The MDL Programming Language Primer

Michael Dornbrook
Marc Blank

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

6. Simple Functions 27

6.1. General 27
6.2. Defining **FUNCTIONS** 27
6.3. Application of **FUNCTIONs**: Binding 28
6.4. **DEFINEing** Some Simple **FUNCTIONs** 30
6.5. Pretty-Printing 33
6.5.1. Editors and Pretty Printing 34
6.6. Loading a File 34

7. MDL TYPEs 35

7.1. **TYPEs** and **PRIMTYPEs** 35
7.2. Introduction to MDL Structures 36
7.3. The **TYPE?** Predicate 36
7.4. Printing of MDL Objects 37
7.5. Significance of **PRIMTYPEs / CHTYPE** 37
7.6. Creating new **TYPEs** 38

8. MDL Structures 41

8.1. Equality 41
8.2. PRIMTYPE LIST 41
8.2.1. Creating **LISTs** 43
8.2.2. **EVALing LISTs** 45
8.2.3. Manipulating **LISTs** 45
8.2.4. **FIXes** First in **FORMs** 52
8.2.5. **FORMs** 52
8.2.6. **FALSEs** 53
8.2.7. **SEGMENTs** 53
8.3. PRIMTYPE VECTOR 55
8.3.1. Creating **VECTORs** 55
8.3.2. **EVALing VECTORs** 56
8.3.3. Manipulating **VECTORs** 56
8.3.4. **UVECTORs** 58
8.4. PRIMTYPE STRING 59
8.4.1. **ASCII** 60
8.4.2. Creating **STRINGs** 60
8.4.3. **EVALing STRINGs** 61
8.4.4. Manipulating **STRINGs** 61
8.5. Building Large Structures 62
8.6. Searching Structures 62
8.7. Garbage: Quoting Structures 63
8.8. Garbage: Building Lists 63
8.9. Structured **NEWTYPEs** 65
8.10. Summary of MDL Structures 65
8.11. Practice Quiz 66

9. Programming Constructs 69

9.1. Boolean Operators 69

9.1.1. **NOT** 69
9.1.2. **AND** 69
9.1.3. **OR** 70
9.2. **COND** 70
9.2.1. Examples 71
9.3. Shortcuts with Conditionals 72
9.3.1. Using **AND** and **OR** with **CONDs** 72
9.3.2. Embedded Unconditionals 73
9.4. Examples 74

10. Looping **77**

10.1. **PROG** 77
10.2. **REPEAT** 78
10.3. Non-local **RETURNs**, etc. 78
10.4. **MAPF** 79
10.4.1. Looping Through a Structure 79
10.4.2. Other Than One Structure 80
10.4.3. Using Intermediate Results 81
10.4.4. **MAPRET** and **MAPSTOP** 81
10.4.5. **MAPR** 83
10.4.6. **MAPF/R Summary** 84
10.5. Looping vs. Recursion 85

11. Argument Lists in FUNCTIONs **87**

11.1. Arguments Not **EVAL**ed 87
11.2. Optional Arguments 88
11.3. Arbitrary Numbers of **EVAL**ed Arguments 88
11.4. Arbitrary Numbers of un-**EVAL**ed Arguments 89
11.5. Temporary Variables 90
11.6. Order of Evaluation in Argument Lists 91
11.7. Variable Declarations 91
11.8. Structures: **DECL**s and **NEWTYPE**s 93
11.8.1. To **NEWTYPE** or Not To **NEWTYPE** 94
11.9. Good Habits / Bad Habits 95
11.10. Review of Argument List Syntax 95

12. Input/Output **97**

12.1. Basics of I/O 97
12.2. Conversion I/O - Input 98
12.2.1. **READ** 98
12.2.2. **READCHR** 98
12.2.2.1. **NEXTCHR** 98
12.3. Conversion I/O - Output 98
12.3.1. **PRINT** 99
12.3.2. **PRIN1** 99
12.3.3. **PRINC** 99
12.3.4. **CRLF** 99

12.4. CHANNEL (the TYPE) 100
12.4.1. OPEN 100
12.4.2. FILE-EXISTS? 101
12.4.3. CLOSE 101
12.4.4. CHANLIST 101
12.4.5. INCHAN and OUTCHAN 101
12.5. End-of-File "Routine" 102
12.6. Additional I/O SUBRs 102
12.6.1. READSTRING 103
12.6.2. PRINTSTRING 104
12.7. SAVE Files 105
12.7.1. SAVE 105
12.7.2. RESTORE 105
12.8. PARSE, LPARSE, and UNPARSE 105
12.9. Other I/O Functions 106
12.9.1. FLOAD 106
12.9.2. SNAME 106
12.9.3. FILE-LENGTH 107
12.9.4. RESET 107
12.9.5. RENAME 107
12.10. Terminal CHANNELs 108
12.10.1. TYI 108

13. Making Tables 109

13.1. Use a LIST 109
13.2. Use a VECTOR 110
13.3. Use an ATOM 110
13.4. Use an Association 111
13.4.1. Hashing 112
13.5. Use an OBLIST 112
13.6. OBLISTs, READ, and PRINT 114

14. Debugging MDL Programs - An Introduction 115

14.1. Method 1: Start Over 116
14.2. Method 2: Forcing FRAMEs to Return Values 117
14.3. Method 3: Use EDIT to Repair your FUNCTIONs 118
14.4. Method 4: Altering FRAMEs / RETRY 120
14.5. Summary 120

Index 123

List of Figures

Figure 8-1: The MDL notion of equality is demonstrated in this figure, which shows the distinction between single-equal =? and double-equal ==?. 42
Figure 8-2: The LIST (1 2 3) 43
Figure 8-3: Removing a LIST element by moving only one pointer 43
Figure 8-4: REST of a LIST 46
Figure 8-5: PUTs into LISTs 46
Figure 8-6: Pointers *vs.* Structures 47
Figure 8-7: PUTREST 48
Figure 8-8: Removing an element from a LIST using PUTREST 49
Figure 8-9: Splicing LISTs together using PUTREST 50
Figure 8-10: The VECTOR [1 2 3 4] 55
Figure 8-11: REST of a VECTOR 57
Figure 8-12: BACK of a VECTOR 58
Figure 14-1: Diagram for the example in this chapter 121

Introduction

Over the years the original MDL (pronounced "Muddle") Primer by Greg Pfister [Pfister 72] became more and more a reference manual and less a Primer from which a novice could learn the language. Some of the text of the original has been re-used in this document, but much has been eliminated, changed, or re-ordered, and a reasonable amount of new material has been added. In particular, a number of figures and many more examples have been added to make some of the more difficult concepts easier to understand.

This Primer is intended as an introduction to MDL. After assimilating the information contained herein, you should be able to write very good programs. However, for any individual topic in the MDL Primer there is likely to be more information available in *The MDL Programming Language* [Galley 79] and *The MDL Programming Environment* [Lebling 80], and there are many topics in these documents which are not addressed in the Primer. Anyone who plans to do any serious work with MDL should read these documents.

One of the difficulties in writing a Primer is to make it useful to those who don't know anything at all about programming without boring those who know a lot of the basics. Hopefully those at both extremes will find this to be easy to read. If you are a complete novice, however, there may be some unfamiliar references and some material which doesn't make sense on your first reading.

Why MDL?

Many people ask this. It is often hard for those who use MDL to put into words their reasons for liking it. Those of us who use MDL are convinced that it is a better language than any other we've encountered. Unfortunately, very little has been done to convince others of this and spread the use of this marvelous tool.

MDL was created in the early 1970's by a group at the Dynamic Modelling/Computer Graphics division of MIT's Project MAC (later renamed the Laboratory for Computer Science). It is an offshoot of the original Lisp. There have been quite a few offshoots of Lisp in the past 10 years - MacLisp, InterLisp, Lisp Machine Lisp, Lisp1.5, UCI Lisp, Franz Lisp, etc., etc. - but none of them are like MDL.

Since MDL is a distant relative of Lisp and many of those first learning MDL have some familiarity with Lisp, a short comparison of the two languages follows. If you are not familiar with Lisp (or, better still, with <u>any</u> other languages) count your blessings (you don't have any bad habits to unlearn) and skip the following discussion.

MDL's similarities to Lisp: MDL shares the advantages of Lisp over the more popular languages such as Basic, Fortran, Cobol, Algol, Pascal, etc.

- It has an interpreter which allows real-time interaction and allows you to define and test individual functions separately.

- Its syntax is very simple.

- Any data object or function can be passed as an argument or returned as a value.

- It has list structures equivalent to Lisp's.

- Recursive functions can be written quite easily.

The similarities between MDL and Lisp are such that in many cases a few minor changes to Lisp code will convert it into working MDL code. Given the other features of MDL, no MDL programmer would write the program in the same Lisp style.

MDL's dissimilarities to Lisp: Many objections to Lisp are answered in MDL.

- Strongly typed languages provide much better error detection tools than Lisp. MDL allows declarations of all variable types to whatever level of complexity is desired. A variable can be declared to be one of several types.

- Recursion is a useful tool, but often is not a very efficient way to solve the problem. Lisp's motto "To iterate is human, to recurse divine," is not one of MDL's tenets. MDL allows recursion, but provides excellent facilities for iteration.

- MDL has a very powerful set of data structures - Lists, Strings, Vectors, and Uniform Vectors. Although lists are a very useful and flexible form of structure, they are certainly not optimal in all cases. MDL's various structures allow the user to save space and access time. MDL's structures are also "first class," in that the standard functions for manipulating data structures can be used on all of them equivalently.

- Probably the biggest complaint against Lisp-like languages is that they are unsuitable for "production programming" because they are too slow. MDL has an excellent compiler which as far as we know is the best compiler for a Lisp-like language. It produces machine code equivalent in efficiency to Fortran and Cobol, which are considered very efficient.

- MDL has a rich library of useful program aids. The editing and debugging functions are among the best. The package system allows building of very large programs from small sections, usually written by different people, without worrying about variable name conflicts.

- Probably the most distinctive feature of MDL is its mechanism for user-defined types, which is the best of any language with which we are familiar. User-defined types have been retrofitted on some of the newer versions of Lisp, but in most cases they can be used only with special functions and cannot be used in the same general way that Lists can.

Hopefully some of your questions have been answered and you have some ready answers when you get flak from your non-MDL programming friends. Learning MDL should be an enjoyable and worthwhile experience. Your reactions to this Primer and suggestions for changes are always welcome. Good luck!

Warning! You are about to embark on an undertaking fraught with peril. MDL programming has been proven to be habit-forming. Once you begin, you may find the habit hard to kick!

Acknowledgments

We are deeply indebted to our predecessors for their work on this topic: Greg Pfister, who wrote the original *A Muddle Primer* [Pfister 72], and Stuart Galley, who updated that document and added significantly to it to create *The MDL Programming Language* [Galley 79] document. Some of the text and examples of the original documents survive here, and some other material was simply rewritten in an order and style which we consider more comprehensible.

Special thanks to Chris Reeve, Dave Lebling, Stu Galley, Poh Lim, Thomas Michalek, Dave Scrimshaw, Tim Anderson, Mark Plotnick, and Prof. J.C.R. Licklider for their many comments and suggestions.

No document on MDL would be complete without acknowledging the "original implementors." If not for their inspiring work, this fine language would not exist. We are forever grateful to Gerald Sussman, Carl Hewitt, Chris Reeve, Dave Cressey, and Bruce Daniels. Thanks are also extended to the many unnamed hackers who have improved the language and the programming environment over the years.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract N00014-75-C-0661.

This document was prepared using Scribe and printed on the Xerox Dover printer.

(c) Copyright 1981 Massachusetts Institute
of Technology. All rights reserved.

1. Basic Interaction

The purpose of this chapter is to provide you with that minimal amount of information needed to experiment with MDL (pronounced, affectionately, as Muddle) while reading this document. It is strongly recommended that you do experiment, especially upon reaching chapter 6 (page 27) (Simple Functions).

1.1. Loading MDL

First, catch your rabbit. Somehow get the interpreter running -- the program in the file **SYS:TS MDL** in the ITS version or **SYS:MDL.SAV** in the Tenex version or **SYS:MDL.EXE** in the Tops-20 version. [Just type **:MDL** to ITS, **MDL** or **MUDDLE** to Tops-20.] The interpreter will then type

```
MUDDLE nnn IN OPERATION.
LISTENING-AT-LEVEL 1 PROCESS 1
```

and then wait for you to type something.

The program which you are now running is an interpreter for the language MDL. All it knows how to do is interpret MDL expressions. There is no special "command language"; you communicate with the program -- make it do things for you -- by actually typing legal MDL expressions, which it then interprets. Everything you can do at a terminal can be done in a program, and vice versa, in exactly the same way.

The program will be referred to as just "MDL" (or "the interpreter") from here on.

1.2. Typing

Typing a character at MDL normally just causes that character to be echoed (printed on your terminal) and remembered in a buffer. The only characters for which this is normally not true act as follows:

Typing the "Escape" or "Alt-Mode" key, which we will always refer to as $ (dollar-sign), causes

MDL to echo dollar-sign and causes the contents of the buffer (the characters which you've typed) to be interpreted as an expression(s) in MDL. When this interpretation is done, the result will be printed and MDL will wait for more typing.

Typing the rubout character (DEL in the ITS and Tops-20 versions, control-A in the Tenex version) causes the last character in the buffer -- the one most recently typed -- to be thrown away (deleted). If you now immediately type another rubout, once again the last character is deleted -- namely, the second most recently typed. Etc. The character deleted is echoed, so you can see what you're doing. On some "display" terminals, rubout will "echo" by causing the deleted character to disappear. If no characters are in the buffer, rubout echoes as carriage-return line-feed.

Typing ↑@ (control-atsign) deletes everything you have typed since the last $, and prints a carriage-return line-feed.

Typing ↑D (control-D) causes the current input buffer to be typed back out at you. This allows you to see what you really have, without the confusing re-echoed characters produced by rubout.

Typing ↑L (control-L) produces the same effect as typing ↑D, except that, if your terminal is a "display" terminal (for example, VT100, VT52, H19, ...), the screen is cleared before the input buffer is retyped.

Typing ↑G (control-G) causes MDL to stop whatever it is doing and act as if an error had occurred (section 1.3 (page 9)). ↑G is generally most useful for temporary interruptions to check the progress of a computation. ↑G is "reversible" -- that is, it does not destroy any of the "state" of the computation it interrupts. To "undo" a ↑G, type the characters

```
<ERRET T>$
```

(This is discussed more fully far below, in chapter 14, page 115 (Debugging MDL Programs).)

Typing ↑S (control-S) causes MDL to <u>throw away</u> what it is currently doing and return to a normal "listening" state. (In the Tenex and Tops-20 versions, ↑O also should have the same effect.) ↑S is generally most useful for aborting infinite loops and similar terrible things. ↑S <u>destroys</u> whatever is going on, and so it is <u>not</u> reversible.

Most expressions in MDL include "brackets" (generically meant) that must be correctly paired and nested. If you end your typing with the pair of characters !$ (exclamation-point ESC), all currently unpaired brackets (but not double-quotes, which bracket strings of characters) will automatically be paired and interpretation will start. Without the !, MDL will just sit there waiting for you to pair them. If you have improperly nested parentheses, brackets, etc., within the expression you typed, an error will occur, and MDL will tell you what is wrong.

Once the brackets are properly paired and $ (ESC) is typed, MDL will immediately echo carriage-return and line-feed, and the next thing it prints will be the result of the evaluation. Thus, if a plain $ is not so echoed, you have some expression unclosed. In that case, if you have not typed any

characters beyond the **$**, you can usually rub out the **$** and other characters back to the beginning of the unclosed expression. Otherwise, what you have typed is beyond the help of rubout and **↑@**; if you want to abort it, use **↑S**.

MDL accepts and distinguishes between upper and lower case. All "built-in functions" must be referenced in upper case.

1.3. Errors -- Simple Considerations

When MDL decides for some reason that something is wrong, the standard sequence of evaluation is interrupted and an error function is called. This produces the following terminal output:

> ***ERROR***
> *often-hyphenated-reason*
> *function-in-which-error-occurred*
> **LISTENING-AT-LEVEL** *integer* **PROCESS** *integer*

You can now interact with MDL as usual, typing expressions and having them evaluated. There exist facilities (built-in functions) allowing you to find out what went wrong, restart, or abandon whatever was going on. In particular, you can recover from an error -- that is, undo everything but side effects and return to the initial typing phase -- by typing the following first line, to which MDL will respond with the second line:

```
<ERRET>$
LISTENING-AT-LEVEL 1 PROCESS 1
```

If you type the following line while still in the error state (before **<ERRET>**), MDL will print the **FRAMES** it went through to evaluate the function:

```
<FRAMES>$
```

Typing **FR&** (pronounced 'frampersand') instead of **FRAMES** will cause MDL to print a condensed, usually more readable output.

This will also be explained in chapter 14.

2. MDL Basics

In a general sense, when you are interacting with MDL, you are dealing with a world inhabited only by a particular set of things: MDL objects.

2.1. Introduction to MDL TYPES

MDL objects are best considered as abstract entities with abstract properties. The properties of a particular MDL object depend upon the class of MDL objects to which it belongs. This class is known as the **TYPE** of the object, and every MDL object has one. Easily recognized **TYPE**s include **FIX** (integers) and **FLOAT** (real numbers). Examples of these might be **1** and **2.67**, respectively. An abbreviation often used is to refer to "a **FIX**" when referring to a MDL object whose **TYPE** is **FIX**. For example, **1** is a **FIX** and **2.67** is a **FLOAT**.

MDL **TYPE**s can be divided into two general classes: those with internal structure and those without internal structure. The former will be referred to as being *structured*. *Structured* objects are those which can be thought of as an ordered series of items held together in some way. There are a number of ways in which these items can be held together, and each of these is represented by a series of MDL objects between a set of matched brackets (e.g. **<>**, **()**, **[]**, **{}** **""**). As will be seen later, each bracket type represents a different **TYPE** of MDL object, and some represent different ways of internally storing the series of objects. Depending on the application, one of these may be more suitable than another.

Here are some MDL objects which are not *structured*:

```
20
20.0
TWENTY
```

The first two are examples of **TYPE**s **FIX** and **FLOAT**, as noted above. The last is an **ATOM**, roughly speaking an identifier or a variable, and will be discussed in Chapter 3.

Here are some MDL objects which are *structured*:

```
<+ 1 2>
(+ 1 2)
[+ 1 2]
"+ 1 2"
```

These represent very similar notions: an ordered series of the MDL objects +, 1, and 2, the first of these being an ATOM and the rest FIXes. These brackets correspond to the TYPEs FORM, LIST, and VECTOR. The first of these, a FORM, is central to MDL, as it represents the application of a *function* to arguments. The others will be considered later.

2.2. Printing of MDL Objects

We have already seen the printed representation of some MDL objects: FIXes, FLOATs, FORMs, LISTs and VECTORs. As will be mentioned later, MDL allows an almost unlimited number of data types. Obviously, there are not enough bracket types to make each data type recognizable. Therefore, most MDL types have a kind of generalized way of printing. This format is like this:

#type-name value

where *type-name* is the name of a MDL TYPE and *value* describes the 'value' of the object. Suffice it for now to say that an object which prints like:

```
#FALSE ()
```

is of TYPE FALSE and an object which prints like

```
#MUMBLE [1 2 3]
```

is of TYPE MUMBLE.

2.3. MDL FORMS

A FORM in MDL is printed as: an open angle bracket (<), the name of the *function* to be applied, the arguments to which the *function* is being applied, and finally a closing angle bracket (>). MDL's angle brackets are one of its distinguishing features (almost as distinctive as Lisp's parentheses).

MDL has a large number of built-in *functions*. These are usually of TYPE SUBR (short for subroutine). For example:

```
<+ 1 2 3>
```

will, when given to the MDL interpreter, return **6**. The way in which the name for a *function*, in this case **+**, is associated with its functional part (i.e. in this case, the *thing* which actually performs the addition) is described later. Suffice it for now to say that these *functions* can be referenced by their name (an **ATOM**), as was done in the example.

2.4. Prefix Notation

MDL is a distant relative, a much-improved descendant of LISP. The "desirable features" of LISP were included in MDL. One of those features, prefix notation, you have just seen.

Prefix notation, sometimes referred to as Polish notation, is different from the infix notation of ordinary arithmetic and reverse-Polish notation of some calculators. Below are some examples of equivalents in infix and prefix notation:

```
4 + 7
<+ 4 7>

8 - 6
<- 8 6>

6 - (3 + 2)
<- 6 <+ 3 2>>

9 + (4 * 6 - 6 / 3)
<+ 9 <- <* 4 6> </ 6 3>>>

7 + 3 + 4 + 8 + 11
<+ 7 3 4 8 11>
```

It will take you some time to become accustomed to prefix notation. One thing you will have to keep in mind is balancing of brackets. Notice that with prefix notation an operator can take an arbitrary number of arguments and that the nesting is never ambiguous (*i.e.* the parentheses of infix notation are not necessary).

2.5. Evaluation of FORMS

Evaluation of a MDL **FORM** proceeds from left to right. The first item is the name of the function which will be applied to the arguments which follow. The arguments may themselves be **FORM**s which

will be evaluated in the same way. For example, this **FORM**:

```
<+ <+ 1 2> 3>
```

when evaluated will apply the addition function to the evaluation of the first argument (which, since it is itself a form, will be recursively evaluated until it returns a value) and then to the evaluation of the second argument. The arguments may be much more complex than this and require many levels of evaluation before a result is returned. It is important to note that unlike many other languages, every evaluation has a resulting value. As we will see, even such operations as printing or setting the values of variables return values.

2.6. Introduction to Truth

In MDL, anything which does not evaluate to an object of **TYPE FALSE** is considered true. If an expression returns false, MDL usually prints it as **#FALSE ()**.

3. Read, Evaluate, and Print

3.1. General

Once you type **$** and all brackets are correctly paired and nested, the current contents of the input buffer go through processing by three functions successively: first **READ**, which passes its output to **EVAL** ("evaluate"), which passes its output to **PRINT**, which types its output on the terminal.

Functionally,

READ: printed representations --> MDL objects

EVAL: MDL objects --> MDL objects

PRINT: MDL objects --> printed representations

That is, **READ** takes ASCII text, such as is typed in at a terminal, and creates the MDL objects represented by that text. **PRINT** takes MDL objects, creates ASCII text representations of them, and types them out. **EVAL**, which is the really important one, performs transformations on MDL objects.

3.2. EVAL and TYPEs

The laws of the MDL world are defined by **EVAL**. In a very real sense, **EVAL** is the only MDL object which "acts", which "does something". In "acting", **EVAL** is always "following the directions" of some MDL object. Every MDL object should be looked upon as supplying a set of directions to **EVAL**; what these directions are depends heavily on the **TYPE** of the MDL object.

Since **EVAL** is so ever-present, an abbreviation is in order: "evaluates to *something*" or "**EVAL**s to *something*" should be taken as an abbreviation for "when given to **EVAL**, causes **EVAL** to return *something*".

3.3. Example (TYPE FIX)

```
1$
1
```

The following has occurred:

First, **READ** recognized the character **1** as the representation for an object of **TYPE FIX**, in particular the one which corresponds to the integer one. (**FIX** means integer, because the decimal point is understood always to be in a fixed position: at the right-hand end.) **READ** built the MDL object corresponding to the decimal representation typed, and returned it.

Then **EVAL** noted that its input was of **TYPE FIX**. An object of **TYPE FIX** evaluates to itself, so **EVAL** returned its input undisturbed.

Then **PRINT** saw that its input was of **TYPE FIX**, and printed on the terminal the decimal character representation of the corresponding integer.

3.4. Example (TYPE FLOAT)

```
1.0$
1.0
```

What went on was entirely analogous to the preceding example, except that the MDL object was of **TYPE FLOAT**. (**FLOAT** means a real number (of limited precision), because the decimal point can float around to any convenient position: an internal exponent part tells where it "really" belongs.)

3.5. FIXes and FLOATs versus READ: Specifics

3.5.1. READ and FIXed-point Numbers

READ considers any grouping of characters which are solely digits to be a **FIX**, and the radix of the representation is decimal (*i.e.* the base is 10) by default. A - (hyphen) immediately preceding such a grouping represents a negative **FIX**. The largest **FIX** representable on the PDP-10 is two to the 35th power minus one, or 34,359,738,367 (decimal); the smallest is one less than the negative of that number. If you attempt to type in a **FIX** outside that range, **READ** converts it to a **FLOAT**; if a program you write attempts to produce a **FIX** outside that range, an overflow error will occur (unless overflow errors are disabled).

