; they are never allocated automatically on entry to a procedure, and are always accessed through pointers (the MAINSAIL "." operator is the equivalent of the C "->" operator; there is no equivalent of the C "." operator).

C pointers may point to any data type, but MAINSAIL pointers always point to records (the equivalent of structure objects). C's unions can allow different views of an object by imposing different storage templates on it. MAINSAIL has a somewhat similar notion in prefix classes, but the difference is that a MAINSAIL record is not just a series of storage locations that can be viewed however the program likes, but actually has a class associated with it when it is created. Consequently, some views of the record are "right" and others inherently "wrong".

A MAINSAIL array is a special kind of pointer that points to a series of elements of the same data type instead of to a record of heterogeneous fields. Like records, arrays must be allocated at runtime. Like records, arrays are tagged with information about their structure, and this information can be examined by a program (e.g., "a.ub1" is the upper bound of the first dimension of an array a). Character strings are not arrays, but have a special data type of their own.

By default, the MAINSAIL compiler emits code that checks at runtime that pointers used are non-null and that array element references are within the bounds of an array. C cannot provide such checks, since C structures are not tagged.

B.7. Strings

Many C programmers who learn MAINSAIL are baffled by the string data type. It is difficult for them to understand MAINSAIL strings as anything other than read-only arrays, but MAINSAIL strings are not arrays at all, and provide a number of facilities that arrays do not.

A string is implemented as a character address and a length (the combination is called a "string descriptor"). The character address usually points into an area of memory called "string space", which is managed by the MAINSAIL runtime system. Space is never allocated explicitly for a string in string space; it is always allocated and reclaimed automatically by the MAINSAIL runtime system.

No explicit facility is provided for accessing individual characters by position in the string, other than first and last; i.e., individual characters are not accessed by index (although subtstrings can be constructed using indices, and the first or last character taken). The first or last character can be removed from the string, or a character can be added to the beginning or end of the string, or two entire strings can be concatenated. It is not possible to change a character in the middle of a string; a new string must be constructed with the desired text and the string descriptor made to point at it.

The string rules ensure that unless a programmer "cheats" by using the charadr data type to change storage in an uncontrolled fashion, a string variable always points at the same text until explicitly changed. String operations on one string cannot change the text to which another string points. Strings thus do not act like pointers, since altering a field of a data structure using a pointer changes the field when accessed using any other pointer pointing to the same place. MAINSAIL ensures that when a string's text is modified in such a way that it cannot be shared by other string descriptors, the text is copied if necessary and the string descriptor made to point to the new location.

Many C programmers are tempted by the similarity of MAINSAIL's substring syntax to C's element access syntax to step through a string with code looking something like:

```
FOR i := 1 UPTO length(s) DO
    ... first(s[i FOR 1]) ...
```

It is more efficient, and produces more readable code, to use the system procedure cRead to read and remove the first character on each iteration of the loop:

```
WHILE s DO
    ... cRead(s) ...
```

Strings can likewise be built up sequentially using cWrite, which extends the space occupied by the string automatically as required.

B.8. Garbage Collection

Garbage collection is usually triggered automatically by the MAINSAIL runtime system, although there are mechanisms for suppressing or explicitly causing collections. A garbage collection rummages through memory reclaiming the space occupied by data sections, records, and arrays that are no longer referenced by accessible pointers. Garbage collections also reclaim unreferenced string space.

A collection may entail moving data structures about, so that pointers and the character pointer components of string descriptors may change in value following a collection. This is normally transparent to the programmer, except when using the data types address and charadr. Address and charadr are intended to point into static memory, and so are not updated if they point to collectable data. It is not usually necessary for the programmer to use addresses or charadrs that reference collectable data, so this is rarely a problem in practice.

Although garbage collection automatically frees the space occupied by inaccessible data, it is still a good idea to reclaim space explicitly by calling the system procedure "dispose" if the program's structure allows it. Judicious use of dispose can reduce the frequency and scope of garbage collections, which can be quite time consuming when there is a lot of garbage to collect.

B.9. Input and Output

MAINSAIL's I/O facilities were designed from the beginning for portability among operating systems. They were also designed to provide complete functionality, including random access to files. C provides some high-level I/O routines that are often not used by programs because they are inadequate; instead, programs must use the lower level provided by UNIX or the UNIX compatibility library.

The MAINSAIL Structure Blaster provides a way to read or write an entire data structure with a single procedure call. The tagged nature of MAINSAIL data structures makes it possible for MAINSAIL to find all relevant data from a single root pointer; this cannot be done in C.

MAINSAIL STREAMS provides a portable approach to low-level device and process control, along with a higher-level facility, RPC (Remote Procedure Calls), for interprocess communication.

B.10. Coroutines and Exceptions

Coroutines and exceptions are two methods of controlling a program's execution that have no standard C equivalents. It is difficult to duplicate their behavior in a language that does not have these facilities built in.

A coroutine is a context that preserves the state of a procedure so that its execution may be resumed at the preserved state by a procedure in some other coroutine. Each coroutine has its own procedure stack. The coroutines must cooperate; rescheduling of coroutines is done not by timer interrupt but by implicit or explicit calls to resume another coroutine. Careful use of coroutines can give the illusion of simultaneous execution of different programs in a single MAINSAIL process. MAINSAIL STREAMS can be used in conjunction with coroutines to achieve genuine simultaneous execution: several processes are set up on different machines, and a coroutine is set up to talk to each. The programming of a coroutine talking to a single remote process is simpler than having a single thread of execution talking to multiple processes.

Exceptions are something like the setjmp/longjmp facility provided by UNIX, except that they are more flexible. When an exception occurs, the exception context is preserved so that, in addition to aborting the code that caused the exception, the program has the option of resuming the code that caused the exception. It is also possible to propagate the exception; if a given handler does not know how to handle it, it is possible that a calling procedure does.

Exceptions are less error-prone than setjmp/longjmp's. Problems like those with accidental returns to nonexistent environments or with improperly restored register variables cannot occur in MAINSAIL exceptions.

B.11. Tools

The MAINSAIL environment contains a large set of highly portable tools. Among the most important of these for the programmer are MAINDEBUG, the MAINSAIL debugger, and MAINPM, the performance monitor. MAINDEBUG is a source-level debugger that interfaces conveniently to the MAINSAIL text editor, MAINEDIT (which includes a front end that emulates vi, the popular UNIX editor). MAINPM is useful for tracking time usage and memory consumption in a program.

Index

XIDAK, Inc., 530 Oak Grove Avenue, M/S 101, Menlo Park, CA 94025, (415) 324-8745