3.5.2. READ and PRINT versus FLOATing-point Numbers

PRINT can produce, and **READ** can understand, two different formats for objects of **TYPE FLOAT**. The first is "decimal-point" notation, the second is "scientific" notation. Decimal radix is always used for representations of **FLOAT**s.

"Decimal-point" notation for a **FLOAT** consists of an arbitrarily long string of digits containing one **.** (period) which is followed by at least one digit. **READ** will make a **FLOAT** out of any such object, with a limit of precision of one part in 2 to the 27th power. (**FIX**ed and **FLOAT**ing-point numbers are stored in one 36-bit PDP-10 word. **FLOAT**ing-point numbers give up precision to gain their greater range.)

"Scientific" notation consists of:

1. a number, the mantissa

2. immediately followed by **E** or **e** (upper or lower case letter E),

3. immediately followed by an exponent,

where the mantissa is an arbitrarily long string of digits, with or without a decimal point (see following note); and the "exponent" is up to two digits worth of **FIX**. This notation represents the "number" to the "exponent" power of ten. Note: if the mantissa as above would by itself be a **FIX**, and if the "exponent" is positive, and if the result is within the allowed range of **FIX**es, then the result will be a **FIX**. For example, **READ** understands **10E1** as **100** (a **FIX**), but **10E-1** as **1.0000000** (a **FLOAT**).

The largest-magnitude **FLOAT** which can be handled without overflow is **1.7014118E+38** (decimal radix). The smallest-magnitude **FLOAT** which can be handled without underflow is **.14693679E-38**.

Examples:

```
1.001$
1.001000


.001$
1.0E-3


143E2$
14300


12345678912345$
1.2345678E+12
```

4. Atoms and Their Values

4.1. Example (TYPE ATOM, PNAME)

In the previous chapter, the handling of **FIX**ed and **Floating** point numbers by **READ**, **EVAL**, and **PRINT** was discussed. If you type:

```
GEORGE$
GEORGE
```

a lot more happens.

READ noted that what was typed had no special meaning, and therefore assumed that it was the representation of an object of **TYPE ATOM**. ("Atom" means "more or less indivisible.") **READ** therefore attempted to look up the representation in a table it keeps for such purposes. If **READ** finds an **ATOM** in its table whose representation matches the representation just received, that **ATOM** is returned as **READ**'s value. If the look-up fails, **READ** creates a new **ATOM**, puts it in the table with the representation read, and returns the new **ATOM**. Nothing which could in any way be referenced as a legal "value" is attached to the new **ATOM**. The initially-typed representation of an **ATOM** becomes its **PNAME**, meaning its name for **PRINT** (PRINT NAME). One often abbreviates "object of **TYPE ATOM** with **PNAME** *name*" by saying "**ATOM** *name*". There is a reason for making this careful distinction. Unlike other languages where atoms are names associated with values, a MDL **ATOM** is an object which may have values (global and/or local) but which is distinct from its value(s).

EVAL, given an **ATOM**, returned just that **ATOM**.

PRINT, given an **ATOM**, typed out its **PNAME**.

4.2. READ and PNAMEs

The question "what is a legal **PNAME**?" is actually not a reasonable one to ask; any non-empty string of arbitrary characters can be the **PNAME** of an **ATOM**. However, some **PNAME**s are easier to type to **READ** than others. But even the question "what are easily typed **PNAME**s?" is not too reasonable, because: **READ** decides that a group of characters is a **PNAME** by default; if it can't possibly be anything else, it's a **PNAME**. So, the rules governing the specification of **PNAME**s are messy, and best

expressed in terms of what is not a **PNAME**. For simplicity, you can just consider any uninterrupted group of upper- and lower-case letters and (customarily) hyphens to be a **PNAME**; that will always work. If, for some reason, you need to know all the gory details about legal **PNAME**s, see Subsection 2.6.3 of *The MDL Programming Language* [Galley 79].

4.3. Values of ATOMS

4.3.1. General

Typing **GEORGE** to the MDL interpreter and causing it to create the **ATOM** with **PNAME GEORGE** does not appear to be very useful. **ATOM**s in MDL serve as variables and as names for functions and data structures. They are definitely useful.

The **ATOM** has itself as its value. There are two additional kinds of "value" which can be attached to an **ATOM**. An **ATOM** can have either, both, or neither. They interact in no way. These two additional values are referred to as the local value and the global value of an **ATOM**. The functions which reference the local and global values of an **ATOM**, and some of the characteristics of local versus global values, follow.

4.3.2. SETG

A global value can be assigned to an **ATOM** by the **SUBR SETG** ("set global," pronounced 'set-gee'), as in

```
<SETG atom any>
```

where *atom* must **EVAL** to an **ATOM**, and *any* can **EVAL** to anything. **EVAL** of the second argument becomes the global value of **EVAL** of the first argument. The value returned by the **SETG** is its second argument, namely the new global value of *atom*.

Examples:

```
<SETG FOO <SETG BAR 469>>$
469
```

The above made the global values of both the **ATOM FOO** and the **ATOM BAR** equal to the **FIX**ed-point number **469**.

```
<SETG BAR FOO>$
FOO
```

That made the global value of the **ATOM BAR** equal to the **ATOM FOO**.

4.3.3. GVAL

The **SUBR GVAL** ("global value") is used to reference the global value of an **ATOM**.

```
<GVAL atom>
```

returns as a value the global value of *atom*. If *atom* does not evaluate to an **ATOM**, or if the **ATOM** to which it evaluates has no global value, an error occurs.

GVAL applied to an **ATOM** anywhere, in any function, will return the same value. Any **SETG** anywhere changes the global value for everybody. Global values are context-independent.

READ understands the character **,** (comma) as an abbreviation for an application of **GVAL** to whatever follows it. **PRINT** always translates an application of **GVAL** into the comma format. The following are absolutely equivalent:

```
,atom        <GVAL atom>
```

Assuming the examples in section 4.3.2 (page 20) were carried out in the order given, the following will evaluate as indicated:

```
,FOO$
469
<GVAL FOO>$
469
,BAR$
FOO
,,BAR$
469
```

4.3.4. SET

The **SUBR SET** is used to assign a local value to an **ATOM**. Applications of **SET** are of the form

```
<SET atom any>
```

SET returns **EVAL** of *any* just as **SETG** does.

Examples:

```
<SET BAR <SET FOO 100>>$
100
```

Both **BAR** and **FOO** have been given local values equal to the **FIX**ed-point number **100**.

```
<SET FOO BAR>$
BAR
```

FOO has been given the local value **BAR**.

Note that neither of the above did anything to any global values **FOO** and **BAR** had or might have had.

4.3.5. LVAL

The **SUBR LVAL** is used to return the local value of an **ATOM**. As with **GVAL**, **READ** understands an abbreviation for an application of **LVAL**: the character **.** (period), and **PRINT** produces it. The following two representations are equivalent, and when **EVAL** operates on the corresponding MDL object, it returns the current local value of *atom*:

```
<LVAL atom>      .atom
```

(Note: you will generally hear **.FOO** pronounced as 'dot-foo'). Assume all of the previous examples in this chapter have been done. Then the following evaluate as indicated:

```
.BAR$
100
<LVAL BAR>$
100
.FOO$
BAR
..FOO$
FOO
...FOO$
469
```

5. Built-in Functions

5.1. Evaluation of FORMs

EVAL applied to a **FORM** acts as if following these directions:

First, examine the *func* (first member) of the **FORM**. If it is an **ATOM**, look at its **GVAL**. If it is not an **ATOM**, **EVAL** it and look at the result of the evaluation. If what you are looking at is not something which can be applied to arguments, complain (via the **ERROR** function). Otherwise, inspect what you are looking at and follow its directions in evaluating or not evaluating the arguments and then "apply the function" -- that is, **EVAL** the body of the object gotten from *func*.

5.2. Built-in Functions (TYPE SUBR, TYPE FSUBR)

The built-in functions of MDL come in two varieties: those which have all their arguments **EVAL**ed before operating on them (**TYPE SUBR**, for "subroutine", pronounced 'subber') and those which have none of their arguments **EVAL**ed (**TYPE FSUBR**, historically from Lisp [Weinreb 78], pronounced 'effsubber,' for 'funny-**SUBR**'). Collectively they will be called **F/SUBR**s, although that term is not meaningful to the interpreter. See Appendix 2, Predefined Subroutines, in *The MDL Programming Language* [Galley 79] manual for a listing of all **F/SUBR**s and short descriptions. The term "Subroutine" will be used herein to mean both **F/SUBR**s and compiled user functions.

Unless otherwise stated, every MDL built-in Subroutine mentioned is of **TYPE SUBR**. Also, when it is stated that an argument of a **SUBR** must be of a particular **TYPE**, note that this means that **EVAL** of the argument must be of the particular **TYPE**.

Another convenient abbreviation which will be used is "the **SUBR** *pname*" in place of "the **SUBR** which is initially the **GVAL** of the **ATOM** of **PNAME** *pname*". "The **FSUBR** *pname*" will be used with a similar meaning. These distinctions are necessary. The **SUBR** is actually the global value of the **ATOM** of **PNAME** *pname*. The important point is that the **ATOM** effectively points at the "real *function*." For instance,

```
<GVAL SET>$
#SUBR *000000746516*
```

If you were so inclined, you could change the **ATOM** which points to a given **FUNCTION** or have many **ATOM**s point to the same **FUNCTION**. All built-in **SUBR**s and **FSUBR**s shall be referred to in this book by the **ATOM** which points to them when MDL starts up. The point is that there is nothing sacred about these names, but for clarity's sake it is recommended that you not rename them.

5.3. Examples (+ and FIX; Arithmetic)

```
<+ 2 4 6>$
12
```

The **SUBR** + adds numbers. Most of the usual arithmetic functions are MDL **SUBR**s: **+**, **-**, *****, **/**, **MIN**, **MAX**, **MOD**, **SIN**, **COS**, **ATAN**, **SQRT**, **LOG**, **EXP**, **ABS**. (See Appendix 2 of *The MDL Programming Language* [Galley 79] manual for short descriptions of these.) All except **MOD**, which wants **FIX**es, are indifferent as to whether their arguments are **FLOAT** or **FIX** or a mixture. In the last case, they exhibit "contagious **FLOAT**ing": one argument of **TYPE FLOAT** forces the result to be of **TYPE FLOAT**.

```
<FIX 1.0>$ 1
```

The **SUBR FIX** explicitly returns a **FIX**ed-point number corresponding to a **FLOAT**ing-point number (it truncates). The **SUBR FLOAT** returns the **FLOAT**ing point number equivalent to its argument.

```
<+ 5 <* 2 3>>$
11
<SQRT <+ <* 3 3> <* 4 4>>>$
5.0
<- 5 3 2>$
0
<- 5>$
-5
<MIN 1 2.0>$
1.0
</ 11 7 2.0>$
0.5
```

Note this last result: the division of two **FIX**es gives a **FIX** with truncation, not rounding, of the remainder; the intermediate result remains a **FIX** until a **FLOAT** argument is encountered.

5.4. Arithmetic: Details

+, -, *, /, **MIN**, and **MAX** all take any number of arguments, doing the operation with the first argument and the second, then with that result and the third argument, etc. If called with no arguments, each returns the identity for its operation (0, 0, 1, 1, the greatest FLOAT, and the least FLOAT, respectively); if called with one argument, each acts as if the identity and the argument had been supplied. They all will cause an overflow or underflow error if any result, intermediate or final, is too large or too small for the machine's capacity. Examples:

```
<+>$          </>$          </ 3.0>$          <- 2>$
0             1             0.33333333        -2
```

One arithmetic function that always requires some discussion is the pseudo-random-number generator. MDL's is named **RANDOM**, and it always returns a FIX, uniformly distributed over the whole range of FIXes. Example ("pick a number from one to ten"):

```
<+ 1 <MOD <RANDOM> 10>>$
4
```

5.5. Simple Predicates

The best analogy for a predicate in MDL (or LISP) is the predicate of an English-language question such as "Is John taller than Jim?" MDL answers such a question with true or false. If there is no other useful information to return, MDL will return T for true (á la LISP) or **#FALSE ()** for false.

The MDL predicate 0? takes one argument which can be either a FIX or a FLOAT. It evaluates to T only if its argument is exactly equal to 0 or 0.0.

```
<0? 1.2>$
#FALSE ()
```

The predicate 1? evaluates to T only if its argument is exactly equal to 1 or 1.0. The predicate G? takes two arguments, which again can be either FIXes or FLOATs. It evaluates to T only if the first argument is algebraically greater than the second. L=? is the Boolean complement of G?; that is, it is T only if the first argument is not algebraically greater than the second.

```
<L=? 3 4>$
T
```

Similarly, L? evaluates to T only if its first argument is algebraically less than its second argument. G=? is the Boolean complement of L?.

==? takes two arguments of <u>any</u> TYPE. In the case of arguments which are FIXes or FLOATs, it

returns T for two FIXes of the same value or for two FLOATs of exactly the same value. A FIX can never be ==? to a FLOAT.

```
<==? 17 17>$
T
<==? 1.0 1>$
#FALSE ()
```

To compare a FIX to an equivalent FLOAT, the SUBRs FIX or FLOAT are used:

```
<SET A 17>$
17
<SET B 17.0>$
17.0
<==? .A <FIX .B>>$
T
<==? <FLOAT .A> .B>$
T
```

N==? is the Boolean complement of ==?.

GASSIGNED? checks whether an ATOM has been assigned a global value.

```
<GASSIGNED? GAFWEEP>$
#FALSE ()
<SETG GAFWEEP 4023>$
4023
<GASSIGNED? GAFWEEP>$
T
```

ASSIGNED? is the corresponding predicate which checks whether an ATOM has been assigned a local value.

If you wish to compare the LVALs of two ATOMs, A and B, where the LVAL of A is known to be a FIX and the LVAL of B is known to be a FLOAT, use the SUBRs FIX or FLOAT:

```
<==? <FLOAT .A> .B>$
```

or

```
<==? .A <FIX .B>>$
```

6. Simple Functions

6.1. General

The MDL equivalent of a "program" (uncompiled) is an object of **TYPE FUNCTION**. Actually, full-blown "programs" are usually composed of sets of **FUNCTION**s, with most **FUNCTION**s in the set acting as "subprograms".

A **FUNCTION** may be considered to be a **SUBR** or **FSUBR** which you yourself define. It is "run" by using a **FORM** to apply it to arguments (for example, <*function arg-1 arg-2* ... >), and it always "returns" a single object, which becomes the value of the **FORM** that applied it. The single object may be ignored by whatever "ran" the **FUNCTION** (equivalent to "returning no value"), or it may be a structured object containing many objects (equivalent to "returning many values"). MDL is an "applicative" language, in contrast to "imperative" languages such as Fortran. In MDL, it is impossible to return values through arguments in the normal case (i.e. "call by name"); they are returned normally as the value of the **FORM** itself, or as side effects to structured objects or global values.

In this chapter a simple subset of the **FUNCTION**s you can write is presented, namely **FUNCTION**s which "act like" **SUBR**s with a fixed number of arguments. While this class corresponds to about 90% of the **FUNCTION**s ever written, you won't be able to do very much with them until you read further and learn more about MDL's control and manipulatory machinery. However, all that machinery is just a bunch of **SUBR**s and **FSUBR**s, and you already know how to "use" them; you just need to be told what they do. Once you have **FUNCTION**s under your belt, you can immediately make use of everything presented from this point on in this document. In fact, we recommend that you do so.

6.2. Defining FUNCTIONS

```
<DEFINE SQUARE (X) <* .X .X>>$
SQUARE
```

DEFINE is a MDL **FSUBR** (remember that **FSUBR**s have <u>none</u> of their arguments **EVAL**ed) for defining your own **FUNCTION**s. It takes an **ATOM** as the "name" for the **FUNCTION**, a list of arguments, and the **FORM**s which make up the body of the **FUNCTION**. **DEFINE SETG**s **EVAL** of its first argument (the **ATOM**) to an object of **TYPE FUNCTION** made from the other arguments and returns **EVAL** of the

first argument (the **ATOM** "naming" the **FUNCTION**).

If **EVAL** of **DEFINE**'s first argument already has a **GVAL**, **DEFINE** produces an error. This helps to keep you from accidentally redefining things -- such as MDL **SUBR**s and **FSUBR**s (if you want to be able to redefine without getting this error, type **<SET REDEFINE T>**. The **ATOM SQUARE** has been **SETG**ed to the **FUNCTION** which computes the square of a number. To use **SQUARE**, apply it to an argument in a **FORM**:

```
<SQUARE 5>$
25
<SQUARE 1.5>$
2.25
```

Using **SQUARE** with the wrong type of argument (anything other than a **FIX** or **FLOAT**) will produce an error. Using **SQUARE** with the wrong number of arguments (anything other than one) will also produce an error.

Taking the **GVAL** of **SQUARE** will show you what a **FUNCTION** looks like:

```
,SQUARE$
#FUNCTION ((X) <* .X .X>)
```

What **DEFINE** did was to **SETG SQUARE** to **#FUNCTION ((X) <* .X .X>)**. You could define a **FUNCTION** the same way, if you wished, or you could apply the **FUNCTION** directly:

```
<#FUNCTION ((X) <* .X .X>) 5>$
25
<#FUNCTION ((X) <* .X .X>) 1.5>$
2.25
```

Obviously, this would become quite tedious.

6.3. Application of FUNCTIONs: Binding

In order to make clear exactly what is happening in each of the examples in this section, **FUNCTION**s will be applied in the tedious, non-standard method just shown.

FUNCTIONs, like **SUBR**s and **FSUBR**s, are applied using **FORM**s. So,

```
<#FUNCTION ((X) <* .X .X>)  5>$
25
```

applied the indicated **FUNCTION** to **5** and returned **25**.

What **EVAL** does when applying a **FUNCTION** is the following:

1. Create a "world" in which the **ATOMs** of the argument **LIST** have been <u>SET</u> to the values to which the **FUNCTION** was applied, and all other **ATOMs** have their original values. This is called "binding". (In the above, this is a "world" in which **X** is **SET** to **5**.)

2. In that new "world", evaluate all the objects in the body of the **FUNCTION**, one after the other, from first to last. (In the above, this means evaluate <* .X .X> in a "world" where **X** is **SET** to **5**.)

3. Throw away the "world" created, and restore the **LVALs** of all **ATOMs** bound in this application of the **FUNCTION** to their originals (if any). This is called "unbinding". (In the above, this simply gives **X** back the local value, if any, that it had before binding.)

4. Return as a value the <u>last</u> <u>value</u> <u>obtained</u> when the **FUNCTION**'s body was evaluated in step (2). (In the above, this means return **25** as the value.)

The fact that such "worlds" are separate from the **FUNCTIONs** which cause their generation means that <u>all</u> MDL **FUNCTIONs** can be used recursively. (For those of you who understand the term, MDL is "dynamically scoped.")

The only thing that is at all troublesome in this sequence is the effect of creating these new "worlds", in particular, the fact that the <u>previous</u> world is restored. This means that if, inside a **FUNCTION**, you **SET** one of its argument **ATOMs** to something, that new **LVAL** will <u>not</u> be remembered when **EVAL** leaves the **FUNCTION**. However, if you **SET** an **ATOM** which is <u>not</u> in the argument **LIST** (or **SETG** <u>any</u> **ATOM**) the new local (or global) value <u>will</u> be remembered. Examples:

```
<SET X 0>$
0
<#FUNCTION ((X) <SET X <* .X .X>>)  5>$
25
.X$
0
```

On the other hand,

```
<SET Z 0>$
0
<#FUNCTION ((X) <SET Z <* .X .X>>)  5>$
25
.Z$
25
```

By using PRINT as a SUBR, we can "see" that an argument's LVAL really is changed while EVALuating the body of a FUNCTION:

```
<SET X 5>$
5
<#FUNCTION ((X) <PRINT .X> <+ .X 10>)  3>$

3 13
.X$
5
```

The first number after the application FORM was typed out by the PRINT; the second is the value of the application.

Remembering that LVALs of ATOMs not in argument LISTs are not changed, we can reference them within FUNCTIONs, as in

```
<SET Z 100>$
100
<#FUNCTION ((Y) </ .Z .Y>)  5>$
20
```

ATOMs used like Z in the above examples are referred to as "free variables". The use of free variables, while often quite convenient, is rather dangerous unless you know exactly how a FUNCTION will always be used: if a FUNCTION containing free variables is used within a FUNCTION within a FUNCTION within . . ., one of those FUNCTIONs might just happen to use your free variable in its argument LIST, binding it to some unknown value and possibly causing your use of it to be erroneous. Please note that "dangerous", as used above, really means that it may be effectively impossible (1) for other people to use your FUNCTIONs, and (2) for you to use your FUNCTIONs a month (two weeks?) later.

6.4. DEFINEing Some Simple FUNCTIONs

Using SQUARE as defined above, let's DEFINE a FUNCTION to compute the length of the hypotenuse of a right triangle given the lengths of the two sides:

```
<DEFINE HYPOT (SIDE-1 SIDE-2)
        <SQRT <+ <SQUARE .SIDE-1> <SQUARE .SIDE-2>>>>$
HYPOT
<HYPOT 3 4>$
5.0
```

SQRT is the **SUBR** which returns the square root of its argument. It always returns a **FLOAT**.

A whimsical **FUNCTION**:

```
<DEFINE ONE (THETA)
        <+ <SQUARE <SIN .THETA>>
           <SQUARE <COS .THETA>>>>$
ONE
<ONE 5>$
0.99999994
<ONE 0.23>$
0.99999999
```

ONE always returns (approximately) one, since the sum of the squares of sin(x) and cos(x) is unity for any x. (**SIN** and **COS** always return **FLOAT**s, and each takes its argument in radians. **ATAN** (arctangent) returns its value in radians. Any other trigonometric function can be composed from these three.)

MDL doesn't have a general "to the power" **SUBR**, so let's define one using **LOG** and **EXP** (log base e, and e to a power, respectively; again, they return **FLOAT**s).

```
<DEFINE ** (NUM PWR)
        <EXP <* .PWR <LOG .NUM>>>>$
**
<** 2 2>$
4.0000001
<** 5 3>$
125.00000
<** 25 0.5>$
5.0000001
```

Two **FUNCTION**s which use a single global variable (Since the **GVAL** is used, it cannot be rebound.):

```
<DEFINE START ()
        <SETG GV 0>>$
START
<DEFINE STEP ()
        <SETG GV <+ ,GV 1>>>$
STEP
<START>$
0
<STEP>$
1
<STEP>$
2
<STEP>$
3
```

START and **STEP** take no arguments, so their argument **LIST**s are empty.

An interesting, but pathological, **FUNCTION**:

```
<DEFINE INC (ATM)
        <SET .ATM <+ ..ATM 1>>>$
INC
<SET A 0>$
0
<INC A>$
1
<INC A>$
2
.A$
2
```

INC takes an ATOM as an argument, and **SET**s that **ATOM** to its current **LVAL** plus **1**. Note that inside **INC**, the **ATOM ATM** is **SET** to the **ATOM** which is its argument; thus **..ATM** returns the **LVAL** of the argument. However, there is a problem:

```
<SET ATM 0>$
0
<INC ATM>$

*ERROR*
ARG-WRONG-TYPE
+
LISTENING-AT-LEVEL 2 PROCESS 1
```

The error occurred because **.ATM** was **ATM**, the argument to **INC**, and thus **..ATM** was **ATM** also. We

really want the outermost **.** in **..ATM** to be done in the "world" (**ENVIRONMENT**) which existed __just before__ **INC** was entered -- and this definition of **INC** does both applications of **LVAL** in its own "world".

6.5. Pretty-Printing

In MDL, carriage-returns, linefeeds, tabs, etc., are just separators, like spaces. At least one space is needed between MDL objects, but there is no maximum number.

```
<**         3
                                    4>$
81
```

Using only one space at all times results in code which is effectively unreadable. This is even demonstrable with tiny **FUNCTION**s similar to the ones created in this chapter. For example:

```
<DEFINE ZERO (THETA)
        <- <+ <SQUARE <SIN .THETA>>
              <SQUARE <COS .THETA>>>
           <+ <SQUARE <SIN .THETA>>
              <SQUARE <COS .THETA>>>>>$
ZERO
```

Typing **,ZERO$** to MDL will cause it to return:

```
#FUNCTION ((THETA) <- <+ <SQUARE <SIN .THETA>> <SQUARE <COS
.THETA>>> <+ <SQUARE <SIN .THETA>> <SQUARE <COS .THETA>>>>)
```

Long **FUNCTION**s printed like this would be __very__ difficult to read. MDL has a "pretty-printer" (for full details see *The MDL Programming Environment* [Lebling 80]), called **PPRINT** which prints functions with spacing similar to the examples in this chapter.

```
<PPRINT ZERO>$

<DEFINE ZERO (THETA)
        <- <+ <SQUARE <SIN .THETA>> <SQUARE <COS .THETA>>>
           <+ <SQUARE <SIN .THETA>> <SQUARE <COS .THETA>>>>>
```

The general idea behind MDL pretty printing is: if all the arguments to a *function* fit on one line they are printed on one line, if not, arguments are printed on successive lines indented by the same amount. This allows you to see the level of "nesting" at a glance, and makes it easier to see what is happening.

6.5.1. Editors and Pretty Printing

A good display editor (such as RMODE [Lebling 77] or EMACS [Stallman 79]) will have built-in commands which assist you in formatting your programs in pretty-print style. It is strongly recommended that you get in the habit of using these tools from the beginning. Your code will be more easily understood by others and, more importantly, by you several months after you write it. Bracket balancing also becomes much easier and errors with brackets become quite rare.

6.6. Loading a File

If you have a MDL program in a file, you can "load" it by typing

```
<FLOAD file>$
```

where *file* is the name of the file, in standard operating-system syntax, enclosed in "s (double-quotes). In the Tenex and Tops-20 versions, if the file name extension is **.MUD**, the extension can be omitted. For instance, to load the file **ZERO.MUD** you could type one of the following:

```
<FLOAD "ZERO">$    <FLOAD "ZERO.MUD">$
```

Once you type **$**, MDL will process the text in the file (including **FLOAD**s of yet other files) exactly as if you had typed it on a terminal and followed it with **$**, except that "values" produced by the computations are not printed. When MDL is finished processing the file, it will print **"DONE"**.

If there is more than one generation of the file **ZERO.MUD**, MDL will load the highest one unless a generation number is specifically included in the argument to **FLOAD** (e.g. **<FLOAD "ZERO.MUD.69105">**).

When MDL starts running, it will **FLOAD** the file **"MUDDLE.INIT"** (Tenex and Tops-20 versions), if it exists. This allows you to have your working file or any other files you wish loaded into your MDL when you begin a session. It also allows you to "customize" your MDL by setting certain flags, redefining **FUNCTION**s, etc.

7. MDL TYPEs

In Chapter 2, we provided an introduction to the MDL **TYPE** system. This chapter will expand on that introduction and explain the creation of user-definable MDL **TYPE**s.

7.1. TYPEs and PRIMTYPEs

In Chapter 2 it was stated that every MDL object has a **TYPE**. The **SUBR TYPE**, given a MDL object, returns an **ATOM** which is the name of the object's **TYPE**.

```
<TYPE 12>$
FIX
<TYPE (1 2 3)>$
LIST
```

In MDL, each **TYPE** can be thought of as a member of a smaller number of more 'primitive' **TYPE**s. In MDL, these 'primitive' **TYPE**s are known as **PRIMTYPE**s. Just as every MDL object has a **TYPE**, so every MDL object has a **PRIMTYPE**. A **SUBR** called **PRIMTYPE**, given a MDL object, returns an **ATOM** which is the name of the object's **PRIMTYPE**.

We have already seen examples of a number of MDL **PRIMTYPE**s without ever mentioning the notion of **PRIMTYPE**. Here are the most important **PRIMTYPE**s in MDL.

- **WORD** - the **PRIMTYPE** of all **FIX**es, **FLOAT**s, and **CHARACTER**s. Any MDL object which can be thought of as a number will be of **PRIMTYPE WORD** (**CHARACTER**s are internally stored as their ASCII values).

- **ATOM** - the **PRIMTYPE** of **ATOM**s.

- **LIST** - the **PRIMTYPE** of **LIST**s, **FORM**s, and **FALSE**s.

- **VECTOR** - the **PRIMTYPE** of **VECTOR**s.

- **STRING** - the **PRIMTYPE** of **STRING**s.

7.2. Introduction to MDL Structures

As we saw in Chapter 2, MDL objects may be either *structured* or not. A *Structure* can be thought of as an ordered series of MDL objects. MDL has a number of different 'classes' of *structures*, each with different properties. These 'classes' are the *structured* **PRIMTYPE**s: **LIST**, **VECTOR**, and **STRING**. In Chapter 2, it was also noted that matching brackets are used to represent these *structured* objects. Each of the *structured* **PRIMTYPE**s has its own unique bracket type by which it can be identified. The brackets used for the *structured* **PRIMTYPE**s are as follows:

- **LIST** - matching parentheses
- **VECTOR** - matching square brackets
- **STRING** - paired double quotes

```
<SET A (1 2 3)>$
(1 2 3)
<TYPE .A>$
LIST
<TYPE <TYPE .A>>$
ATOM
<PRIMTYPE .A>$
LIST
<SET B <+ 1 2>>$
3                          ;"Oops!"
<SET B '<+ 1 2>>$
<+ 1 2>                    ;"That's better!"
<TYPE .B>$
FORM
<PRIMTYPE .B>$
LIST
```

In the example, notice that the **FORM** **<+ 1 2>** will get evaluated in the call to **SET**. In order to **SET B** to the **FORM** instead of the result of its evaluation, a single-quote is placed before the **FORM**. The single-quote tells MDL *not* to evaluate the following object.

7.3. The TYPE? Predicate

The **SUBR** **TYPE?** can be used to check the **TYPE** of a given object against a particular set of **TYPE** names. **TYPE?** takes a MDL object and any number of **ATOM**s, which must each be the name of a MDL **TYPE**. If the object is not one of those **TYPE** names given, **TYPE?** returns **#FALSE ()**. Otherwise, it returns the **TYPE** of the object.

```
<TYPE? 10 ATOM VECTOR>$
#FALSE ()
<TYPE? 10 FIX FLOAT ATOM>$
FIX
```

7.4. Printing of MDL Objects

In general, the printing of a MDL object is dependent on the **PRIMTYPE** of that object. MDL objects will usually be printed as follows:

*#type-name object-as-if-***PRIMTYPE**

Usually, if the **TYPE** of the object and the **PRIMTYPE** of the object *are not* the same, the number-sign and *type-name are* printed. There are a few exceptions to this: the **TYPE**s **FIX**, **FLOAT**, **CHARACTER**, and **FORM** all print in a more simplified manner because of their common use.

We have already seen an example of this 'number-sign notation' with the **TYPE FALSE**. You may have noticed that it prints as a number-sign, the **ATOM FALSE**, and an 'empty' **LIST**. The meaning of this is that **FALSE**s are of **PRIMTYPE LIST**: the **#FALSE** must be used in both input and output to distinguish the object from objects of **TYPE LIST**. In general, you can tell the **PRIMTYPE** of an unknown type in 'number-sign notation' by looking at the part *after* the *type-name*. If it has square-brackets, it's a **PRIMTYPE VECTOR**. Parentheses, it's a **PRIMTYPE LIST**. Etc.

```
<PRIMTYPE #TABLE [1 2 3]>$
VECTOR
<PRIMTYPE #TEXT "ABCDE">$
STRING
<PRIMTYPE #NUMBER 10>
WORD
```

Note that the **TYPE**s **TABLE**, **TEXT**, and **NUMBER** are not defined in MDL; a user might have created them, however (see later), and their **PRIMTYPE**s are obvious from the part *after* the *type-name*.

7.5. Significance of PRIMTYPEs / CHTYPE

The notion of **PRIMTYPE** is very important. The **PRIMTYPE** of an object tells MDL what the object looks like *internally to MDL*. As far as MDL is concerned, any two objects of the same **PRIMTYPE** are more or less interchangeable (e.g. most **SUBR**s which can be used on **LIST**s can also be used on **FALSE**s.)

This notion of interchangeability is a very powerful one. In fact, MDL allows you to arbitrarily change the TYPE of virtually *any* MDL object to another TYPE, as long as objects of that other TYPE have the same PRIMTYPE as the original. The SUBR which 'changes' TYPEs is called CHTYPE (pronounced 'chitype'). It takes a MDL object and the name of a TYPE (ATOM), and returns the MDL object 'changed' to that TYPE.

```
<CHTYPE (+ 1 2) FORM>$
<+ 1 2>
<CHTYPE (A B C) FALSE>$
#FALSE (A B C)
<CHTYPE 2.5 LIST>$

*ERROR*
STORAGE-TYPES-DIFFER
CHTYPE
LISTENING-AT-LEVEL 2 PROCESS 1
```

Often one would like to know what the PRIMTYPE of an object of a certain TYPE would be. This can be found out by using the SUBR TYPEPRIM: given a name of a TYPE, it returns the name of the PRIMTYPE of objects of that TYPE.

```
<TYPEPRIM FALSE>$
LIST
<TYPEPRIM FLOAT>$
WORD
```

To restate the conditions for a successful CHTYPE in terms of TYPEPRIM: the PRIMTYPE of the first argument must be the same as the TYPEPRIM of the second. Isn't that much clearer?

7.6. Creating new TYPEs

Given the interchangeability among objects with the same PRIMTYPE, it should not be surprising that MDL will allow you to create any arbitrary new TYPE, so long as you define it to have a known MDL PRIMTYPE. The SUBR which creates new TYPEs is, not surprisingly, NEWTYPE. NEWTYPE takes an ATOM (the name for your new TYPE) and the name of the TYPEPRIM for that new TYPE (also an ATOM). It returns its first argument. NEWTYPEs will defaultly print out (and can be read back) in 'number-sign notation'.

```
<NEWTYPE TABLE VECTOR>$
TABLE
<SET X #TABLE [JOE 1 JOHN 2]>$
#TABLE [JOE 1 JOHN 2]
<CHTYPE .X VECTOR>$
[JOE 1 JOHN 2]
```

There are only two ways to create an object of a user-defined **TYPE**: type the object in directly (as was done in the previous example) or to use the **SUBR CHTYPE** explicitly.

```
<CHTYPE [JIM 2 JANE 4] TABLE>$
#TABLE [JIM 2 JANE 4]
```

8. MDL Structures

As we saw in Chapter 2, MDL objects may be either *structured* or not. It was stated that structures can be thought of as ordered series of MDL objects and that different classes of structures existed. In this chapter we will describe the common structures used in MDL.

8.1. Equality

It is necessary here to mention the notion of equality. In MDL, there are two types of equal: double-equal and single-equal. The **SUBR**s which represent these concepts are **==?** and **=?**, respectively. Simply stated, two MDL objects which are the same thing are double-equal. Two objects which *look* the same, i.e. are printed the same way, are single-equal. This confusing distinction is unimportant for objects which aren't *structured*. Two *non-structured* objects which print the same *are* the same. For example, there is one and only one MDL object representing the **FIX 19**. However, one can easily build two *structures* at two different times which *look* the same, but which are not the same. This will be explained below in the discussion about **LIST**s. As an example of the use of **=?**, assume that you have written a program which takes some input from the user and wants to see if he typed the word **FOO**. Let's assume an input routine called **INPUT** which returns a **STRING**.

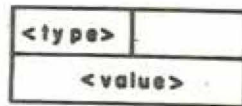
```
<SET STR <INPUT>>$
"FOO"
<==? .STR "FOO">$
#FALSE ()
<=? .STR "FOO">$
T
```

This is because the two **STRING**s were not identical; they look the same, however, and therefore are **=?**. Figure 8-1 purports to demonstrate the distinction between types of equality.

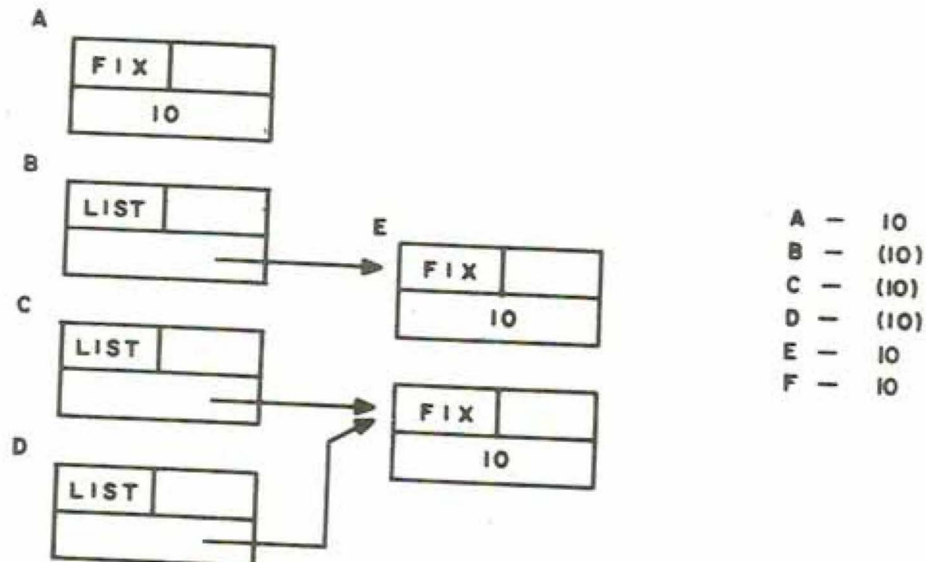
8.2. PRIMTYPE LIST

MDL objects of **PRIMTYPE LIST** may be thought of as an ordered series of MDL objects, whose connective link is a 'pointer'. This means that in order to find the Nth element of a **LIST** one must look at each of the previous N-1 elements. This is shown in Figure 8-2. This becomes rather tedious

The representation of a MDL object is:



where <type> is the **TYPE** of the object and <value> is a pointer for *structured types*, or a number.



Definition:	Two MDL objects are **==?** if and only if the <type> and <value> parts are the same. They are **=?** if they <u>look</u> the same.
Question:	Are any of **B**, **C**, or **D** **==?** to each other?
Answer:	**C** and **D**. They have the same **TYPE** and point to the <u>same</u> structure. **B** and **C** are **=?**, as are **B** and **D** - they <u>look</u> <u>alike</u>, but are not identical.
Question:	Are any of **A**, **E**, or **F** **==?** to each other?
Answer:	They are all **==?** to each other.

Figure 8-1: The MDL notion of equality is demonstrated in this figure, which shows the distinction between single-equal **=?** and double-equal **==?**.

when one is interested in finding the 245th element of a LIST. You can see that large LISTs have the property of being rather inefficient to 'random-access'. On the other hand, LISTs can easily be modified (adding elements, removing elements, etc) simply by changing the linking 'pointers'. In Figure 8-3 you can see pictorially how an element of a LIST might be removed. Notice that the removed element still 'exists', but that the LIST is no longer 'pointing' at it.

Two SUBRs which should be mentioned here are LENGTH and EMPTY?: the first, given a LIST, returns the number of elements in that LIST (as a FIX), and the second, given a LIST, returns the ATOM T or #FALSE (), i.e. whether the LIST had *no* elements.

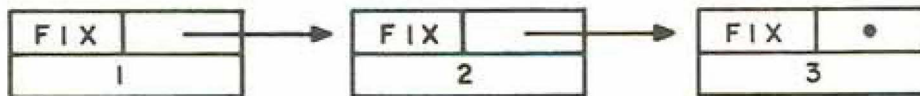
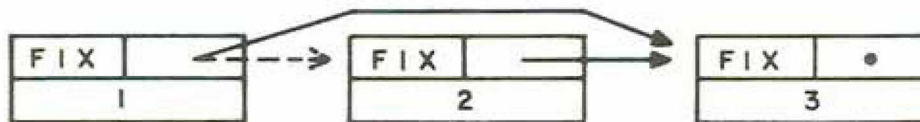


Figure 8-2: The LIST (1 2 3)



This LIST is now (1 3).

Figure 8-3: Removing a LIST element by moving only one pointer

8.2.1. Creating LISTs

Creating a LIST is very simple. Simply type in the printed representation of it, which (as described before) is a series of MDL objects surrounded by parentheses.

```
<SET A (1 TWO 3.0)>$
(1 TWO 3.0)
<SET B (A .A C)>$
(A (1 TWO 3.0) C)
```

Using this method, every MDL object placed between the matching parentheses is EVALed. Thus, the **LVAL** of **A** was placed in the **LIST**.

```
<SET B (X Y Z)>$
(X Y Z)
<LENGTH .B>$
3
<EMPTY? .B>$
#FALSE ()
<DEFINE EMPTY? (LIST) <==? <LENGTH .LIST> 0>>$
EMPTY?
```

The end of the previous example gives a definition of **EMPTY?** in MDL, given only the **SUBRs LENGTH** and **==?**.

A second way to create a **LIST** is with the **SUBR LIST**. This **SUBR** takes any number of arguments, which are **EVALed**, and makes a list with the evaluated arguments as elements. The effect is the same as in the first method.

```
<SET A <LIST 1 TWO 3.0>>$
(1 TWO 3.0)
```

In both methods, a *new* **LIST** is created.

```
<SET A (1 2 3)>$
(1 2 3)
<SET B (1 2 3)>$
(1 2 3)
<==? .A .B>$
#FALSE ()
<=? .A .B>$
T
```

The two lists **A** and **B** are not double-equal because the construction of **LISTs** is guaranteed to generate a *new* **LIST**. They are single-equal by the definition of single-equal.

8.2.2. EVALing LISTs

LISTs, when **EVAL**ed, make a *new* copy of the **LIST** with all of the elements re-**EVAL**ed.

```
<SET A (1 2 3)>$
(1 2 3)
<==? <EVAL .A> .A>$
#FALSE ()
```

8.2.3. Manipulating LISTs

In order to discuss **LIST**s more fully, we need to know a few ways to manipulate them. We will introduce two **SUBR**s here, **NTH** (pronounced 'enth') and **REST**. The **SUBR NTH**, given a **LIST** and a **FIX**, will return the **FIX**th element of the **LIST**. **REST**, given a **LIST** and a **FIX**, will return the **LIST**, with the first **FIX** elements at the beginning removed. The second argument to both **NTH** and **REST** has a default value of 1. Some examples:

```
<SET L (A B C D)>$
(A B C D)
<NTH .L 3>$
C
<NTH .L 2>$
B
<SET LL <REST .L 2>>$
(C D)
<REST .L 4>$
()
.L$
(A B C D)
```

Notice that **REST** has no side-effects. In other words, it simply returns a pointer farther down the 'chain' of elements in the **LIST** without changing anything. This is illustrated in Figure 8-4. Another important operation on **LIST**s is called **PUT**. As its name suggests, **PUT** puts an element into a **LIST**. Given a **LIST**, an element number (**FIX**, as in **NTH**), and an arbitrary object, **PUT** makes the **FIX**th element of **LIST** become that object, and returns the **LIST**. Let's continue from the example given in Figure 8-4 with **L** and **LL** already defined.

```
<PUT .LL 1 HAHA>$
(HAHA D)
.L$
(A B HAHA D)
```

What happened here is shown in Figure 8-5. Since **LL** was a 'subset' of **L**, any change in **LL** was reflected in **L** (the opposite would also be true, i.e. a **PUT** into the third or fourth elements of **L** would

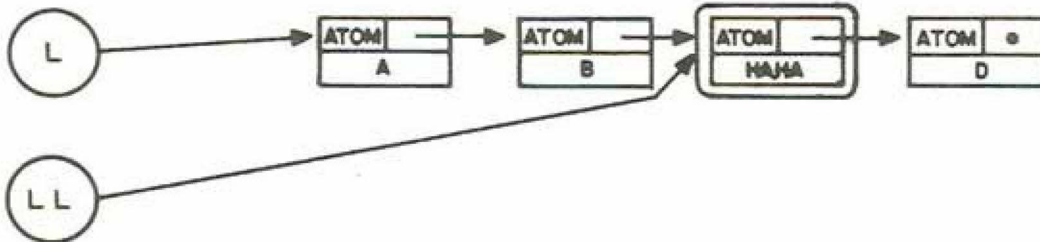
```
<SET L (A B C D)>$
(A B C D)
```

```
<SET LL <REST .L 2>>$
(C D)
```

Notice that the **LIST** `(C D)` is a subset of the **LIST** `(A B C D)` because of the way **REST** works.

Figure 8-4: REST of a LIST

be reflected in **LL**.)



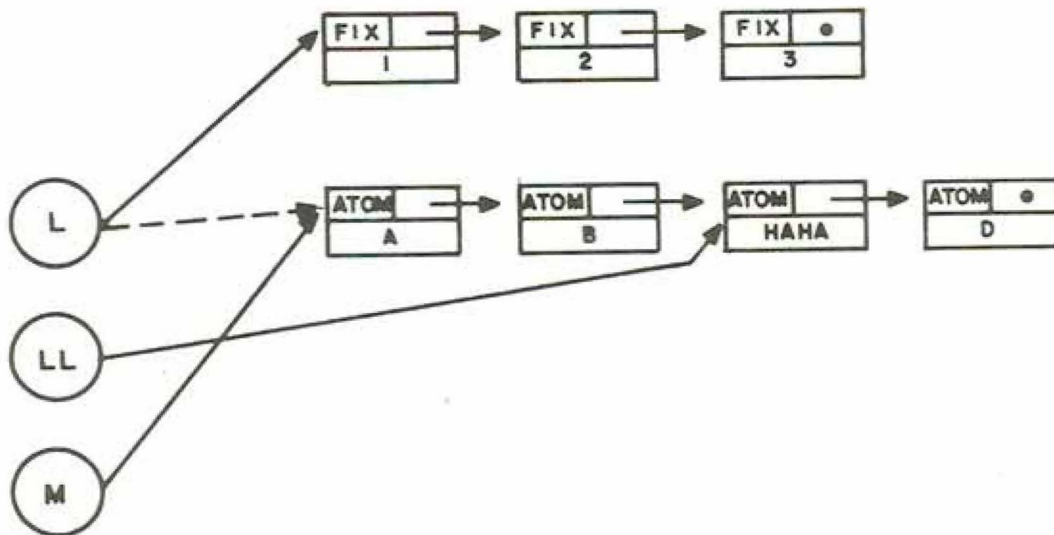
The only effect is that the contents of the third element was changed, from [ATOM C] to [ATOM HAHA]. No pointers have moved.

Figure 8-5: PUTs into LISTs

Let's continue:

```
<SET M .L>$
(A B HAHA D)
<SET L (1 2 3)>$
(1 2 3)
.M$
(A B HAHA D)
.LL$
(HAHA D)
<==? <REST .M 2> .LL>$
T
```

If you understand this, good. Otherwise, pay close attention to Figure 8-6, in which this example is diagrammed. It is of crucial importance that the distinction be learned between a *structure* and a *pointer* to a *structure*. Changing a *structure* (e.g. with **PUT**) will be reflected in any object which points to it. Changing the *pointer* to a *structure* doesn't affect any other pointers. If you don't understand this distinction, you will probably become more and more lost. Ask someone for help.



<SET M .L> made **M** point to where **L** pointed at that time.
<SET L (1 2 3)> merely pointed **L** somewhere else.
The values of **M** and **LL** are not affected, then, by reSETting **L**.

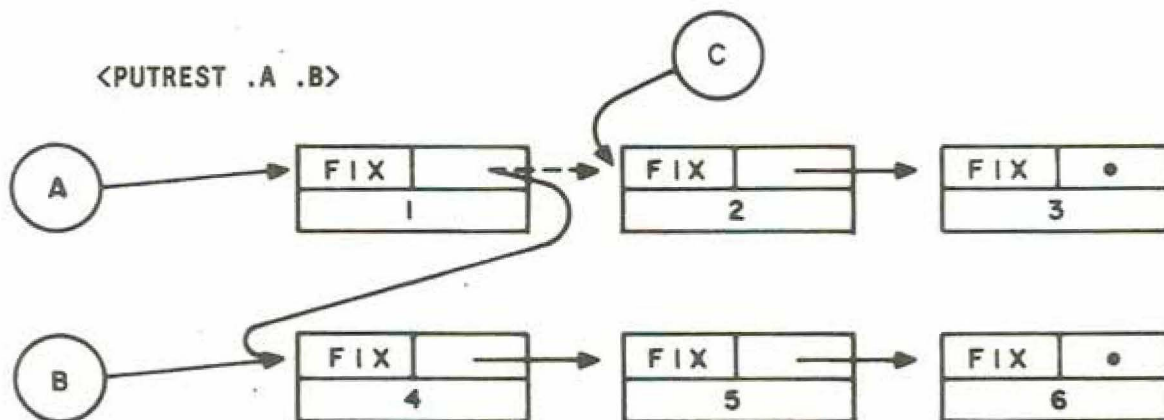
Figure 8-6: Pointers *vs.* Structures

Now things get a little more complex. However, if you understood the previous examples, this should be no different. Earlier, we talked about 'moving' pointers to effect removal of objects from a **LIST**.

This movement can be accomplished in MDL using the **SUBR PUTREST**. **PUTREST** (equivalent to Lisp's replacd) is probably the most confusing **SUBR** to beginners, and even to accomplished MDLers. Its effect is very simple: given two **LISTs**, say A and B, it causes the **REST** of A to become B, and then returns A. This probably sounds very obscure. Before total confusion sets in, take a look at the example and then at Figure 8-7.

```
<SET A (1 2 3)>$
(1 2 3)
<SET B (4 5 6)>$
<PUTREST .A .B>$
(1 4 5 6)
.B$
(4 5 6)
```

All that has happened is that one pointer has been moved: the one connecting the first element of **A** to its succeeding element has been changed to a pointer to **B**. That's all. Notice that any object which points to the same place that **A** points has been changed. However, also notice that any object which points to the **REST** of **A** has *not* been changed.



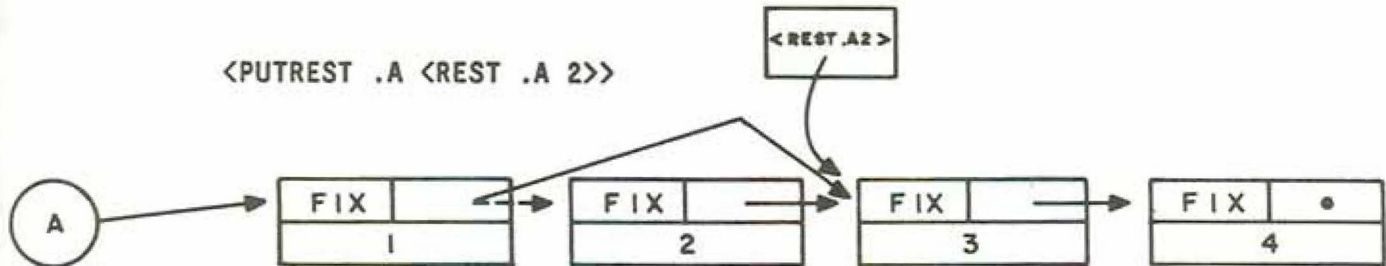
Only one pointer has been moved. **A** is changed, but **B** is not. Notice that a hypothetical **C**, previously **SET** to **REST** of **A**, is also not changed.

Figure 8-7: PUTREST

Using **PUTREST**, it is easy to remove elements from a **LIST**.

```
<SET A (1 2 3 4)>$
(1 2 3 4)
<PUTREST .A <REST .A 2>>$
(1 3 4)
```

What we have done is to make the first element of **A** (**1** in the example) point to the value of **A RESTed** twice. This is demonstrated in Figure 8-8.



The **REST** of **A** has become **A RESTed** twice. The effect is to remove the **FIX** **2** from the **LIST**.

Figure 8-8: Removing an element from a **LIST** using **PUTREST**

Notice that you can *not* use this method to remove the first element of a **LIST**, since **PUTREST** only changes the pointer which connects the first element to the second element. However, one can always use **REST** for this purpose, but be careful:

```
<SET A (1 2 3 4)$
(1 2 3 4)
<REST .A>$
(2 3 4)
.A$
(1 2 3 4)
```

As we noted earlier, **REST** has no side-effects, unlike **PUTREST**, which does. The right thing to do is

```
<SET A <REST .A>>$
(2 3 4)
.A$
(2 3 4)
```

One can cause a **LIST** to terminate at any point by giving **PUTREST** a second argument of an empty

LIST.

```
<SET A (1 2 3 4)>$
(1 2 3 4)
<PUTREST .A ()>$
(1)
```

As advertised, the **REST** of **A** has been made the second argument to **PUTREST**, i.e. the empty **LIST**.

To combine **LIST**s, one can use **PUTREST** also. Try to think of how you would combine the **LIST**s in the following example. Think pointers.

```
<SET A (1 2 3)>$
(1 2 3)
<SET B (4 5 6)>$
(4 5 6)
```

The idea is to make the third element of **A** (the **FIX 3**) to point to the **LIST B**. In terms of **PUTREST**, we want **B** to become the **REST** of which **LIST**? The answer is

```
<PUTREST <REST .A 2> .B>$
(3 4 5 6)
.A$
(1 2 3 4 5 6)
```

PUTREST returned its first argument, which was **A REST**ed twice. **A**, however, was changed. Refer to figure 8-9 if confused.

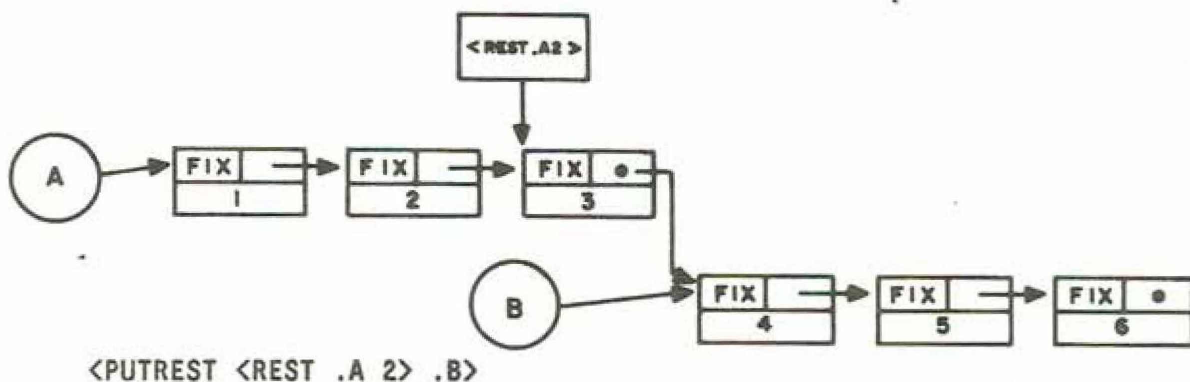


Figure 8-9: Splicing **LIST**s together using **PUTREST**

With your new-found expertise in **PUTREST**ing, you should take a moment and think about how you would build a **LIST** backwards. For example, you wish to append the **FIX 7** to the **LIST** from the last example. What is the correct MDL expression? Hint: You have to create a **LIST** with the **FIX 7** in it.

```
<PUTREST <REST .A 5> (7)>$
(6 7)
.A$
(1 2 3 4 5 6 7)
```

Here are some problems to think about:

```
<SET L (1 2 3 4)>$
(1 2 3 4)
<SET LL <REST .L 1>>$
(2 3 4)
<SET LLL <REST .L 2>>$
(3 4)
<PUTREST <REST .L> <REST .L 3>>$
(2 4)
```

What are the LVALs of L, LL, and LLL now?

```
<SET WALTZ (2 3 1)>$
(2 3 1)
<PUTREST <REST .WALTZ 2> .WALTZ>$
```

If you try this, be ready to type ↑S. What has happened? What is the **LENGTH** of **L** now? Why shouldn't you try to find out? Why is this a waltz?

```
<SET ONES (1 2 3)>$
(1 2 3)
<PUTREST .ONES .ONES>$
```

What about this?

The last two examples demonstrate an important notion, that of circularity. There is absolutely no restriction on the creation of circular and self-referencing structures. However, you should be sure you know what you're doing. For example, finding the **LENGTH** of **ONES** or of **WALTZ** in the previous examples is quite time-consuming. The **SUBR** called **LENGTH?** can be of use here. Given a **LIST** and a **FIX**, **LENGTH?** will return the **LENGTH** of the **LIST** if it is less than or equal to **FIX**. Otherwise, it will return **#FALSE ()**. This is useful if you suspect a **LIST** is self-referencing or to check on whether a **LIST** is at least a certain length. For example, prior to trying to get the 12th element of a **LIST** of uncertain size, one might check that

```
<LENGTH? .LIST 11>
```

returned #FALSE (), i.e. there are at least 12 elements in the LIST.

8.2.4. FIXes First in FORMs

If the first element of a FORM is a FIX *or* an ATOM whose GVAL is a FIX, this is considered to be a shorthand call to NTH or PUT, depending on whether it is given one or two arguments, respectively. Thus, the following two are EVALuated identically.

```
<11 .FOO>
<NTH .FOO 11>
```

So are these:

```
<11 .FOO .BAR>
<PUT .FOO 11 .BAR>
```

Here is an example of an ATOM being used first in a FORM with the same effect:

```
<SET L (FOO BAR BLETCH)>$
(FOO BAR BLETCH)
<SETG FIRST 1>$
1
<SETG SECOND 2>$
2
<FIRST .L>$
FOO
<SECOND .L>$
BAR
<FIRST .L FROB>$
(FROB BAR BLETCH)
```

8.2.5. FORMs

As described earlier, FORMs are used to apply *functions* to arguments, and are printed with angle brackets. However, FORMs are simply another variety of PRIMTYPE LIST and all of the operations which can be done on LISTs can be done on them. Since they are evaluated in a special way, creating a FORM by inputting elements between angle brackets will require a single-quote. This is not true if you are using the SUBR FORM.

```
<SET A <+ 1 2 3>>$
6
<SET A '<+ 1 2 3>>$
<+ 1 2 3>
<SET A <FORM + 1 2 3>>$
<+ 1 2 3>
```

A special note should be made of the empty **FORM**: it evaluates to an empty **FALSE**. This is simply a shorthand notation.

```
<>$
#FALSE ()
```

8.2.6. FALSEs

Previously, you have seen examples of MDL objects of **TYPE FALSE**. All of them thus far have been **EMPTY?**, although this is not always the case. MDL objects of **TYPE FALSE** are **PRIMTYPE LIST**s and, as has been stated before, can be used in the same ways as any other **PRIMTYPE LIST**. In particular, one can create **FALSE**s with any number of arbitrary elements. One use of this might be to distinguish between two types of failures in a *function*. Thus, the **FALSE** can have two types of meaning: its **TYPE** (which is **FALSE**) and its contents. One might simply want to detect failure by checking the **TYPE**, but one might additionally want to detect failure and also have other information about the failure available.

```
<SET VAL <OPEN "READ" "FOO.BAR">>$
#FALSE ("File not found" "FOO.BAR" 69105)
<1 .VAL>$
"File not found"
```

In this example, the **SUBR OPEN** was called in an attempt to open a file called **FOO.BAR**. The **OPEN** failed, and returned a **FALSE** which contained three pieces of information: the reason (a **STRING**), the file name (a **STRING**), and an internal error code (a **FIX**). One might have written a **FUNCTION** using **OPEN** which only cares if **OPEN** returns a **FALSE** or not. On the other hand, one might want to print out the reason for the failure to the **FUNCTION**'s user. This would have been impossible had **FALSE**s not been able to carry additional information. As you will find when doing your own programming, this is a significant feature of MDL.

8.2.7. SEGMENTs

A **SEGMENT** is a **PRIMTYPE LIST**, which is handled very specially by MDL. **SEGMENT**s print as an exclamation point followed by a **FORM**. When **EVAL**ed inside an expression, its meaning is as follows: pretend that instead of using this **SEGMENT**, use instead all of the elements you get from **EVAL**ing the **FORM**. There is an important implication here: that the **FORM**, when **EVAL**ed, returns a *structure*. An

error will occur if this is not the case. Here are some examples:

```
<SET A (1 2 3)>$
(1 2 3)
<SET B (!.A 4 5 6)>$
(1 2 3 4 5 6)   ;"This is a new list, not shared with A"
<SET C (!.A !.B 7)>$
(1 2 3 1 2 3 4 5 6 7)   ;"No sharing here, either"

<+ !.A>$
6

<SET L (BAR 10)>$
(BAR 10)
<SET !.L>$
10
.BAR$
10
```

This last example is quite pathological: note, however, that it is perfectly legitimate. The **FORM** was read by MDL as having two elements, the **ATOM SET** and a **SEGMENT**. When the **FORM** was **EVAL**ed, the **SEGMENT** acted as if it was really all of the elements of **.L**, i.e. the **ATOM BAR** and the **FIX 10**. This is simply the case of **SET**ting **BAR** to **10**.

One last very important note: There is one way to add elements to the beginning of a **LIST** without copying. This is the case in which a **SEGMENT** is the last element of a **LIST** and the **SEGMENT**'s **FORM** **EVAL**s to a **LIST**. In this case only, there is no copying and the structures will share. This is similar to CONS in LISP.

```
<SET L (FOO BAR BLETCH)>$
(FOO BAR BLETCH)
<SET LL (1 2 3 !.L)>$   ;"The last element is a SEGMENT which
                          evaluates to a LIST"
(1 2 3 FOO BAR BLETCH)
<PUT .LL 4 SHARED>$
(1 2 3 SHARED BAR BLETCH)
.L$
(SHARED BAR BLETCH)
```

LISTs are the most appropriate *structure* to use when elements are going to be added or removed. The special use of **SEGMENT**s shown in the last example is the best way of adding elements to the front of a **LIST**. However, the resulting **LIST** will be 'backward', in that the most recently added element will be at the 'front' rather than at the 'back' of the **LIST**. Later on, we will demonstrate the correct way to add elements to the end of a **LIST**. As an exercise, see if you can figure out a method for doing so using **PUTREST**.

Remember that the previous use of **SEGMENT**s *is an exception*: in the case of objects of **PRIMTYPE VECTOR** and **STRING** (next sections) the use of a **SEGMENT** will cause copying of the elements from the *structure* which is the **EVAL**uation of the **FORM**.

8.3. PRIMTYPE VECTOR

A MDL object of **PRIMTYPE VECTOR** can be thought of as a linear array of MDL objects. In a **VECTOR**, it is trivial to access the Nth element, as this simply requires finding the correct offset into the structure (see Figure 8-10). Similarly, it is trivial to replace the Nth element with something else. On the other hand, there is *no* way to add elements to a **VECTOR** without creating a new one, and removing elements can be simulated, although it is rather difficult. If your eyes skipped back to the section on **PRIMTYPE LIST**, you might notice that these properties of the two **PRIMTYPE**s are reversed. This, then, is the rationale for having different structure 'classes'. The programmer is free to choose the structure 'class' (i.e. **PRIMTYPE**) he wants, based on the way in which it is to be used in a program. For example, a structure which is always of known length should probably be a **VECTOR**, while one which must undergo changes in size should probably be a **LIST**.

Schematic representation of a **VECTOR**

```
[1 2 3 4]
```

FIX	
1	
FIX	
2	
FIX	
3	
FIX	
4	

Figure 8-10: The **VECTOR** `[1 2 3 4]`

8.3.1. Creating VECTORs

Creating a **VECTOR** is completely analogous to creating a **LIST**. There are two options: you can type in the printed representation of a **VECTOR**, or you can use the **SUBR VECTOR**.

```
<SET A [1 2 3]>$
[1 2 3]
<SET B <VECTOR 1 2 3>>$
[1 2 3]
<==? .A .B>$
#FALSE ()
<=? .A .B>$
T

<SET C [.A !.B]>$
[[1 2 3] 1 2 3]                ;"C does not share with B!"
```

Both of these methods always create a *new* **VECTOR**. Therefore, two objects created in separate calls to **VECTOR** will never be **==?**.

8.3.2. EVALing VECTORs

As with **LIST**s, **EVAL** of a **VECTOR** makes a *new* copy of the **VECTOR**, with all of the elements **EVAL**ed.

```
<SET A [<+ 1 2> '<+ 1 2> ''<+ 1 2>]>$
[3 <+ 1 2> '<+ 1 2>]
<SET A <EVAL .A>>$
[3 3 <+ 1 2>]
<EVAL .A>$
[3 3 3]
```

8.3.3. Manipulating VECTORs

The **SUBR**s **NTH**, **REST**, **PUT**, **LENGTH**, **EMPTY?**, and **LENGTH?** all work on **VECTOR**s just as they do on **LIST**s.

```
<SET A [ONE TWO 3]>$
[ONE TWO 3]
<NTH .A 2>$
TWO

<PUT .A 3 THREE>$
[ONE TWO THREE]
.A$
[ONE TWO THREE]
```

```
<LENGTH .A>$
3
<REST .A 2>$
[THREE]
.A$
[ONE TWO THREE]
```

RESTing VECTORs is shown in Figure 8-11.

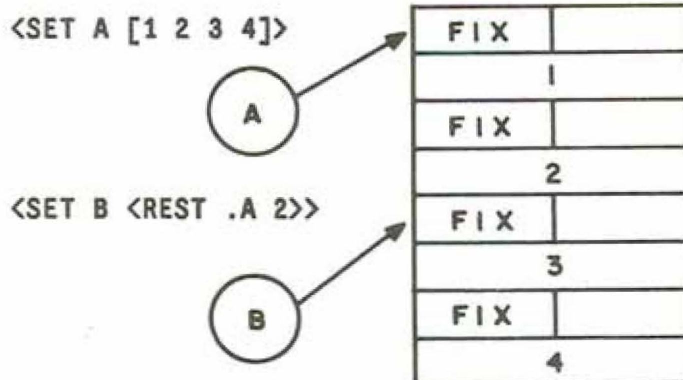


Figure 8-11: REST of a VECTOR

Since VECTORs are not pointer structures, the PUTREST operation will not work on them. However, there are a few operations which are possible with VECTORs which are not possible with LISTs due to their structure. The first of these is the inverse of REST: it is called BACK. Given a VECTOR and a FIX, it tries to replace elements to the front of the VECTOR which were previously RESTed off. Like REST, BACK has no side-effects. It simply returns a pointer to a different location in the VECTOR. An error will occur if you attempt to BACK more elements than have been RESTed. The SUBR TOP, however, given a VECTOR, will BACK as far as is legally possible.

```
<SET A [1 2 3 4]>$
[1 2 3 4]
<SET B <REST .A 2>>$
[3 4]
<PUT .B 1 HAHA>$
[HAHA 4]
.A$
[1 2 HAHA 4]              ;"B is a subset of A"
```

```
<BACK .B>$
[2 HAHA 4]
.B$
[HAHA 4]                    ;"BACK has no side-effects"
<TOP .B>$
[1 2 HAHA 4]
<==? <TOP .B> .A>$
T
```

BACKing of **VECTOR**s is diagrammed in Figure 8-12.

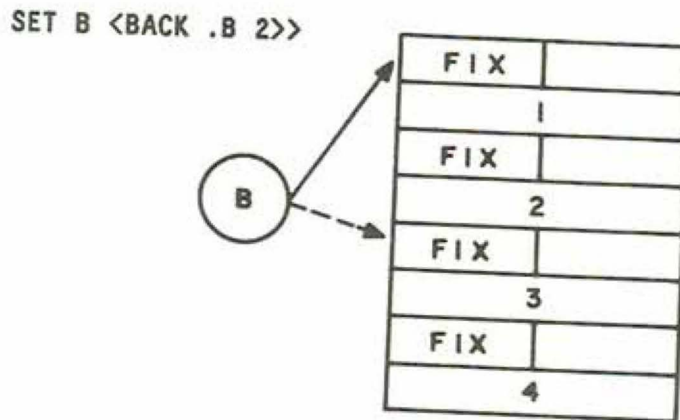


Figure 8-12: **BACK** of a **VECTOR**

8.3.4. UVECTORs

Although infrequently used, MDL has a **PRIMTYPE** called **UVECTOR**, for Uniform **VECTOR**. **UVECTOR**s are identical to **VECTOR**s in most ways *except* that every element of a **UVECTOR** must have the same **TYPE**. **UVECTOR**s have a special input and output form: an exclamation point followed by paired square brackets. Here are some **UVECTOR**s:

```
<SET A ![A B C]>$
![A B C!]   ;"Don't worry about the other ! before the ]"
<1 .A>$
A
<REST .A 2>$
![C!]
```

Analogously to a **VECTOR**, there are two ways to create a **UVECTOR**: type it in, or use the **SUBR UVECTOR**. When typing in a **UVECTOR** be careful that everything you type in is of the same **TYPE**, <u>before</u> EVALuation, as well as after!

```
<SET X 10>$
10
<SET A ![20 .X]>$

*ERROR*
TYPES-DIFFER-IN-UNIFORM-VECTOR
READ
LISTENING-AT-LEVEL 2 PROCESS 1
```

This error occurred because **.X** is a **FORM**, *even though* it **EVAL**s to a **FIX**. To do this properly,

```
<SET A <UVECTOR 20 .X>>$
![20 10]
```

A **SUBR** called **UTYPE** returns the name of the **TYPE** of the elements of a given **UVECTOR**. There are a few **TYPE**s which are illegal elements of **UVECTOR**s: the only one you are likely to come across is **STRING**.

UVECTORs are useful only for efficiency. They take up roughly half the storage of **VECTOR**s. All other considerations are the same as for **VECTOR**s.

8.4. PRIMTYPE STRING

A MDL **STRING** is a sequence of MDL objects of **TYPE CHARACTER**. Objects of **TYPE CHARACTER** are represented by the sequence of characters: exclamation-point, backslash, and the character itself.

```
!\A$
!\A
<TYPE !\A>$
CHARACTER
<ASCII !\A>$
65
<ASCII 65>$
!\A
```

This example has also demonstrated the use of the SUBR ASCII, which given a CHARACTER returns its ASCII value, or given a FIX gives the CHARACTER with that ASCII value. A STRING is represented as a sequence of characters surrounded by double-quotes.

8.4.1. ASCII

ASCII, as used in MDL, is the name of a 7-bit code (i.e. 0000000 - 1111111 base two, corresponding to 0 - 127 in base ten) used to represent keyboard characters (upper and lower case, control characters, punctuation, etc.) as small integers.

8.4.2. Creating STRINGs

STRINGs are created in exactly the same ways as the other structures.

```
<SET A "THIS IS A STRING">$
"THIS IS A STRING"
<SET A <STRING !\T !\H !\I !\S>>$
"THIS"
```

However, STRING is more powerful than this, as its arguments can be either CHARACTERs or other STRINGs.

```
<SET A "THIS IS A">$
"THIS IS A"
<STRING .A " STRING">$
"THIS IS A STRING"
.A$
"THIS IS A"
```

As with VECTORs, all STRINGs created this way are new, i.e. not shared. To put a double-quote inside a string, you must place a backslash before the double-quote. This can be confusing.

```
<SET A "\"\"">$
"\"\""
<LENGTH .A>$
2
```

This STRING has two elements, each a double-quote. To put a backslash into a STRING, the backslash must be preceded by another backslash. This is even more confusing.

```
<SET A "\\\"">$
"\\\""
<LENGTH .A>$
2
```

This is another STRING of two elements: a backslash and a double-quote.

8.4.3. EVALing STRINGs

STRINGs are unlike LISTs and VECTORs in that they evaluate to themselves, rather than to copies of themselves.

8.4.4. Manipulating STRINGs

STRINGs are manipulated exactly as are VECTORs. The CHARACTERs are stored sequentially; thus PUTREST will not work, but BACK and TOP will. The only difference is that the only legal third argument to PUT of a STRING is a CHARACTER. Anything else will cause an error.

It is important to note that STRINGs contain <u>only</u> CHARACTERs. CHARACTERs with special meanings elsewhwere in MDL are simply CHARACTERs in STRINGs.

```
<SET A "1 2 3 <+ 2 2>">$
"1 2 3 <+ 2 2>"
<3 .A>$
2
```

The above example shows that the 'FORM' in the STRING is not EVALed, it is merely 7 CHARACTERs in a STRING. Spaces <u>are</u> CHARACTERs.

There is a major difference between the following two structures:

```
<SET L ("1 2 3" "4 5 6" "7 8 9")>$
("1 2 3" "4 5 6" "7 8 9")
<SET S "(1 2 3) (4 5 6) (7 8 9)">$
"(1 2 3) (4 5 6) (7 8 9)"
<LENGTH .L>$
3
<LENGTH .S>$
23
```

8.5. Building Large Structures

It is occasionally useful to create a large *structure*. It would be very painful to create, say, a **VECTOR** of 100 elements by calling the **SUBR VECTOR** with 100 arguments. MDL provides a way to create *structures* of a specific size, namely the **SUBRs ILIST**, **IVECTOR**, and **ISTRING**. These take two arguments, a **FIX** (the number of elements) and a MDL object. MDL will build a *structure* (**LIST**, **VECTOR**, or **STRING**, respectively) with **FIX** elements, each of which is the result of evaluating the second argument to the **SUBR**. Some examples:

```
<ILIST 10 0>$
(0 0 0 0 0 0 0 0 0 0)
<IVECTOR 5 <>>$
[#FALSE () #FALSE () #FALSE () #FALSE () #FALSE ()]
<ISTRING 30 !\W>$
"WWWWWWWWWWWWWWWWWWWWWWWWWWWWWW"
```

The second argument to **ILIST** and **IVECTOR** can be any MDL object. The second argument to **ISTRING** must be a **CHARACTER**.

8.6. Searching Structures

There are two **SUBRs** which, given an arbitary MDL object and an arbitrary *structure*, will look through the *structure* for that object. They are called **MEMQ** and **MEMBER**. What they do is this: starting with the *structure*, they look at the first element and see if it is *equal* to the object in question. If so, they return the *structure*. Otherwise, they **REST** the structure by **1**, and repeat the procedure. When the *structure* becomes **EMPTY?** (i.e. the object wasn't in the original *structure*), these **SUBRs** return **#FALSE ()**. This means that a successful **MEMQ** or **MEMBER** will return the original *structure* **REST**ed down such that its first element is the object searched for.

You may have noticed a rather ambiguous 'equal' in the last paragraph, and this is the distinction between **MEMQ** and **MEMBER**. In **MEMQ**, the test is double-equal (**==?**), while in **MEMBER** the test is single-equal (**=?**).

```
<SET L (ONE 2 3.0 "FOUR")>$
(ONE 2 3.0 "FOUR")
<MEMQ 2 .L>$
(2 3.0 "FOUR")
<MEMQ "FOUR" .L>$
#FALSE ()
<MEMBER "FOUR" .L>$
("FOUR")
```

8.7. Garbage: Quoting Structures

Often in writing programs, one includes a structure in a **FORM**. For example, one might have a **FORM** that looks like this:

```
<NTH [ADD SUB MUL DIV] .OPCODE>
```

This is very inefficient because the **VECTOR** in the **FORM** is **EVAL**ed every time the **FORM** is **EVAL**ed with the result that a *new* **VECTOR** is created. This creates a lot of 'garbage', where 'garbage' is defined as some piece of structure which is no longer used (i.e. there are no pointers to it). Since your MDL resides in a machine with finite memory, it pays to think about ways of making programs relatively storage-efficient. The proper way of writing the **FORM** in the previous example is

```
<NTH '[ADD SUB MUL DIV] .OPCODE>
```

As mentioned earlier, the quote will ensure that the **VECTOR** will not be **EVAL**ed when the **FORM** is **EVAL**ed. Thus, only one copy of the **VECTOR** will exist. Note that 'quoting' *structures* in this way should be used for **VECTOR**s and **LIST**s. **STRING**s **EVAL** to themselves! You are warned: *NEVER* do a **PUT** into a quoted *structure*!

8.8. Garbage: Building Lists

It is often necessary in a program to build up a **LIST** of elements. Assume that you have a **FUNCTION** which gets elements one at a time and wants to build a list in the order in which they were received. Assume a **LIST** L and an element to be added, say **.OBJ**. One way of doing this is as follows:

```
<SET L (!.L .OBJ)>
```

This is not good practice, as the **LIST** created is a *copy* of the old one with an element added at the

end. Assuming that nothing else but the value of **L** points to the **LIST .L**, the *old* **LIST .L** will become garbage. If you assume adding 100 elements using this method, it becomes clear that thousands of **LIST** elements are needlessly becoming garbage. Equally as bad would be to use a **VECTOR**. The best way of doing this is using **PUTREST**. Follow this example:

```
<SET L (1 2 3)>$
(1 2 3)
<PUTREST <REST .L 2> (4)>$
(3 4)
.L$
(1 2 3 4)
```

Notice that in the general case, one can add an element to the end of a **LIST** by saying:

```
<PUTREST <REST .LIST <- <LENGTH .LIST> 1>> (.ELEMENT)>
```

This is good programming except that **LENGTH** and **REST** get called, both of which are quite slow for long **LIST**s. Remember that **LENGTH** must follow all of the pointers to the end to count up the elements. Here's another way of doing this:

```
<SETG L (T)>$
(T)
<SETG LL ,L>$
(T)
<DEFINE ADD-TO-END (ELEMENT)
        <SETG LL <REST <PUTREST .LL (.ELEMENT)> 1>>>$
ADD-TO-END
<ADD-TO-END 100>$
(100)
<ADD-TO-END 200>$
(200)
.L$
(T 100 200)
```

Notice that both **L** and **LL** needed to have at least one element at the beginning so that **PUTREST** would work. Remember that the first argument to **PUTREST** cannot be **EMPTY?**, by definition. The effect of the program **ADD-TO-END** is to append the element to a **LIST (LL)** which is **REST**ed each time. This saves having to perform long **LENGTH** and **REST** operations. Since **LL** is a sub-list of **L**, **L** is being changed with every **PUTREST**. Thus, **L** is the complete **LIST**, and **LL** is always **L REST**ed down to its last element. You should remember, of course, that the initial **T** in the **LIST** should be removed at a later time....

8.9. Structured NEWTYPEs

In the previous chapter, we saw that MDL is a type-extensible language in that the programmer can create his own TYPEs. Typically, an object of a NEWTYPE will be a structure that is a model of some real world entity with the elements of the structure models of parts or aspects of that entity. The creator of the NEWTYPE will usually provide *functions* for manipulating the NEWTYPE objects in all of the ways which are considered meaningful for the intended uses of that NEWTYPE. This means that other users of the NEWTYPE can use these creator-defined manipulation routines and *never need to know the internal structure of the* NEWTYPE. This provides both modularity of programming and data abstraction.

For example, suppose you wanted to deal with airline schedules. If you were to construct a set of programs that define and manipulate a NEWTYPE called FLIGHT, then you could make that set into a standard package of programs and call on it to handle all information pertaining to scheduled airline flights. Since all FLIGHTs would have the same quantity of information (more or less) and you would want quick access to individual elements, you would not want the TYPEPRIM to be LIST. Since the elements would be of various TYPEs, you would not want the TYPEPRIM to be UVECTOR. The natural choice would be a TYPEPRIM of VECTOR.

Now, the individual elements of a FLIGHT would, no doubt, have TYPEs and meanings that don't change. The elements of a FLIGHT might be airline code, flight number, originating-airport code, list of intermediate stops, destination-airport code, type of aircraft, days of operation, etc. Each and every FLIGHT would have the airline code for its first element (say), the flight number for its second, and so on. It is natural to invent names (ATOMs) for these elements and always refer to the elements by name. For example, you could <SETG AIRLINE 1>. Then, if the local value of F were a FLIGHT, <AIRLINE .F> would return the airline code, and <AIRLINE .F AA> would set the airline code to AA. Once that is done, you can forget about which element comes first; all you need to know are the names of the offsets.

The next step is to notice that, outside the package of FLIGHT functions, no one needs to know whether AIRLINE is just an offset or in fact a function of some kind. For example, the scheduled duration of a flight might not be explicitly stored in a FLIGHT, just the scheduled times of departure and arrival. But, if the package had the proper DURATION function for calculating the duration, then the call <DURATION .F> could return the duration, no matter how it is found. In this way the internal details of the package are conveniently hidden from view and abstracted away.

8.10. Summary of MDL Structures

A few points should be obvious from the previous discussions of the various *structured* PRIMTYPEs:

1. All *structures* can be created in the same two ways: Either type in the printed representation, or use the SUBR whose name is the name of the PRIMTYPE.

2. When LISTs and VECTORs are EVALed, a *new copy* of the *structure* is made, whose

elements are EVAL of the elements of the original. STRINGs EVALuate to themselves.

3. SEGMENTs are a very important and powerful feature of MDL. If you don't undersand their uses, re-read the appropriate section.

4. The SUBRs NTH, REST, PUT, LENGTH, LENGTH?, EMPTY?, MEMBER, and MEMQ work on *all structures*.

5. The SUBRs BACK and TOP work on all consecutively stored structures (i.e. VECTORs, UVECTORs, and STRINGs).

6. The SUBR PUTREST works on LISTs only, and should be used to append elements to the end of LISTs. To add elements to the 'front' of LISTs, use the construct (.ELEMENT !.LIST). Other ways of adding elements to LISTs create unnecessary garbage.

7. The SUBRs MEMQ and MEMBER can be used to find a MDL object in an arbitrary *structure*. MEMQ uses ==? as a test; MEMBER uses =?. Both return the original *structure* RESTed down to the MDL object which was found, or #FALSE ().

8.11. Practice Quiz

A large amount of important material has been covered in this chapter. Test your understanding by trying the following quiz, then check your answers by typing them to the MDL interpreter.

Please write below each line the result of typing that line into a MDL.

9. Programming Constructs

In order to write any interesting programs, an ability is required to test for various conditions and take action only if those conditions are met. This chapter introduces the MDL **SUBR**s and **FSUBR**s needed to do this.

9.1. Boolean Operators

9.1.1. NOT

The MDL predicate **NOT** takes one argument of any **TYPE**. It evaluates to **T** only if its argument evaluates to a **FALSE**, and to **#FALSE ()** otherwise.

```
<NOT <L=? 4 3>>$
T
```

9.1.2. AND

AND is an **FSUBR**, and it takes any number of arguments. It evaluates its arguments from first toward last as they appear in the **FORM**. As soon as one of them evaluates to a **FALSE**, it returns that **FALSE**, ignoring any remaining arguments. If none of them evaluate to **FALSE**, it returns **EVAL** of its last argument. **<AND>** returns **T**.

```
<AND <G? 4 3> <SET A 5> <L? 4 3> <SET B 7>>$
#FALSE ()
.A$
5
```

```
.B$

*ERROR*
UNBOUND VARIABLE
B
LVAL
LISTENING AT LEVEL n PROCESS 1

<AND <G? 4 3> <L? 3 4> <SET C 10>>$
10
.C$
10
```

AND? is the **SUBR** equivalent to **AND** (all its arguments are evaluated before any of them is tested).

9.1.3. OR

OR is also an **FSUBR** and also takes any number of arguments. It evaluates its arguments from first to last as they appear in the **FORM**. As soon as one of them evaluates to a non-**FALSE**, **OR** returns that non-**FALSE** value, ignoring any remaining arguments. If this never occurs, it returns the last **FALSE** it saw. **<OR>** returns **#FALSE ()**.

```
<OR <L? 4 3> <SET D 6> <SET E 13>>$
6
```

Setting **D** to **6** returned **6**, which is not a **FALSE**, so it was returned by **OR** and **E** was never set to **13**.

OR? is the **SUBR** equivalent to **OR**.

When you understand the following example, you should have no trouble with MDL's boolean operators. What is interesting about these two expressions?

```
<NOT <OR .FOO .BAR .BLETCH>>

<AND <NOT .FOO> <NOT .BAR> <NOT .BLETCH>>
```

9.2. COND

The MDL subroutine which is most used for varying evaluation depending on a truth value is the **FSUBR COND** ("conditional"). The arguments to **COND**, called **COND** clauses, must be **LISTs**, and

there must be at least one. **COND** always returns the result of the last evaluation it performs. The following rules determine the order of evaluations performed.

1. Evaluate the first element of each clause (from first toward last) until either a non-**FALSE** object results or the clauses are exhausted.

2. If a non-**FALSE** object is found in (1), immediately evaluate the remaining elements (if any) of that clause and ignore any remaining clauses.

In other words, **COND** goes walking down its clauses, **EVAL**ing the first element of each clause, looking for a non-**FALSE** result. As soon as it finds a non-**FALSE**, it forgets about all the other clauses and evaluates, in order, the other elements (if any) of the current clause and returns the last thing it evaluates. If it can't find a non-**FALSE** first element, it returns the last **FALSE** it saw.

9.2.1. Examples

```
<SET F '(1)>$
(1)
<COND (<EMPTY? .F>
       EMP)
      (<1? <LENGTH .F>>
       ONE)>$
ONE

<SET F ()>$
()
<COND (<EMPTY? .F>
       EMP)
      (<1? <LENGTH .F>>
       ONE)>$
EMP

<SET F '(1 2 3)>$
(1 2 3)
<COND (<EMPTY? .F>
       EMP)
      (<1? <LENGTH .F>>
       ONE)>$
#FALSE ()
<COND (<LENGTH? .F 2>
       SMALL)
      (BIG)>$
BIG
```

```
<DEFINE FACT (N)          ;"the standard recursive factorial"
        <COND (<0? .N> 1)
              (ELSE <* .N <FACT <- .N 1>>>)>>$
FACT
<FACT 5>$
120
```

In the last example, the use of **ELSE** was not necessary, but it makes it a bit easier to read the program. The atom **T** is often used for the same purpose.

9.3. Shortcuts with Conditionals

9.3.1. Using AND and OR with CONDs

Since **AND** and **OR** are **FSUBRs**, they can be used as miniature **CONDs**, but this is usually bad programming style. A construct of the form

<AND *pre-conditions action(s)***>**

or

<OR *pre-exclusions action(s)***>**

will allow *action(s)* to be evaluated only if all the *pre-conditions* are true or only if all the *pre-exclusions* are false, respectively. By nesting and using both **AND** and **OR**, fairly powerful constructs can be made. However, using **AND** and **OR** in this way can lead to some major problems. If any of your *actions* returns false or true unexpectedly, the following ones will never be evaluated. Even worse, programmers who get in the habit of doing this tend to write programs which are very difficult for anyone else to follow.

AND and **OR** are intended to be used in **COND** clauses. If you wanted to make sure that an argument called **ARG** passed to a function was a **FIX** between **5** and **10** inclusively:

```
<COND (<AND <==? <TYPE .ARG> FIX>
            <G=? .ARG 5>
            <L=? .ARG 10>>
       <what you want to do>)
      (ELSE <what you want to do otherwise>)>
```

If, instead, you wanted to make sure the argument was a **FIX** <u>outside</u> that range:

```
<COND (<AND <==? <TYPE .ARG> FIX>
            <OR <G? .ARG 10>
                <L? .ARG 5>>>
       <what you want to do>)
      (ELSE <what you want to do otherwise>)>
```

9.3.2. Embedded Unconditionals

One of the disadvantages of **COND** is that there is no straightforward way to do things unconditionally in between tests. One way around this problem is to insert a dummy clause that never succeeds, because its only element is an **AND** that returns a **FALSE** for the test (this method is strongly discouraged). Example:

```
<COND (<0? .N> <F0 .N>)
      (<1? .N> <F1 .N>)
      (<AND <SET N <* 2 <FIX </ .N 2>>>>
            ;"Round .N down to even number."
            <>>)
      (<LENGTH? .VEC .N> '[])
      (T <REST .VEC <+ 1 .N>>)>
```

The preferred method is to increase the nesting with a new **COND** after the unconditional part. This method does not make the code appear to a human reader as though it does something other than what it really does. The above example should be done this way:

```
<COND (<0? .N> <F0 .N>)
      (<1? .N> <F1 .N>)
      (T
       <SET N <* 2 <FIX </ .N 2>>>>
       <COND (<LENGTH? .VEC .N> '[])
             (T <REST .VEC <+ 1 .N>>)>)>
```

9.4. Examples

The following program will print all the prime factors of a given number:

```
<DEFINE FACTOR (N)
        <FACTOR-FROM .N 2>>

<DEFINE FACTOR-FROM (N TEST-DIVISOR)
        <COND (<G? <* .TEST-DIVISOR .TEST-DIVISOR> .N>
               <CRLF>
               .N)
              (<0? <MOD .N .TEST-DIVISOR>>
               <PRINT .TEST-DIVISOR>
               <FACTOR-FROM </ .N .TEST-DIVISOR>
                            .TEST-DIVISOR>)
              (ELSE <FACTOR-FROM .N <+ .TEST-DIVISOR 1>>)>>
```

If you are not familiar with recursion you should trace this by hand for a simple case like **<FACTOR 12>**. The first **COND** clause tests to see if the test divisor is greater than the square root of **N**. If it is, **N** must be prime so a carriage-return/line-feed is printed (see chapter 12) and the value of **N** is returned. The second clause checks whether **N** is divisible by **TEST-DIVISOR** and, if it is, prints that **TEST-DIVISOR** and then recursively calls **FACTOR-FROM** with the quotient of **N** and **TEST-DIVISOR** and with **TEST-DIVISOR** again. The third clause is executed if both the first and second return a **FALSE**. In that case, **FACTOR-FROM** is called recursively with **N** and **TEST-DIVISOR** incremented by one.

If you are confused about how this works, try typing it to a Muddle and experiment with it. You should be able to improve it (for instance there is no reason to test even numbers after **2** has been tested). Why does the program return only prime factors? Can you improve the program so that it tests only with prime numbers?

One way to write a test for prime numbers would be:

```
<DEFINE PRIME? (X)
        <==? .X <FACTOR .X>>>$
```

This would work, however you would probably want to write a new version of **FACTOR** for this which didn't print anything. Would this test for prime numbers improve **FACTOR-FROM**? Why not?

It is hard to imagine a program of any complexity without **COND** clauses. The following example is a very small part of a fairly famous program called Zork. (One of the implementors of Zork has been heard to say that Zork is a huge conditional).

```
<DEFINE RUSTY-KNIFE-FUNCTION ()
      <COND (<==? ,VERB TAKE>
             <COND (<IN? ,SWORD ,PLAYER>
                    <PRINC
"As you pick up the rusty knife, your sword gives a single
pulse of blinding blue light.">
                    <CRLF>)>
             <>)
            (<OR <AND <==? ,INDIRECT-OBJECT ,RUSTY-KNIFE>
                      <MEMQ ,VERB '[ATTACK KILL]>>
                 <AND <MEMQ ,VERB '[SWING THROW]>
                      <==? ,DIRECT-OBJECT ,RUSTY-KNIFE>
                      ,INDIRECT-OBJECT>>
             <REMOVE ,RUSTY-KNIFE>
             <JIGS-UP
"As the knife approaches its victim, your mind is submerged by
an overmastering will.  Slowly, your hand turns, until the
rusty blade is an inch from your neck.  The knife seems to sing
as it savagely slits your throat.">)>>
```

This function is called whenever "the rusty knife" is referred to in any way. This function checks whether the verb is "take" and the player has the "sword." If so the first message is printed and `#FALSE ()` is returned. If the verb was not "take", it checks whether either the indirect object is "rusty knife" and the verb is "attack" or "kill", or the verb is "swing" or "throw" and the direct object is "rusty knife" and there is an indirect object. If so, the "rusty knife" is removed from the game, an interesting message is printed, and the player dies.

10. Looping

One of MDL's strongest points is its variety of powerful looping constructs. These will be covered in this chapter.

10.1. PROG

PROG makes it possible to encapsulate sections of MDL code. A **PROG** is very much like a **FUNCTION** in syntax. It takes a **LIST** which is similar in some respects to an argument list, and an arbitrary number of MDL objects which are **EVAL**uated in turn. It returns the result of the **EVAL**uation of the last object in its body. Here is a prosaic **PROG**:

```
<PROG () <SETG A <>> <SETG B <>> <INITIALIZE>>
```

Notice that each of the three **FORM**s in the **PROG** could have been done without using a **PROG**. **PROG**s, however, are a bit more useful than this would indicate.

First, the **LIST** can contain any number of

- **ATOM**s

- **LIST**s containing an **ATOM** and an arbitrary initial value for that **ATOM**.

All of these **ATOM**s will be re-bound inside the **PROG** (i.e. as if a new **FUNCTION** were entered.) When the **PROG** returns, the **ATOM**s will be unbound (i.e. re-bound to their old values, if any.) Thus, a **PROG** can be thought of as a mini-**FUNCTION** of no arguments.

```
<PROG (A B (C 10) (D .FOO)) ....>
```

In this example, four new bindings are made. The **ATOM**s **A** and **B** are bound, but not assigned a value. The **ATOM C** is bound to **10** and the **ATOM D** is bound to the current **LVAL** of the **ATOM FOO**. **ATOM**s should be placed in this 'argument' **LIST** when they are used as temporary variables inside the **PROG**. A full explanation of the use of temporary variables is in section 11.5.

More importantly, a **PROG** can be restarted or caused to return from the middle *any time* using the **SUBR**s **AGAIN** and **RETURN**. At this point, it is sufficient to say that **AGAIN** with no arguments starts

executing the body of the **PROG** from the beginning (but bindings are *not* redone). **RETURN** of one argument forces the **PROG** to return that argument. Notice that **AGAIN** and **RETURN** as described will always refer to the nearest surrounding **PROG** *in the current* **FUNCTION**.

PROG turns out to be fairly useless in MDL, but the **FSUBR REPEAT**, which is very similar, is enormously useful.

10.2. REPEAT

REPEAT has the same syntax as **PROG** and may be thought of as a **PROG** in which the last item in the body is **<AGAIN>**. In other words, the body of the **REPEAT** will be repeatedly executed until a **RETURN** is done. There is no other way to leave a **REPEAT** except with a **RETURN**.

```
<REPEAT ((CNT 5))
     #DECL ((CNT) FIX)
     <COND (<L? <SET CNT <- .CNT 1>> 0> <RETURN T>)
           (T <PRINT .CNT>)>>$

4
3
2
1
0 T
```

10.3. Non-local RETURNs, etc.

There are cases in which one might like to **RETURN** or **AGAIN** to someplace other than the nearest **PROG** or **REPEAT**, or for that matter someplace in a different **FUNCTION**. MDL allows you to 'name' any **PROG**, **REPEAT**, or **FUNCTION** by placing the **STRING "NAME"** followed by an **ATOM** at the *end* of an argument list or **PROG/REPEAT** list. This has the effect of binding that **ATOM** to an object of **TYPE ACTIVATION** which becomes a legal additional argument to both **AGAIN** and **RETURN**. Thus, **AGAIN** can take an optional **ACTIVATION**, and **RETURN** takes a return value and an optional **ACTIVATION**.

The most common use of **RETURN/AGAIN**s to 'named activations' is in error handling. Assume that you have a **FUNCTION FOO** which calls a **FUNCTION BAR** which calls a **FUNCTION BLETCH** which notices something wrong. **BLETCH** might want to cause **FUNCTION FOO** to return a **FALSE**, for example, or print an error message. This is only possible if the **FUNCTION FOO** is defined to have a 'named activation', whose 'name' is known to **FUNCTION BLETCH**.

```
<DEFINE FOO (A)
    <COND (<PROG ("NAME" ACT) <BAR .A>>
           <PRINT .A> T)
          (T <PRINT "ERROR IN YOUR PROGRAM"> T)>>$
FOO
<DEFINE BAR (X)
    <BLETCH <* .X .X>>>$
BAR
<DEFINE BLETCH (Z)
    <COND (<G? .Z 10> <RETURN #FALSE () .ACT>)
          (T <SQRT .Z>)>>$
BLETCH
<FOO 2>$
2 T
<FOO 4>$
"ERROR IN YOUR PROGRAM" T
```

10.4. MAPF

10.4.1. Looping Through a Structure

MAPF (pronounced 'map-eff' for 'map-first') is mainly used to apply a *function* to each element of a *structure*, in turn. In this most simple form, its first argument is a **FALSE**, its second argument a *loop-function*, and its third argument a *structure*. Here is a simple **MAPF**:

```
<MAPF <>
      <FUNCTION (X)
          <PRINT .X>>
      (1 2 3 4)>$
1
2
3
4 4
```

The last **4** is the result of the **MAPF** (the result of the last application of the *loop-function* to an element of the *structure*.

An **FSUBR** called **FUNCTION** is used in many places in this chapter. **FUNCTION** is very much like **DEFINE**, except that no name is specified. **FUNCTIONs** created with **FUNCTION** are said to be 'anonymous'. They cannot be used outside the **FORM** in which they are imbedded, since they have no name by which they can be referenced. Of course, if the *loop function* you wish to use had already been **DEFINEd**, you would refer to it in a **MAPF** as the global value of its name.

```
<DEFINE FOO (A B) <+ .A .B>>$
FOO
.FOO$
#FUNCTION ((A B) <+ .A .B>)
<FUNCTION (A B) <+ .A .B>>$
#FUNCTION ((A B) <+ .A .B>)
```

A **MAPF** can be prematurely stopped at any time if the *loop-function* calls the **SUBR MAPLEAVE**. **MAPLEAVE** takes one argument: it stops the **MAPF** and causes the **MAPF** to return its argument.

```
<DEFINE BAR (L)
    <MAPF <>
          <FUNCTION (X)
              <COND (<G? .X 10> <MAPLEAVE FOO>)
                    (T <PRINT .X>)>>
          .L>>$
BAR

<BAR (1 2 3 4)>$
1
2
3
4 4

<BAR (1 2 15 7 5 2)>$
1
2 FOO
```

10.4.2. Other Than One Structure

One can simultaneously loop through any number of *structures* using **MAPF**. **MAPF** will apply the *loop-function* to the first elements of each of the *structures*. The **MAPF** will stop when *any* of the *structures* becomes **EMPTY?**.

```
<MAPF <>
      <FUNCTION (A B C)
          <PRINT <+ .A .B .C>>>
      (1 3 5 7)
      (2 4 6 8)
      (3 5 7 9)>$
6
12
18
24 24
```

```
<MAPF <>
      <FUNCTION (A B C)
          <PRINT <+ .A .B .C>>>
      (1 3 5)
      (2 4 6)
      (3)>$
 6 6
```

'Other Than One' also includes zero, but this is a special case. A **MAPF** with only two arguments is something like a **REPEAT** loop. It can only be terminated by an explicit call to **MAPLEAVE**. See section 10.5. If any of the structures is empty to begin with, **MAPF** returns **#FALSE ()**.

10.4.3. Using Intermediate Results

By now you must be wondering why there is a **FALSE** as the first argument to **MAPF**. In fact, a **FALSE** tells MDL not to do anything with the results of applying the *loop-function* to the elements of the *structure*. However, if the first argument to **MAPF** is something which can be applied to arguments (i.e. a **FUNCTION** or **SUBR**), then MDL will 'save' the results of applying the *loop-function* and, when the looping is finished, apply the first argument (called the *final-function*) to *all* of the 'saved' results. An example:

```
<MAPF ,LIST
      <FUNCTION (X Y)
          <+ .X <SQRT .Y>>>
      (1 2 3)
      (1 4 9)>$
(2 4 6)

<MAPF ,+
      <FUNCTION (X Y)
          <+ .X <SQRT .Y>>>
      (1 2 3)
      (1 4 9)>$
12
```

In the first case, we built a **LIST** out of the results of the *loop-function*. In the second, we simply added up all of the results.

10.4.4. MAPRET and MAPSTOP

There are cases in which you might want to have an arbitrary number of results 'saved'. This can be done with the **SUBR MAPRET** which takes any number of arguments (including zero), causes the

function to terminate, and 'saves' all of its arguments.

```
<DEFINE PRIME-LIST (L)
    <MAPF ,LIST
          <FUNCTION (NUM)
              <COND (<PRIME? .NUM> .NUM)
                    (T <MAPRET>)>>
          .L>>$
PRIME-LIST
<PRIME-LIST (2 4 11 55 73)>$
(2 11 73)
```

What happened here was that only prime numbers were allowed to be 'saved'. Whenever **PRIME?** returned **FALSE**, a **MAPRET** of no arguments was done; thus, no values were 'saved' for this call to the *loop-function*. **MAPRET**, of course, will not work if there is no *final-function* to return results to.

Assuming the *function* **PRIME?** described in chapter 9 has been written:

```
<DEFINE PRIME-AND-SQUARES-LIST (L)
    <MAPF ,LIST
          <FUNCTION (NUM)
              <COND (<PRIME? .NUM> <MAPRET .NUM <* .NUM .NUM>>)
                    (T <MAPRET>)>>
          .L>>$
PRIME-AND-SQUARES-LIST
<PRIME-AND-SQUARES-LIST (2 4 11 55)>$
(2 4 11 121)
```

A more useful function:

```
<DEFINE UPPERCASIFY-1 (STR)
        <MAPF ,STRING
              <FUNCTION (CHAR)
                   <SET ASC <ASCII .CHAR>>
                   <COND (<AND <G=? .ASC <ASCII !\a>>
                               <L=? .ASC <ASCII !\z>>>
                          <MAPRET <ASCII <- .ASC 32>>>)
                         (ELSE <MAPRET .CHAR>)>>
              .STR>>$
UPPERCASIFY
<SET Z-STR "Now is the time for all good men to <FOO .BAR>">
<UPPERCASIFY .Z-STR>$
"NOW IS THE TIME FOR ALL GOOD MEN TO <FOO .BAR>"
.Z-STR$
"Now is the time for all good men to <FOO .BAR>"
```

The **SUBR MAPSTOP** is the same as **MAPRET**, except that, after 'saving' its arguments, it finishes the **MAPF**, allowing the *final-function* to be applied to all of the 'saved' results. Like **MAPRET**, **MAPSTOP** can only be used if there is a *final-function*.

10.4.5. MAPR

The **SUBR MAPR** (for 'map-rest,' pronounced 'map-ar') is exactly like **MAPF** in every respect except that the arguments passed to the *loop-function*, rather than being successive elements of the *structures*, are the *structures* themselves **REST**ed down successively. The names for the two *map* **SUBR**s are mnemonic: **MAPF**irst and **MAPR**est.

```
<MAPR <>
      <FUNCTION (X)
          <PRINT .X>>
      (1 2 3 4)>$
(1 2 3 4)
(2 3 4)
(3 4)
(4) (4)
```

MAPR is useful if it is necessary to change elements of the *structure(s)* that you are *map*ping down. Here is a **FUNCTION** which takes a structure full of numbers and changes it to contain double the old values:

```
<DEFINE DOUBLE (STR)
    <MAPR <>
          <FUNCTION (S)
              <PUT .S 1 <* <1 .S> 2>>>
          .STR>>$
DOUBLE
<SET L (1 2 3)>$
(1 2 3)
<DOUBLE .L>$
(6)
.L$
(2 4 6)
```

In **UPPERCASIFY-1** a **MAPF** was used which generated a new structure. Using **MAPR**, a new function can be written which modifies the original string:

```
<DEFINE UPPERCASIFY-2 (STR)
        <MAPR <>
              <FUNCTION (STR1)
                   <SET ASC <ASCII <1 .STR1>>>
                   <COND (<AND <G=? .ASC <ASCII !\a>>
                               <L=? .ASC <ASCII !\z>>>
                          <PUT .STR1 1 <ASCII <- .ASC 32>>>)>>
              .STR>
        .STR>$
UPPERCASIFY-2
<UPPERCASIFY-2 <SET STR "Now is the time for <BAR .BLETCH>">>$
"NOW IS THE TIME FOR <BAR .BLETCH>"
.STR$
"NOW IS THE TIME FOR <BAR .BLETCH>"
```

MAPR is not always used to change an existing structure. The following example shows another use. This function marches down a structure and builds a new structure of the same type in which the elements of the first structure appear only once.

```
<DEFINE UNIQUIFY (STRUC)
        <COND (<NOT <STRUCTURED? .STRUC>>
               #FALSE ("WRONG TYPE OF ARGUMENT"))
              (ELSE
               <CHTYPE
                <MAPR ,<PRIMTYPE .STRUC>
                      <FUNCTION (S)
                                <COND (<MEMQ <1 .S> <REST .S>>
                                       <MAPRET>)
                                      (ELSE <MAPRET <1 .S>>)>>
                      .STRUC>
               <TYPE .STRUC>>)>>$
UNIQUIFY
<UNIQUIFY #FROB (1 2 33 2 1 5)>$
#FROB (33 2 1 5)
```

If you wished to be able to remove elements which "look the same," such as structures which are **=?** (like `(1 2 3)` and `(1 2 3)`, or `"FROTZ"` and `"FROTZ"`), you would have to replace the **MEMQ** with **MEMBER**, which is slower.

10.4.6. MAPF/R Summary

The syntax for **MAPF/R** is as follows:

```
<MAPF/R final-function
        loop-function
        structure-1
        ...
        structure-n>
```

with only the first two arguments required.

10.5. Looping vs. Recursion

In the previous chapter, the "standard recursive factorial" was shown. It can now be rewritten using the looping constructs introduced in this chapter.

```
<DEFINE FACT (N)
        <REPEAT ((ANS 1))
                <COND (<0? .N> <RETURN .ANS>)
                      (ELSE <SET ANS <* .ANS .N>>
                            <SET N <- .N 1>>)>>>
```

Some might argue that this is a larger and more complicated program to write than the recursive form, and therefore inferior. The iterative form just shown, however, is faster and more efficient.

The same program can also be written using **MAPF**.

```
<DEFINE FACT (N)
        <SET ANS 1>
        <MAPF <>
              <FUNCTION ()
                        <COND (<0? .N> <MAPLEAVE .ANS>)
                              (ELSE <SET ANS <* .ANS .N>>
                                    <SET N <- .N 1>>)>>>>
```

Or, more elegantly:

```
<DEFINE FACT (N)
        <MAPF ,*
              <FUNCTION ()
                  <COND (<0? .N> <MAPSTOP>)
                        (ELSE <+ 1 <SET N <- .N 1>>>)>>>>
    ;"<FACT 0> will return 1 since * of no arguments
      returns 1."
```

As pointed out earlier, a **MAPF** does not have to take any structures as arguments as long as the second argument, the *looping function*, does not take any arguments.

Although recursion can be a very simple and elegant way to solve a problem, iteration is often more efficient. Take, for example, the factorial example of chapter 9. If you only intend to use the function with very small numbers the recursive form will not cost much to use and does have the advantage of being slightly easier to write. However, since every recursive call of a function requires the creation of a new 'environment,' calculating factorial of a large number will take a lot of time and computer memory. The iterative forms of factorial shown above would be *much* more efficient.

In summary, the advantage of the looping techniques described in this chapter over recursion is that the overhead of calls is eliminated. However, a long program (say, bigger than half a printed page) may be more difficult to write iteratively than recursively and hence more difficult to maintain. A program whose repetition is controlled by a structured object (for example, "walking a tree" to visit each monad in the object) often should use looping for covering one "level" of the structure and recursion to change "levels".

11. Argument Lists in FUNCTIONs

In Chapter 6, the creation of a simple type of **FUNCTION** was explained: a **FUNCTION** taking a fixed number of arguments all of which get **EVAL**ed. While this may be sufficient for writing most of your **FUNCTION**s, there are other ways in which you might like arguments to be handled. Some of these might include:

- **FUNCTION**s which can take an arbitrary number of arguments (like the **SUBR**s +, -, **LIST**, **VECTOR**, etc.)

- **FUNCTION**s which act more like **FSUBR**s (i.e. they don't have their arguments **EVAL**ed.)

- **FUNCTION**s which can take optional arguments, which can be defaulted.

In fact, all of these things (and a few more) can be done easily by specifying them in the argument list of the **FUNCTION**. The remainder of this chapter will describe the complete syntax for MDL argument lists.

11.1. Arguments Not EVALed

Placing a single-quote before an **ATOM** in the argument list will cause that **ATOM** to be given the value of its respective argument *without* **EVAL**uation.

```
<DEFINE FOO ('ITEM) .ITEM>$
FOO
<FOO <+ 1 2>>$
<+ 1 2>
```

Were the **ATOM ITEM** not quoted in the argument list, the **FUNCTION** would have returned the **FIX 3**. Quoting arguments, as it turns out, is not used often in MDL.

11.2. Optional Arguments

MDL can be told to expect optional arguments by placing the **STRING "OPTIONAL"** in the argument list *after* all of the required arguments. Following the **STRING** can be any number of **ATOMs**, which will be bound to the values of the optional arguments, if given. To specify that an optional argument is to have a default value (i.e. if not passed as an argument), place a **LIST** containing the **ATOM** and the default value in place of just the **ATOM**. Here's an example:

```
<DEFINE ADD-ONE (NUM "OPTIONAL" (HOW-MANY 1))
        <+ .NUM .HOW-MANY>>$
ADD-ONE
<ADD-ONE 10>$
11
<ADD-ONE 10 2>$
12
```

This rather useless **FUNCTION** adds the **LVAL** of **HOW-MANY** to its first argument. **HOW-MANY** is an optional argument, whose default value is the **FIX 1**. Therefore, with one argument, **ADD-ONE** adds one to its argument. With two arguments, it adds them.

As was mentioned earlier, it isn't necessary to supply a default value for an optional argument. If there is no default value, and the optional argument is not supplied, the **ATOM** gets bound, but is not assigned a value. **LVAL** of that **ATOM** will generate an error, because an **ATOM** must be *both* bound and assigned to have a local value. One can tell whether an **ATOM** has been assigned a value by using the **SUBR ASSIGNED?**, which returns **T** if its argument (an **ATOM**) is assigned; otherwise **#FALSE ()**. The following definition of **ADD-ONE** acts identically to the previous one:

```
<DEFINE ADD-ONE (NUM "OPTIONAL" HOW-MANY)
        <COND (<NOT <ASSIGNED? HOW-MANY>>
               <SET HOW-MANY 1>)>
        <+ .NUM .HOW-MANY>>$
ADD-ONE
```

The use of single-quoted **ATOMs** is allowed with optional arguments as well as required ones. You may supply your own example, if you can think of one. We can't.

11.3. Arbitrary Numbers of EVALed Arguments

At any place in the argument list, after any required and optional (if any) arguments, you can specify that *all* of the remaining arguments (supplied at the time of call) be **EVALed** and grouped together in a special structure called a **TUPLE** (for all practical purposes, **TUPLEs** may be thought of as **VECTORs**, and can be manipulated in the same ways). To do this, place the **STRING "TUPLE"** followed by an **ATOM** in the argument list. The **ATOM** will be bound to the **TUPLE**. Here are

some examples:

```
<DEFINE MY+ ("TUPLE" NUMBERS) <+ !.NUMBERS>>$
MY+
<MY+ 1 2 3 4 5 6>$
21
<MY+>$
0

<DEFINE MY-STRING ("TUPLE" STRINGS) <STRING !.STRINGS>>$
MY-STRING
<MY-STRING "THIS" "IS" "A" "BIG" "STRING" !\!>$
"THISISABIGSTRING!"

<DEFINE TIMES-PLUS (NUM "TUPLE" NUMBERS)
        <* .NUM <+ !.NUMBERS>>>$
TIMES-PLUS
<TIMES-PLUS 4 1 2 3>$
24
```

11.4. Arbitrary Numbers of un-EVALed Arguments

Instead of using **"TUPLE"**, one could have used the **STRING "ARGS"**. This has the effect of binding the following **ATOM** to a **LIST** of all of the remaining arguments, *un*EVAL*uated*. In fact, the **ATOM** is bound to the **LIST** which is the **FORM** used to call the **FUNCTION REST**ed down to the remaining arguments. The use of **"ARGS"** allows one to write **FSUBR**s in MDL.

```
<DEFINE FOO ("ARGS" L) .L>$
FOO
<SET F '<FOO 1 2 3>>$
<FOO 1 2 3>
<SET LL <EVAL .F>>$
(1 2 3)
<==? .LL <REST .F 1>>$
T
```

In the previous example, we explicitly called the **SUBR EVAL**, which caused **EVAL**uation of the **FORM** **<FOO 1 2 3>**. This returned the **LIST (1 2 3)**, which is **==?** to **<REST .F 1>**.

Now we will write a **FUNCTION** to simulate the **FSUBR DEFINE** in MDL: this is just what MDL does internally when the **FSUBR DEFINE** is called.

```
<DEFINE MY-DEFINE (NAM "ARGS" L)
        <SETG .NAM <CHTYPE .L FUNCTION>>
        .NAM>$
MY-DEFINE
<MY-DEFINE FOO (A B C) <+ .A .B .C>>$
FOO
,FOO$
#FUNCTION ((A B C) <+ .A .B .C>)
<FOO 1 2 3>$
6
```

Now that we have simulated **DEFINE**, let's try our hand at **AND**.

```
<DEFINE MY-AND ("ARGS" L)
        <REPEAT ((LAST T))
            <COND (<EMPTY? .L> <RETURN .LAST>)
                  (<NOT <SET LAST <EVAL <1 .L>>>>
                   <RETURN .LAST>)
                  (T <SET L <REST .L>>)>>>$
MY-AND
```

This will exactly simulate the behavior of **AND**. The **REPEAT** loop initializes the **ATOM LAST** to **T**, because **AND** of no arguments is defined to return **T**. The loop itself first checks on whether the **LIST L** has become **EMPTY?**. If so, the **AND** was successful, and **LAST** is returned. Otherwise, **LAST** is **SET** to **EVAL** of **<1 .L>**. If that is a **FALSE**, it is returned. Otherwise, **L** is **REST**ed once and the loop is repeated.

As an exercise, write **OR** and **COND**. It is legitimate to use **COND** in your **COND** simulator, but if you call your simulator **COND**, watch out.

11.5. Temporary Variables

You may recall that chapter 6 referred to "free variables" as those variables (**ATOM**s) whose local values are **SET** or accessed inside a **FUNCTION**, but which are not bound inside that function. One should always avoid using "free variables" in MDL programs, but there is often a need for variables whose values will contain temporary results. You can specify such variables to be bound inside a **FUNCTION** by including the **"AUX"** (for auxiliary) followed by any number of **ATOM**s or **LIST**s of **ATOM**s and values (like **"OPTIONAL"** arguments) at the end of the argument list.

```
<DEFINE SUM ("TUPLE" NUMS "AUX" (SUM 0))
        <REPEAT ()
            <COND (<EMPTY? .NUMS> <RETURN .SUM>)
                  (T
                   <SET SUM <+ .SUM <1 .NUMS>>>
                   <SET NUMS <REST .NUMS>>)>>>$
SUM
<SUM 1 2 3 4>$
10
```

The **FUNCTION SUM**, in this example, simulates the **SUBR +**. The **ATOM SUM** is initialized to zero in the argument list. The following is identical in effect, although poor in style:

```
<DEFINE SUM ("TUPLE" NUMS "AUX" SUM)
        <SET SUM 0>
        <REPEAT ... >>
```

It should be noted that the part of the argument list which follows **"EXTRA"** or **"AUX"** is identical in syntax and meaning to the 'argument list' which is the first argument to **PROG** and **REPEAT**.

11.6. Order of Evaluation in Argument Lists

Unlike many other languages, including LISP, **ATOM**ic bindings *after* the required arguments are done from left to right, rather than simultaneously. This means that, for example, the default values for optional arguments and extra variables can refer to the values of other **ATOM**s to their left in the argument list.

```
<DEFINE FOO (A "OPTIONAL" (B <+ .A 10>) "AUX" (C <FOOBAR .B>))
        ....>
```

The previous example shows an example of what is possible in argument lists due to MDL's order of evaluation.

11.7. Variable Declarations

MDL has a built-in facility for checking the **TYPE**s of arguments to **FUNCTION**s as well as other temporary variables. This is analogous to the checking which is done when **F/SUBR**s are called: if you call the **SUBR +** with an **ATOM**, for example, MDL will generate an error. In MDL, variables can be declared to be of a certain **TYPE** or group of **TYPE**s. This is done by placing an object of **TYPE DECL** (**PRIMTYPE LIST**) immediately after the **FUNCTION**'s argument list. It may also follow the

argument list of a **PROG** or **REPEAT** and declare the variables bound within. The **DECL** (for "declaration," pronounced 'deckle') has the form of repeating pairs of **LIST**s of **ATOM**s (the **ATOM**s to be declared) and the declaration proper. The simplest **TYPE** declaration is the name of a **TYPE** or the **ATOM ANY**.

```
#DECL ((FOO BAR) FIX (BLETCH) ATOM (MUMBLE) ANY)
```

This declares the **ATOM**s **FOO** and **BAR** to be **FIX**es, the **ATOM BLETCH** to be an **ATOM**, and the **ATOM MUMBLE** to be anything.

Another declaration form is the union of different **TYPE**s, which is specified by a **FORM** whose first element is the **ATOM OR** and the remainder legal **TYPE** names.

```
#DECL ((NUM) <OR FIX FLOAT> (STRUC) <OR LIST VECTOR>)
```

Also useful is the form **<PRIMTYPE** *name-of-a-PRIMTYPE*>, which specifies anything of that **PRIMTYPE**. For example:

```
#DECL ((PL) <PRIMTYPE LIST>)
```

will allow **PL** to be a **LIST**, a **FORM**, a **SEGMENT**, or any other **PRIMTYPE LIST**.

In fact, the full-blown MDL declaration syntax is far more baroque than has been described, but these simple forms will suffice in almost all cases. For more information on **DECL**, consult *The MDL Programming Language* [Galley 79]. Here are some examples of old friends, now including **DECL**s.

```
<DEFINE MY-AND ("ARGS" L)
        #DECL ((L) LIST)
        <REPEAT ((LAST T))
            #DECL ((LAST) ANY)
            <COND (<EMPTY? .L> <RETURN .LAST>)
                  (<NOT <SET LAST <EVAL <1 .L>>>>
                   <RETURN .LAST>)
                  (T <SET L <REST .L>>)>>>$
MY-AND
```

```
<DEFINE ADD-ONE (NUM "OPTIONAL" (HOW-MANY 1))
        #DECL ((NUM) <OR FIX FLOAT> (HOW-MANY) FIX)
        <+ .NUM .HOW-MANY>>$
ADD-ONE
<ADD-ONE 2.3 1.2>$

*ERROR*
TYPE-MISMATCH
HOW-MANY  ;"The ATOM of incorrect TYPE"
FIX       ;"The DECL for that ATOM"
1.2       ;"What the ATOM was about to be SET to"
EVAL      ;"EVALing the FORM <ADD-ONE 2.3 1.2> caused it"
LISTENING-AT-LEVEL 2 PROCESS 1
```

Declarations have a number of purposes. First, they make your code easier for someone else to understand, as the sorts of arguments your **FUNCTIONs** take can be deduced from them. It will also help you read your own code at a later time when you may have forgotten how it all works. Second, it helps in debugging your programs, since an error will be caused if the declaration is violated. Finally, when your **FUNCTIONs** eventually get compiled, much better code can be produced with the information given by your declarations. Always **DECL** your **FUNCTIONs**!

11.8. Structures: DECLs and NEWTYPEs

Before closing our discussion of **DECLs**, one special type of declaration should be considered: that of *structures*. The syntax for this declaration is:

```
<type-name <PRIMTYPE typeprim>
            declaration-for-first-element
            declaration-for-second-element
            ...
            declaration-for-last-element>
```

For example, we could **DECL** a **VECTOR** of three elements as follows:

```
<VECTOR FIX LIST <VECTOR ATOM ATOM>>
```

This declares the **VECTOR** to have a **FIX**, a **LIST**, and a **VECTOR** which must contain two **ATOMs**. There may be more elements in a *structure* than those **DECLed**. Any additional elements will be considered to have the **DECL ANY**.

```
<<PRIMTYPE LIST> ATOM <OR FIX FLOAT>>
```

This declares a *structure* of **PRIMTYPE LIST** containing an **ATOM** and either a **FIX** or a **FLOAT**.

We originally described the **SUBR NEWTYPE** as taking two arguments: the new **TYPE** name and its **TYPEPRIM**. *Structured* **NEWTYPE**s can take a third argument as well: a declaration, as described above. Let's use the airline problem from our earlier discussion. We will define a **FLIGHT** as follows:

```
<NEWTYPE FLIGHT
                  VECTOR
                  '<<PRIMTYPE VECTOR> ATOM FIX FIX>>$
FLIGHT
<SETG AIRLINE 1>$
1
<SETG FLIGHT-NUMBER 2>$
2
<SETG DURATION 3>$
3
```

Notice that the declaration of **FLIGHT** is quoted: this is because it is a **FORM** and **NEWTYPE** is a **SUBR**.

Now that **FLIGHT** is a legal **TYPE**, it can be used in declarations. In fact, it is a lot easier to say

```
#DECL ((FL) FLIGHT)
```

than to say

```
#DECL ((FL) <VECTOR ATOM FIX FIX>)
```

especially when you add another ten elements to the definition of **FLIGHT**s. It is also a lot clearer for both yourself and others to read.

11.8.1. To NEWTYPE or Not To NEWTYPE

That is the question most frequently asked. Should I make my table of house members a **NEWTYPE**? Should it just be a **VECTOR**? Sad to tell, there is no cut and dried answer. In general, whenever a structure has a 'significant' amount of internal structure, or some readily understood 'outside world meaning', it is a good idea to make it a **NEWTYPE**. Most people would deem a structure to have 'significant' internal structure at the point when they type out the whole darn **DECL** for the ninety-fourth time. Others think ahead.

11.9. Good Habits / Bad Habits

This chapter has some suggestions for good programming practice. You may ignore them at your risk, but we have found that people learning MDL are always more successful if they develop good habits early on in their MDLing. Here are the good habits:

- *Always* use **"AUX"** to bind temporary variables in your functions. *Don't* use "free variables"!

- *Always* **DECL** your **FUNCTION**s, **PROG**s, and **REPEAT**s. Even if a variable can have *any* value, it is good practice to **DECL** it as such, so that it is clear that you haven't simply forgotten.

11.10. Review of Argument List Syntax

Here is a full-blown, ultra-hairy, and incredibly strange argument list:

```
<DEFINE HAIR (A 'B "OPTIONAL" (C 10) D
                   "TUPLE" NUMS
                   "AUX" (E <+ .A .B>)
                         (F <SQRT </ .E .C>>)
                   "NAME" FOO)
        #DECL ((B C E) FIX
               (A E) <OR FIX FLOAT>
               (F D) FLOAT
               (NUMS) TUPLE
               (FOO) ACTIVATION)
        <COND (<NOT <ASSIGNED? D>>
               <SET D <ATAN .B>>)>
        <+ .A .B .C .D .E .F !.NUMS>>$
HAIR
```

This poor excuse for a **FUNCTION** takes two required arguments, the second of which is un**EVAL**uated, two optional arguments, one of which defaults to **10**, and any number of other arguments, bunched together in a **TUPLE** called **NUMS**. Two temporary variables **E** and **F** are also used, both of which refer to the **LVAL**s of other **ATOM**s to their left in the argument list.

12.2. Conversion I/O - Input

All of the following input Subroutines, when directed at a terminal, hang until $ (ESC) is typed and allow normal use of rubout, ↑D, ↑L and ↑@.

12.2.1. READ

<READ>

This returns the entire MDL object whose character representation is next in the input stream. Successive <READ>s return successive objects. This is precisely the **SUBR READ** mentioned in chapter 3 (page 15).

12.2.2. READCHR

<READCHR>

("read character") returns the next **CHARACTER** in the input stream. Successive <READCHR>s return successive **CHARACTER**s.

12.2.2.1. NEXTCHR

<NEXTCHR>

("next character") returns the **CHARACTER** which **READCHR** will return the next time **READCHR** is called (if **READCHR** is the next input **SUBR** called. Multiple <NEXTCHR>s, with no input operations between them, all return the same thing.

12.3. Conversion I/O - Output

If an object to be output requires (or can tolerate) separators within it (for example, between the elements in a structured object or after the **TYPE** name in "# notation"), these conversion-output **SUBR**s will use a carriage-return/line-feed separator to prevent overflowing a line. Overflow is detected in advance from elements of the **CHANNEL** in use.

12.3.1. PRINT

<PRINT *any*>

This outputs, in order,

1. a carriage-return line-feed,
2. the character representation of EVAL of its argument (PRINT is a SUBR), and
3. a space

and then returns EVAL of its argument. This is precisely the SUBR PRINT mentioned in chapter 3 (page 15).

12.3.2. PRIN1

<PRIN1 *any*>

outputs just the character representation of, and returns, EVAL of *any*.

12.3.3. PRINC

<PRINC *any*>

("print characters") acts exactly like PRIN1, except that

1. if its argument is a STRING or a CHARACTER, it suppresses the surrounding "s or initial !\ respectively; or,
2. if its argument is an ATOM, it suppresses any \s or OBLIST trailers which would otherwise be necessary.

If PRINC's argument is a structure containing STRINGs, CHARACTERs, or ATOMs, the services mentioned will be done for all of them. Ditto for the ATOM used to name the TYPE in "# notation".

12.3.4. CRLF

<CRLF>

("carriage-return line-feed") outputs a carriage-return line-feed and then returns **T**.

12.4. CHANNEL (the TYPE)

MDL I/O 'channels' are represented by an object of **TYPE CHANNEL**, which is of **PRIMTYPE VECTOR**. The internal structure of a **CHANNEL**s is not frequently examined or manipulated. Those interested can consult the MDL manual for details.

12.4.1. OPEN

The **SUBR OPEN** is used to create and return a **CHANNEL**. It takes two arguments, a *mode* and a *file-name*, both of which must be **STRING**s. If successful, **OPEN** returns a **CHANNEL**; otherwise, it returns a **FALSE** containing the reason for the failure and the *file-name* (both **STRING**s.)

There are two commonly used *modes*: **"READ"** and **"PRINT"**. These are used, reasonably enough, for input and output, respectively. These modes input and output ASCII characters (i.e. conversion I/O).

File names are dependent on the host operating system. The following applies only to TOPS-20 systems. File names are composed of four parts: the device, the directory, a first file name, and a second file name. A typical file name might be:

```
"DSK:<MARC>CALCULATOR.MUD"
```

MDL will use certain defaults for these four parts, if they are not specified explicitly. These are **DSK**, your working directory, **INPUT**, and **MUD**, respectively. These defaults can be overridden by **SETG**ing the **ATOM**s **DEV**, **SNM**, **NM1**, and **NM2** to the defaults you desire. These defaults must be **STRING**s. For some devices, some of the four parts of the file name are ignored, for example the line printer and the terminal (called **TPL** and **TTY**).

Here are some examples of the uses of **OPEN**:

<OPEN "PRINT" "TPL:"> opens an output channel to the line printer.

<OPEN "PRINT" "<MARC>FOO"> opens an output channel to a disk file called **FOO.MUD**. Remember that the default device is **DSK** (i.e. the disk) and the default second file name is **MUD**.

<OPEN "READ" "FOO.TEST"> opens an input channel to a disk file called **FOO.TEST** in the default file directory (*i.e.* **MARC**).

It is good practice to give all of your MDL files a second name of **MUD**. This allows you to make use of the MDL default second file name and also makes it easier for both you and others to find files of

MDL code. In general, files containing only text should be given the second file name **TXT**.

12.4.2. FILE-EXISTS?

FILE-EXISTS? tests for the existence of a file without creating a **CHANNEL**, which occupies about a hundred machine words of storage. It takes a file-name argument (like **OPEN**) and returns either **T** or a **FALSE** containing the reason (a **STRING**).

12.4.3. CLOSE

CLOSE, given a **CHANNEL**, closes that **CHANNEL**. An error will occur if any input or output is directed to a **CLOSE**d **CHANNEL**.

It is possible to tell whether a **CHANNEL** is currently 'open' or has been **CLOSE**d by looking at the first element of the **CHANNEL** itself. This will always be a **FIX**, and is the 'channel number' assigned by the operating system. A 'channel number' of zero indicates a **CLOSE**d **CHANNEL**.

12.4.4. CHANLIST

```
<CHANLIST>
```

returns a **LIST** whose elements are all the currently open **CHANNEL**s.

12.4.5. INCHAN and OUTCHAN

The channel used by default for input **SUBR**s is the local value of the **ATOM INCHAN**. The channel used by default for output **SUBR**s is the local value of the **ATOM OUTCHAN**.

You can direct I/O to a **CHANNEL** by **SET**ting **INCHAN** or **OUTCHAN** (remembering their old values somewhere), or by giving the **SUBR** you wish to use an argument of **TYPE CHANNEL**. (These actually have the same effect, because **READ** binds **INCHAN** to an explicit argument, and **PRINT** binds **OUTCHAN** similarly.

By the way, a good trick for playing with **INCHAN** and **OUTCHAN** within a function is to use the **ATOM**s **INCHAN** and **OUTCHAN** as **"AUX"** variables, re-binding their local values to the **CHANNEL** you want. When you leave, of course, the old **LVAL**s are restored (which is the whole point).

INCHAN and **OUTCHAN** also have global values, initially the **CHANNEL**s directed at the terminal running MDL. Initially, **INCHAN**'s and **OUTCHAN**'s local and global values are the same. Whenever an error occurs in MDL, the local values of **INCHAN** and **OUTCHAN** are rebound to the global values of the

same **ATOM**s. Unless you live dangerously and change the global values of these **ATOM**s, this will have the effect of redirecting input and output to your terminal, where you are free to go about debugging.

12.5. End-of-File "Routine"

As mentioned above, an explicit **CHANNEL** is the first optional argument of all **SUBR**s used for conversion I/O. The second optional argument for conversion-input **SUBR**s is an "end-of-file routine" -- that is, something for the input **SUBR** to **EVAL** and return, if it reaches the end of the file it is reading. A typical end-of-file argument is a **QUOTE**d **FORM** which applies a function of yours. The value of this argument used by default is a call to **ERROR**. Note: the **CHANNEL** has been **CLOSE**d by the time this argument is evaluated.

End-of-file routines are *not* used with terminal input!

The following **FUNCTION** counts the occurrences of a character in a file, according to its arguments.

```
<DEFINE CHAR-COUNT (CHAR FILE "EXTRA" CHN)
        #DECL ((CHAR) CHARACTER (FILE) STRING
               (CHN) <OR FALSE CHANNEL>)
        <COND (<SET CHN <OPEN "READ" .FILE>>
               <REPEAT ((CNT 0))
                    #DECL ((CNT) FIX)
                    <COND (<==? <READCHR .CHN '<RETURN .CNT>>
                                .CHAR>
                           <SET CNT <+ .CNT 1>>)>>)>>
```

The idea here is that the **FORM** **<RETURN .CNT>** will be **EVAL**uated when the end-of-file is reached. Had **READCHR** been given only one argument, **ERROR** would have been called when end-of-file occurred. Also notice that the *only* way for this **REPEAT** to terminate is from within the call to **READCHR**.

12.6. Additional I/O SUBRs

There are a few other extremely useful I/O routines which should be mentioned here.

12.6.1. READSTRING

READSTRING provides a mechanism of reading characters into a **STRING** until a specified condition is met. This condition may be one of two types:

1. A specified number of characters has been read.

2. One of a specified set of characters has been read.

READSTRING takes a **STRING** which will be filled with **CHARACTER**s read from its second argument, a **CHANNEL**. An optional third argument specifies the condition on which the **READSTRING** will terminate. If the argument is a **FIX**, **FIX CHARACTER**s will be read from **CHANNEL**. If the argument is a **STRING**, **CHARACTER**s will be read until one is a **MEMBER** of the that **STRING** (**MEMQ** really). *In either case* **READSTRING** will terminate when the **STRING** (i.e. the first argument) is filled, should this occur prior to the meeting of the 'stop condition', or if the end-of-file is reached. If there is no third argument, these latter two conditions are the only ways in which **READSTRING** will terminate.

READSTRING returns the number of **CHARACTER**s read at the time of its termination. Here's how to interpret the return from **READSTRING** (it really isn't all that complicated, but it's hard to explain):

- If there was no third argument, the return will either be the length of the **STRING** or a smaller number. If a smaller number, the end-of-file was reached and that smaller number is the number of **CHARACTER**s read before end-of-file was reached.

- If the third argument was a **FIX** and the return was less than that **FIX**, then end-of-file was reached.

- If the third argument was a **STRING**, the return was the number of **CHARACTER**s read before the termination **CHARACTER** was seen. It is very important to realize that the termination **CHARACTER** is *not* read. In other words, it will not be in the **STRING**, and the next time you try to input from **CHANNEL**, that termination **CHARACTER** will be lying in wait. Not taking this into account is the cause of many an error for novice MDLers.

If the terminating event was end-of-file and *another* **READSTRING** is performed, the end-of-file routine will be **EVAL**ed. As with the other non-terminal-directed I/O input routines, the default end-of-file routine is a call to **ERROR**.

This must seem very confusing, but many of your programs will require reading from the terminal and **READSTRING** is by far the best way to do this in MDL. One of the reasons for this is that **READSTRING** will allow the person inputting to your program to edit his input by means of the rubout key and the like. This facility is *very* hard to simulate if you are reading one **CHARACTER** at a time (e.g. with **READCHR**).

A very important warning regarding **READSTRING**: MDL, you will recall, only starts processing terminal input *after* an escape is typed. This is true also for calls to **READSTRING**. This means that you cannot expect to get a line of input from a user by doing a **READSTRING** with a 'stop condition' of a carriage-return or a line-feed unless this is followed by an escape. There are ways around this 'feature', but they are beyond the scope of this primer. Please consult the manual or a seasoned MDLer for help.

Here's a skeleton of a calculator program:

```
<DEFINE CALC (.. "AUX" .. CNT (BUFFER <ISTRING 100 !\ >) ..)
        #DECL (... (CNT) FIX (BUFFER) STRING ...)
        <REPEAT ()
             <PRINC "
>">
             <SET CNT <READSTRING .BUFFER
                                  .INCHAN
                                  <STRING <ASCII 27>>>>
             <READCHR .INCHAN>
             <COND (<0? .CNT>
                    <PRINC " Thanks for using the calculator!">
                    <RETURN>)
                   (<==? .CNT 100>
                    <PRINC "
Sorry, that one's too big for me.  Please try
something a bit easier, like 2 + 2.">)
                   (T <CALCULATE .BUFFER .CNT>)>>>
```

This calculator program is essentially a large **REPEAT** loop, as you might expect. Each time through, it starts by printing a prompt (a carriage-return followed by a closing angle-bracket). It then reads some input from the terminal into a **STRING** which is initialized at the start of the **FUNCTION** as having a **LENGTH** of **100**. The stop-condition is the presence of an escape (27 decimal in ASCII). Since **READSTRING** *will not read* the terminating escape, a **READCHR** is performed. If one were to check on what the **READCHR** was returning, one would find it to always be an escape. If the value of the call to **READSTRING** is zero, no input was typed before the escape. In this example, the program terminates. If the return from **READSTRING** were **100**, then the person typing to the program has given an excessively long input (this is only true in this example; the **FUNCTION** could have been written to accept much longer inputs) and he is told this. Otherwise, the *function* **CALCULATE** is called with the user's input (**.BUFFER**) and the number of **CHARACTER**s that the user typed (**.CNT**, i.e. the number of 'valid' **CHARACTER**s in **.BUFFER**). With any luck, **CALCULATE** will do something useful with its arguments, like performing the requested calculation and printing the result(s).

Could you write this skeleton without using a **REPEAT**? There are at least two other reasonable ways. If you cannot, try rereading Chapter 10.

12.6.2. PRINTSTRING

The **SUBR PRINTSTRING** is analogous to the **SUBR READSTRING**. It takes three arguments, a **STRING** (the **STRING** to print), a **CHANNEL** (on which to print it), and a **FIX** (the number of characters from the **STRING** to print). If the **LENGTH** of the **STRING** is less than the third argument, **PRINTSTRING** just prints the **STRING**. In any event, **PRINTSTRING** returns the number of characters actually printed.

12.7. SAVE Files

The entire state of MDL can be saved away in a file for later restoration: this is done with the **SUBRs** **SAVE** and **RESTORE**. This is a very different form of I/O from any mentioned up to now; the file used contains an actual image of your MDL address space and is not, in general, "legible" to other MDL routines. **RESTORE**ing a **SAVE** file is much faster than re-**READ**ing the objects it contains.

12.7.1. SAVE

Calling the **SUBR** **SAVE** with a file-name will save away the entire state of your MDL in a file with that name. It then returns **"SAVED"**. When a **RESTORE** is done later (to return to the 'saved' state), the call to **SAVE** returns **"RESTORED"**.

```
<DEFINE SAVE-IT ("OPTIONAL"
                  (FILE "<GUEST>PUBLIC.SAVE")
                  "AUX" (SNM ""))
        <SETUP>
        <COND (<=? "SAVED" <SAVE .FILE>>
                <CLEANUP>
                "Saved.")
               (T
                <PRINC "
Amazing program at your service.">
                <START-RUNNING>)>>
```

12.7.2. RESTORE

RESTORE, given a file-name, completely replaces the contents of the MDL from that file, including the state of execution existing when the **SAVE** was done and the state of all open I/O **CHANNEL**s. If a file which was open when the **SAVE** was done does not exist when the **RESTORE** is done, a message to that effect will appear on the terminal.

A **RESTORE** never returns (unless it gets an error): it causes a **SAVE** done some time ago to return again (this time with the value **"RESTORED"**), even if the **SAVE** was done in the midst of running a program. In the latter case, the program will continue its execution upon **RESTORE**ation.

12.8. PARSE, LPARSE, and UNPARSE

These **SUBR**s are borderline I/O routines. **PARSE**, given a **STRING**, uses **READ**s algorithm for converting text into MDL objects and returns the first one found.

```
<SET STR "(FOO 1 2.3) HO-HUM">$
"(FOO 1 2.3) HO-HUM"
<PARSE .STR>$
(FOO 1 2.3)
```

LPARSE, given a STRING, returns a LIST containing all of the items which READ would have found in the STRING. Using the same example:

```
<LPARSE .STR>$
((FOO 1 2.3) HO-HUM)
```

UNPARSE is the inverse of PARSE. Given a MDL object, UNPARSE returns a STRING, suitable for MDL PRINTing.

```
<UNPARSE (A B C)>$
"(A B C)"
<UNPARSE 3.4>
"3.40000000"
```

All of these SUBRs are very expensive CPU-wise. They should be avoided if at all possible.

12.9. Other I/O Functions

12.9.1. FLOAD

FLOAD, given a file-name, READs and EVALuates everything in the file, in order, and returns "DONE". If the file specified does not exist, FLOAD returns a FALSE containing the reason why.

12.9.2. SNAME

<SNAME *string*> is identical in effect with <SETG SNM *string*>, that is, it causes *string* to become the *dir* argument used by default by all SUBRs which want file specifications (in the absence of a local value for SNM). SNAME returns its argument.

<SNAME> is identical in effect with <GVAL SNM>, that is, it returns the current *dir* used by default.

12.9.3. FILE-LENGTH

FILE-LENGTH, given a **CHANNEL** open for input, returns the length in characters of the file associated with that **CHANNEL**. Doing a **FILE-LENGTH** on an terminal **CHANNEL** is silly.

12.9.4. RESET

```
<RESET channel>
```

returns *channel*, after "resetting" it. Resetting a **CHANNEL** is like **OPEN**ing it afresh, with only the file-name slots preserved. For an input **CHANNEL**, this means emptying all input buffers and, if it is a **CHANNEL** to a file, doing an **ACCESS** to **0** on it. For an output **CHANNEL**, this means returning to the beginning of the file -- which implies, if the mode is not **"PRINTO"**, destroying any output done to it so far. If the opening fails (for example, if the mode slot of *channel* says input, and if the file specified in its real-name slots does not exist), **RESET** (like **OPEN**) returns **#FALSE (***reason:string file-spec:string status:fix***)**.

12.9.5. RENAME

RENAME is for renaming and deleting files. It takes two kinds of arguments:

- (a) two file names, separated by the **ATOM TO**

- (b) one file name

Omitted file-name parts use the same values by default as does **OPEN**. If the operation is successful, **RENAME** returns T, otherwise **#FALSE (***reason:string status:fix***)**.

In case (a) the file specified by the first argument is renamed to the second argument. For example:

```
<RENAME "FOO" TO "BAR">          ;"Rename FOO.MUD to BAR.MUD."
```

In case (b) the single file name specifies a file to be deleted. For example:

```
<RENAME "<MARC>FOO.MUD">  ;"Delete file FOO.MUD
                                  from MARC's directory."
```

12.10. Terminal CHANNELs

MDL automatically adds a line-feed, whenever a carriage-return is input from a terminal **CHANNEL**. In order to type in a lone carriage-return, a carriage-return followed by a rubout must be input. **PRINT**, **PRIN1** and **PRINC** do not automatically add a line-feed when a carriage-return is output. This enables overstriking on a terminal that lacks backspacing capability. It also means that what goes on a terminal and what goes in a file are more likely to look the same.

12.10.1. TYI

TYI, given a terminal input channel, returns one **CHARACTER** from it *when it is typed*, rather than after **$** (ESC) is typed, as is the case with **READCHR**. Novice MDLers tend to use **TYI** to read input from the terminal. This is not recommended as a rule. Use **READSTRING** instead.

13. Making Tables

It seems that MDL programmers are always making tables of one thing or another. Someone's Whois program may want a table relating a person's login name to his full name. Someone's Calculator program may want a table to associate arbitrary variable names with values.

There are any number of ways to implement tables for these types of purposes; some of these may already have come to mind. For our discussion, let's use the example of a calculator program which accepts inputs of the form:

```
A = 4 + 3
B = (A * A) + 7
```

Without considering the actual details of how the calculator might be written, something must be done to keep track of the fact that the variable **A** has a value of 7, and that **B** has a value of 56. What follows are a number of different approaches to solving this sort of problem. Each should be examined carefully and the advantages and disadvantages noted.

13.1. Use a LIST

This is the most common and possibly the most useful approach. For the given example, we can create a **LIST**, which, for example, is the **GVAL** of the **ATOM VARIABLES**.

```
,VARIABLES$
(A 7 B 56)
```

If one wants to add a new variable, say **C**, with a value, say 100, one can do the following:

```
<SETG VARIABLES (C 100 !,VARIABLES)>
```

To check if a variable, say **D**, has a value, one can do this:

```
<MEMQ D ,VARIABLES>
```

To actually get D's most recent value,

```
<COND (<SET M <MEMQ D ,VARIABLES>>
       <2 .M>)>
```

This **COND** clause returns a **FALSE** if **D** doesn't have a value; otherwise, it returns the value.

Removing variables from the **LIST** can be done with **PUTREST**. As an exercise, write a **FUNCTION** which, given a variable name and a **LIST** (like the one we used above), removes the variable and its value from the **LIST**. Be sure you handle the case in which the variable isn't in the **LIST**. One solution to this is given at the end of the chapter. Don't peek, and don't be too frustrated. The **FUNCTION** isn't *that* easy to write.

LISTs are very space-efficient. However, while **LIST**s are practical for smallish tables, larger ones will tend to become very slow to access, since **LIST**s are not random-access structures (see Chapter 7). If your table needs to be more than a hundred elements long, you should probably try something else.

13.2. Use a VECTOR

Think twice before you do. As we saw in Chapter 7, **VECTOR**s have the property that they cannot be added to and cannot have elements removed without creating an entirely new structure (which is very garbage-creating). Therefore, using **VECTOR**s is not a good idea unless the table is 'pre-formed' and elements need never be removed or added. If you have a table of ordered elements of relatively fixed size, use of **VECTOR**s with some sort of binary-searching algorithm is appropriate. For the calculator example, don't use a **VECTOR**.

13.3. Use an ATOM

Another simple approach would be to **SETG** the variable name (which is an **ATOM**) to its value. Then, you can use **GASSIGNED?** to check if it has a value, **GVAL** to get it, and **GUNASSIGN** to remove it. Lookup using **GVAL**s is moderately fast, but there is a problem. Imagine the result of your poor calculator user setting a variable whose name is the name of your program. The use of **SET** and **LVAL** is also perilous.

13.4. Use an Association

MDL allows you to assign a value to a *pair* of MDL objects. This can be done using the **SUBR PUTPROP**. The value of such an 'association' can be retrieved with the **SUBR GETPROP**.

```
<PUTPROP MICHAEL AGE 28>$
MICHAEL
<GETPROP MICHAEL AGE>$
28
```

One can associate *any* three MDL objects using **PUTPROP**. A useless, but legal, use might be:

```
<PUTPROP [1 2 3] (4 5 6) "FOOBAR">$
[1 2 3]
<GETPROP [1 2 3] (4 5 6)>$
#FALSE ()
```

Why did the last **GETPROP** return **#FALSE ()**? Hint: Are either of the arguments to **GETPROP ==?** to the arguments to **PUTPROP**?

By giving **PUTPROP** only two arguments, it returns what **GETPROP** would have returned, and then *removes* the association.

```
<PUTPROP MICHAEL AGE>$
28
<GETPROP MICHAEL AGE>$
#FALSE ()
```

In the calculator example, one could do something like this:

```
<PUTPROP A VARIABLE 7>$
A
<PUTPROP B VARIABLE 56>$
B
```

to set the variables' value. One would retrieve the values like this:

```
<GETPROP A VARIABLE>$
7
<GETPROP B VARIABLE>$
56
```

Associations are fast (they use a hashing scheme with a fixed number of buckets), but rather

space-inefficient. Large numbers of them will tend to crowd your core-image.

13.4.1. Hashing

Hashing, for those unfamiliar with the notion, is an algorithm for table lookup which is based on a 'directed search'. A hash table can be thought of as a VECTOR of LISTs. These LISTs are commonly called 'buckets'. Each actual item in the hash table is found in one of these 'buckets'. What makes hash lookup fast is that there is a simple algorithm for determining which 'bucket' an item is in. Once that determination is made, the 'bucket' is searched linearly for the item. Thus, a hash table of length 100, which contains 1000 items, would have an average of 10 items per 'bucket'. Thus, the access time for looking up an item would be the same as that for MEMQ'ing a LIST of 10 elements plus the small overhead of determining which 'bucket' the item is in. This is obviously much faster than linearly searching a LIST of 1000 elements, by a factor approaching 100.

13.5. Use an OBLIST

OBLISTs are tables of ATOMs which are hashed in such a way that finding an ATOM in one is very fast. Similarly, inserting and removing ATOMs is simple.

To create an OBLIST of your own, use the SUBR MOBLIST (Make OBLIST), which takes a name (ATOM) and the number of hash buckets for the OBLIST (defaultly 17). For best results, the number of buckets should be prime.

```
<SETG FOOBAR <MOBLIST FOOBAR 7>>$
#OBLIST ![() () () () () () ()]
```

Note that there are seven empty LISTs in the OBLIST -- you guessed it ... each LIST is a bucket!

To insert an ATOM into an OBLIST, use the SUBR INSERT. To remove an ATOM from an OBLIST, use the SUBR REMOVE. To look up an ATOM in an OBLIST, use the SUBR LOOKUP. Each of these three SUBRs takes a STRING, the PNAME of the ATOM, and an OBLIST.

```
<INSERT "MIKE" .FOOBAR>$
MIKE!-FOOBAR
.FOOBAR$
#OBLIST ![() () () () (MIKE!-FOOBAR) () ()]
<LOOKUP "MIKE" .FOOBAR>$
MIKE!-FOOBAR
<LOOKUP "BLETCH" .FOOBAR>$
#FALSE ()
```

There's something new here, namely the suffix to the name of the **ATOM**: an exclamation point, a hyphen, and an **ATOM**. This suffix is called an 'oblist-trailer' or simply a 'trailer'. It is there so that **READ** and **PRINT** can distinguish this new **ATOM** whose **PNAME** is **FOOBAR** from an **ATOM** on another **OBLIST** whose **PNAME** is **FOOBAR**. Therefore, to directly reference the **ATOM** of **PNAME BLETCH** in the **FOOBAR OBLIST**, one must type in the following:

```
BLETCH!-FOOBAR
```

In fact, typing **BLETCH!-FOOBAR** *causes* **READ** to create an **ATOM** with **PNAME BLETCH** in the **FOOBAR OBLIST** if none already existed. Not only that, but typing **FROB!-MUMBLE** causes **READ** to create an **ATOM** of **PNAME FROB** in the **MUMBLE OBLIST** (*creating a* **MUMBLE OBLIST** *if necessary*) if none already existed.

If you are interested in a more complete description of **OBLIST**s, refer to the next section. To continue with the calculator example, we might start by creating an **OBLIST** for variables.

```
<SETG VARIABLES <MOBLIST VARIABLES>>$
#OBLIST ....
```

Then, we assign values to **A** and **B** as follows:

```
<DEFINE SET-VARIABLE (NAM VAL "AUX" (PNM <PNAME .NAM>))
    #DECL ((NAM) ATOM (VAL) ANY (PNM) STRING)
    <SETG <COND (<LOOKUP .PNM ,VARIABLES>)
                (T <INSERT .PNM ,VARIABLES>)>
          .VAL>>$
SET-VARIABLE
<SET-VARIABLE A 7>$
7
<SET-VARIABLE B 56>$
56
```

To retrieve values, we might do this:

```
<DEFINE GET-VARIABLE (NAM "AUX" ATM)
    #DECL ((NAM) ATOM (ATM) <OR FALSE ATOM>)
    <COND (<SET ATM <LOOKUP <PNAME .NAM> ,VARIABLES>>
           ,.ATM)>>$
GET-VARIABLE
<GET-VARIABLE B>$
56
<GET-VARIABLE D>$
#FALSE ()
```

Using **OBLIST**s in this way solves the problem mentioned earlier regarding the use of **ATOM**s: that of

variable conflicts. By using ATOMs in your own private OBLIST, there is no danger of mistakenly changing the value of someone else's (or your own...) ATOM. Now, your calculator user can use variable names which are the same as those of your calculator FUNCTIONs without fear of disaster.

To summarize, using OBLISTs is fast. ATOMs are rather large; about the same size as an association. Whereas the hashing table for associations is a fixed size, the hashing table for an OBLIST can be determined when the OBLIST is created. ATOMs are more versatile than associations, and can be used in more ways. Good MDL programmers, given the choice, will use OBLISTs over associations.

13.6. OBLISTs, READ, and PRINT

It was stated in section 4.1 that typing GEORGE to MDL caused READ to "look up the representation [of GEORGE]" in a "table it keeps for such purposes...." It should now be clear that the "table it keeps" is, in fact, an OBLIST, and that it "looks up the representation" by using the SUBR LOOKUP. You are now ready to understand what, in fact, READ does.

When READ encounters something that it determines must be an ATOM (i.e. it can't be anything else), it does LOOKUPs of the PNAME sequentially in all of the OBLISTs in .OBLIST (i.e. the LVAL of the ATOM OBLIST). [MDL sets up .OBLIST to be a LIST of OBLISTs. Initially, .OBLIST has two OBLISTs in it: the INITIAL OBLIST (user ATOMs) and the ROOT OBLIST (MDL's ATOMs). The ATOMs which point to the F/SUBRs all live in ROOT.] The value of the first LOOKUP to succeed becomes the value of the call to READ. If the PNAME isn't found, an ATOM with that PNAME is INSERTed into <1 .OBLIST>.

If READ (of ATOMs) were written in MDL, it might look like this:

```
<DEFINE READ-ATOM (STR)
    #DECL ((STR) STRING)
    <COND (<MAPF <>
                 <FUNCTION (OBL "AUX" ATM)
                     #DECL ((OBL) OBLIST (ATM) <OR FALSE ATOM>)
                     <COND (<SET ATM <LOOKUP .STR .OBL>>
                            <MAPLEAVE .ATM>)>>
                 .OBLIST>)
          (T <INSERT .STR <1 .OBLIST>>)>>
```

However, if an explicit trailer is given, the ATOM is placed in the OBLIST named in the trailer. Trailers may be 'recursive'. For example, A!-B!-C!-D!-E is an ATOM with PNAME A which is in an OBLIST whose name is an ATOM with PNAME B which is on an OBLIST whose name.... The ATOM with PNAME E will reside in one of the OBLISTs in .OBLIST. When PRINT attempts to print an ATOM of this kind, it prints trailers until one of the OBLIST names can be found on an OBLIST in .OBLIST.

14. Debugging MDL Programs - An Introduction

If you have ever written a program which works completely correctly on the first attempt, you most likely have benefited from divine intervention. In the more likely event that one of your MDL programs is "buggy", you will see messages which look like this:

> ***ERROR***
> *reason*
> *information-about-error*
> *function-which-generated-it*
> **LISTENING-AT-LEVEL** *n* **PROCESS** *m*

The *information-about-error* may be one or more than one object. The *n* is an indication of how many levels of errors have occurred, and *m* should be completely ignored. If you ever see a number other than 1 in that position, you probably don't need to be reading this.

The meaning of this gobbledygook is that the MDL **SUBR** **ERROR** was invoked. This may have happened from an explicit call to **ERROR**, as in the following:

```
<ERROR YOU-LOSE BECAUSE MY-FUNCTION>
```

More likely, however, the MDL interpreter discovered an error in your program, such as a variable without a value, or a bad argument to a *function*, and called **ERROR** internally. The effect is the same: ***ERROR*** is printed, followed by all of the arguments to **ERROR**, and MDL starts **LISTEN**ing at the next higher level. In other words, **LISTEN** has been called recursively.

In the remainder of this chapter, we will be discussing the debugging of a particularly trivial error in a sample **FUNCTION**. Please refer frequently to the figure at the end of the chapter in which parts of the example are diagrammed and commented.

In order to correct an error, it is necessary to have some information about the history of MDL's execution at the time of the error. To do this, the *function* **FR&** is called, usually without arguments. Let us assume that the following **FUNCTION** is being called as follows:

```
<DEFINE GT10 (ARG) <G? .AGR 10>>$
GT10
<GT10 11>$
*ERROR*
UNBOUND-VARIABLE
AGR
LVAL
LISTENING-AT-LEVEL 2 PROCESS 1

<FR&>$

0  ERROR        [UNBOUND-VARIABLE!-ERRORS AGR LVAL]
1  LVAL         [AGR]
2  EVAL         [.AGR]
3  EVAL         [<G? .AGR 10>]
4  EVAL         [<GT10 11>]
5  LISTEN       []
TOPLEVEL
```

What is shown here, one to a line, are the **FRAME**s which have been generated by MDL, starting from the one called **LISTEN**, which is where MDL was waiting when the **FORM** **<GT10 11>** was input. The lines above this one are the steps which MDL took until the error occurred, namely in the code for **LVAL**. Each line has a number, by which the **FRAME** can be identified, the **FUNCT** of the **FRAME** (always an **ATOM**), and the **ARGS** of the **FRAME** (always a **TUPLE**). For these purposes, a **TUPLE** can be considered to be a **VECTOR**. Given a **FRAME** number, the **FRAME** can be referenced by invoking the *function* FRM, as follows:

```
<SET F <FRM 3>>$
#FRAME EVAL
<FUNCT .F>$
EVAL
<ARGS .F>$
[<G? .AGR 10>]
```

Having gotten this far, it has become obvious that the problem is that the function **GT10** is incorrect, in that the reference to **AGR** was intended to be a reference to **ARG**. What follows are some ways of fixing the problem, all of which will work. Although this is a trivial example of an error, as the problem itself was easy to spot, the methods of error recovery are always the same!

14.1. Method 1: Start Over

Edit the **FUNCTION** with your favorite text editor and reload it, retype it in to MDL directly, or whatever. Then invoke the **SUBR** **ERRET** with no arguments. This will cause MDL to return to its "top

level", i.e. **LISTENING-AT-LEVEL 1**. All parts of the execution in progress including all **ATOMic** bindings (except those made at "top level") will be lost.

```
<ERRET>$

LISTENING-AT-LEVEL 1 PROCESS 1
```

14.2. Method 2: Forcing FRAMEs to Return Values

It is possible to cause MDL to force an arbitrary **FRAME** to return an arbitrary value and to continue execution from that point. This is done by calling **ERRET** with either one or two arguments. The first argument is the value for the **FRAME** to return, and the second, if given, is the **FRAME** which is to return that value. If no second argument is given, the **FRAME** immediately previous to the **ERROR FRAME** will be used as a default.

```
<ERRET 9>$
#FALSE ()
```

What happened was that the **LVAL FRAME** (i.e. **<FRM 1>** was caused to return 9. Execution continued, such that the **EVAL FRAME** above (i.e. **<FRM 3>**) also returned 9, and the next frame evaluated **<G? 9 10>**, which returned an empty **FALSE**, which became the value of the call to **GT10**. Notice that in this case, the fact that **11** was originally passed to **GT10** has become unimportant. Another way of doing the same thing would have been to say

```
<ERRET 9 <FRM 1>>$
#FALSE () or
<ERRET 9 <FRM 2>>$
#FALSE ()
```

However, saying

```
<ERRET 9 <FRM 3>>$
9
```

has a different result. What happened was that the **FORM <G? .AGR 10>** was forced to return 9. Since that **FORM** was the last in the body of the **FUNCTION**, the result of its evaluation became the result of the evaluation of the **FUNCTION**. Therefore, **GT10** returned 9.

14.3. Method 3: Use EDIT to Repair your FUNCTIONs

In the last method, nothing has been done to correct the real problem, i.e. that the program has a bug in it. One way to solve this is to use the MDL editor, a *function* called **EDIT** to alter the program itself. **EDIT** is usually invoked with the name of a **FUNCTION** to be edited as the only argument. You will now be "talking" to the MDL editor. Commands to the editor should be terminated with an escape, and are one or two characters followed by some arguments, which are usually optional. **EDIT** will display after each command your current "location" in the **FUNCTION** you are editing. To move around, the commands L (left), R (right), U (up), and D (down) are used. These may be followed by a numerical argument, the number of times to perform the command. The arguments must be preceded by a space. A vertical bar is used here to indicate your "position" in the edited **FUNCTION**. In the real MDL editor, the "position" may be indicated by some other characters.

```
<EDIT GT10>$
#FUNCTION (| (ARG) <G? .AGR 10>)
R 2$
#FUNCTION ((ARG) <G? .AGR 10> |)
L$
#FUNCTION ((ARG) | <G? .AGR 10>)

D$
<| G? .AGR 10>
D$
ERROR, YOU CAN'T GO DOWN
<| G? .AGR 10>
R$
<G? | .AGR 10>)
```

To alter the **FUNCTION** the following commands may be used: I (insert), K (kill), and C (change). Insert takes any number of objects as arguments and inserts them all to the right of your "location". Kill takes an optional number (default 1) and removes that many objects from the right of your "location". Change takes one argument and changes the object to the right of your "location" to it.

```
<G? | .AGR 10>
C .ARG$
<G? | .ARG 10>
```

This has had the effect of fixing the error in the program. To exit the editor, use the Q command.

```
<G? | .ARG 10>
Q$T
```

The T was the returned value of the call to **EDIT**. Now, a look at the **FRAMEs** using FR& shows the following:

```
<FR&>$

0  ERROR       [UNBOUND-VARIABLE!-ERRORS AGR LVAL]
1  LVAL        [AGR]
2  EVAL        [.AGR]
3  EVAL        [<G? .ARG 10>]
4  EVAL        [<GT10 11>]
5  LISTEN      []
TOPLEVEL
```

Make sure you understand what has happened. The way the editor works for the case of the C command is to **PUT** the argument to C into the *structure*. The **FORM**s contained in the **ARGS**s of **FRAME**s are simply pointers directly into the **FUNCTION** being executed. Thus, the **PUT** into the **FORM** will change the argument to the **FRAME** which points to it! This is extremely important, and is illustrated in the diagram at the end of the chapter. Think about this very carefully if you don't understand this, and then be sure you convince yourself of why the argument to **<FRM 2>** has *not* changed.

Now that you have done this, it would be useful if you could tell MDL to go back and retry **<FRM 3>**. In fact you can, using the **SUBR RETRY** which takes a **FRAME** as an argument, and simply pretends that nothing past that point in execution has ever happened. This works completely as long as the execution below that point hasn't had any side-effects. A MDL *function* is said to have side-effects if it does anything other than manipulate its local variables and return a value. Stated another way, a *function* with *no* side-effects is a black-box with an input (arguments) and an output (value), but no effect on the 'outside world'. The most blatant side-effecting **SUBR**s are **PUT**, **PUTREST**, **SETG**, and **PRINT**. **SET**ting **ATOM**s which are not bound in a currently executing **FUNCTION** also has side-effects. In a purist structured-programming sense, no *function* should have side-effects (with the obvious exception of printing output). However, there are certainly cases in which **PUT**, **PUTREST**, and **SETG** are tremendously useful, if not vital. Care should be taken, however, since many bugs can be traced to one *function*'s causing a side-effect which causes another *function* to fail.

```
<RETRY <FRM 3>>$
T
```

Question: Would

```
<RETRY <FRM 4>>$
```

have the same effect? The answer is yes, because you are restarting from an earlier level of execution. What would be the effect of **RETRY**ing **<FRM 5>**? Hint: It isn't a return of T. What would be the effect of **RETRY**ing **<FRM 2>**? Hint: It isn't good. If you aren't completely sure of the answer to these, try it in MDL.

14.4. Method 4: Altering FRAMEs / RETRY

Let's try the following, starting from the point of the error:

```
<SET X <1 <ARGS <FRM 3>>>>$
<G? .AGR 10>
<PUT .X 2 '.ARG>$          ;"Why the quote?"
<G? .ARG 10>
<RETRY <FRM 3>>$
T
```

We have done the same thing as we did using method 3, but from a different angle. Question: What does the **FUNCTION GT10** look like now? Hint: Not the same. If you don't see this, you didn't understand why editing the **FUNCTION** worked either.

14.5. Summary

We have presented four different ways of handling errors in MDL. This list is not exhaustive, but it should provide enough background to enable you to handle most situations. If this chapter has been totally confusing, ask someone for help and use method 1 in the meantime. Notice that method 2 does *not* prevent the same error from recurring: it merely corrects the current instance of that error. Methods 3 and 4 correct the general problem and the current instance of the error generated by the problem. However, even though the **FUNCTION** is changed in your MDL, you still must alter it using your favorite editor at a later time (or write out an updated copy of the file directly from MDL). The changes made while in MDL are *not* reflected in your files! They will, however, allow you to proceed without moving back and forth constantly between MDL and your editor.

MDL has many other debugging aids including breakpoints (in **EDIT**), tracing, monitoring the values of local and global variables, and more. For a detailed description of these facilities, consult *The MDL Programming Environment* [Lebling 80].

Selected **FRAME**s during execution of **GT10** as described in the text. Note that the **FRAME**s point directly at the structure of **GT10** (*e.g.* the **FORM** in **FRM3** is **==?** to the second element of the **FUNCTION**).

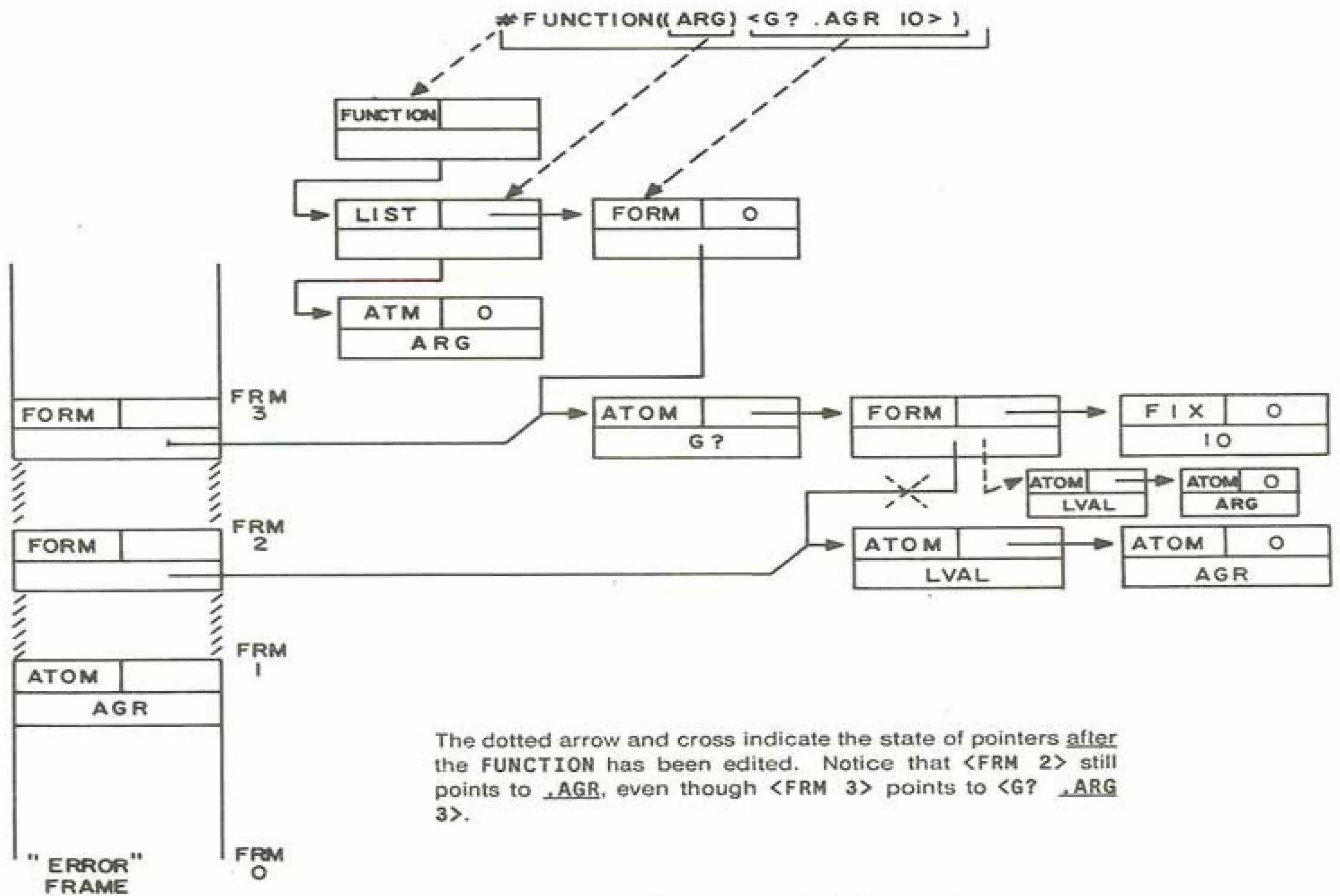


Figure 14-1: Diagram for the example in this chapter

References

[Galley 79]
S. W. Galley and Greg Pfister.
The MDL Programming Language.
M.I.T. Laboratory for Computer Science, 1979.

[Lebling 77]
P. David Lebling, R. V. Baron and Bruce K. Daniels.
RMODE: A Real-time Edit Facility.
Technical Report SYS.04.07-1, MIT LCS Programming Technology Division, October, 1977.

[Lebling 80]
P. David Lebling.
The MDL Programming Environment.
M.I.T. Laboratory for Computer Science, 1980.

[Pfister 72]
Greg Pfister.
A Muddle Primer.
Technical Report SYS.11.01, MIT Project Mac DM/CGS, May, 1972.

[Stallman 79]
Richard M. Stallman.
EMACS.
MIT AI Laboratory, 1979.

[Weinreb 78]
Daniel Weinreb and David Moon.
Lisp Machine Manual.
MIT AI Laboratory, 1978.

Index

! 54
!$ 8
!- 113
![58
!\ 59, 99
!] 58

" 11, 99

12, 37, 99

$ 7, 98, 108

' 36, 63, 87

(11

) 11

* 24

+ 24

, 21

- 24

. 22

/ 24

0? 25

1? 25

< 11

==? 25, 41
=? 41

> 11

ABS 24
ACTIVATION 78
AGAIN 77
AND 69, 72, 90
AND? 70
"ARGS" 89
ASCII 60
ASSIGNED? 26, 88
ATAN 24, 31
ATOM 19, 35, 99
"AUX" 90, 101

BACK 57, 66
Binding 29, 77

CHANLIST 101
CHANNEL 100, 101
CHARACTER 35, 59, 61, 99
CHTYPE 38
CLOSE 101
, 21
COND 70
COS 24, 31
CRLF 99

DECL 91, 93
DEFINE 89
. 22

EDIT 118
EMPTY? 43, 66
=? 41
==? 25, 41
Equality 41
ERRET 9, 116, 117
ERROR 9, 115
EVAL 15, 23, 45, 65, 87, 89
! 54
!\ 59
!- 113
!$ 8
![58
!] 58
EXP 24, 31

False 14, 25
FALSE 35, 37, 53
FILE-EXISTS? 101
FILE-LENGTH 107
FIX 11, 16, 17, 24, 26, 35
FLOAD 34, 106
FLOAT 11, 16, 17, 24, 26, 35
FORM 12, 35

FR& 9, 115
FRAME 116
FRAMES 9
Free variables 30, 90
FSUBR 23, 69, 70, 89
FUNCTION 27, 79

G=? 25
G? 25
Garbage 63
GASSIGNED? 26, 110
GETPROP 111
GUNASSIGN 110
GVAL 21, 31

Hashing 112

ILIST 62
INCHAN 101
INIT 34
INSERT 112
ISTRING 62
Iteration 86
ITS 8
IVECTOR 62

L=? 25
L? 25
LENGTH 43, 66
LENGTH? 51, 66
Lisp 1
LIST 12, 35, 41, 44, 66
LISTEN 115
LOG 24, 31
LOOKUP 112
LPARSE 106
LVAL 22, 29

MAPF 79, 84
MAPLEAVE 80
MAPR 83, 84
MAPRET 81
MAPSTOP 82
MAX 24
MDL 1
MEMBER 62, 66
MEMQ 62, 66
MIN 24
MOBLIST 112
MOD 24
MUDDLE 34
"MUDDLE.INIT" 34

N==? 26
"NAME" 78
NEWTYPE 38, 65, 93
NEXTCHR 98
NOT 69
NTH 45, 52, 66

OBLIST 99, 112, 114
OPEN 100, 107
OPTIONAL 88
OR 70, 72
OR? 70
OUTCHAN 101

PARSE 105
. 22
PNAME 19, 114
PPRINT 33
Predicates 25
Prefix Notation 13
Pretty Printing 33
PRIMTYPE 35, 41, 65
PRIN1 99, 108
PRINC 99, 108
PRINT 15, 17, 99, 108
PRINTSTRING 104
PROG 77
PUT 45, 52, 66
PUTPROP 111
PUTREST 47, 66

' 36, 63, 87

RANDOM 25
READ 15, 16, 98, 114
READCHR 98, 108
READSTRING 103
Recursion 74, 86
REMOVE 112
RENAME 107
REPEAT 78
RESET 107
REST 45, 66
RESTORE 105
RETRY 119
RETURN 77, 78
Rubout 8, 98

SAVE 105
SEGMENT 53, 66
SET 21, 29
SETG 20, 29
SIN 24, 31
SNAME 106
SNM 106
SQRT 24, 30
STRING 35, 59, 66, 99
SUBR 23
Subroutine 23

T 25
Tenex 8, 34
TO 107
TOP 57, 66
Tops-20 8, 34
Trailer 113
True 25

Truth 14
"TUPLE" 88
TYI 108
TYPE 11, 15, 35
TYPE? 36
TYPEPRIM 38

UNPARSE 106
UTYPE 59
UVECTOR 58, 66

VECTOR 12, 35, 54, 66

WORD 35

Zork 74

[11

\ 99

] 11

↑@ 8, 98
↑D 8, 98
↑G 8
↑L 8, 98
↑O 8
↑S 8, 51

{ 11

} 11