# MDL

Primer and Manual

(for versions 54 and 104)

S. W. Galley and Greg Pfister

Laboratory for Computer Science

Massachusetts Institute of Technology

Cambridge   Massachusetts 02139

Abstract

MDL began existence in late 1970 (under the name Muddle) as a successor to Lisp (ref 6), a candidate vehicle for the Dynamic Modeling System, and a possible base for implementation of Planner (ref 5). The original design goals included an interactive integrated environment for programming, debugging, loading, and editing; ease in learning and use; facilities for structured, modular, shared programs; extensibility of syntax, data types and operators; data-type checking for debugging and optional data-type declarations for compiled efficiency; associative storage, multiprocessing, and graphics. Along the way to reaching those goals, it developed flexible input/output (including the ARPA Network), and flexible interrupt and signal handling. It now serves as a base for software prototyping, research, development, education, and implementation of the majority of programs at MIT-DMS: a library of sharable modules, a coherent user interface, special research projects, autonomous daemons, etc.

This document was originally intended to be a simple low-level introduction to MDL. It has, however, acquired a case of elephantiasis and now amounts to a discursive description of the whole interpreter, as realized in <u>MDL version number 54</u> (ITS) or <u>104</u> (TENEX). A low-level introduction may still be had by restricting one's attention to specially-marked sections only. The scope of the document is confined as much as possible to the interpreter itself. Other adjuncts (compiler, assembler, preloaded user programs, library) are mentioned as little as possible, despite their value in promoting the language seen by a user from "basic survival" to "comfortable living". Indeed, MDL could not fulfill the above design goals without the compiler, assembler, structure editor, control-stack printer, context printer, pretty-printer, dynamic loader, and library system -- all of which are not part of the interpreter but programs written in MDL and symbiotic with one another. Further information on these adjuncts can be found in the references.

Acknowledgements

I was not a member of the original group which labored for two years in the design and initial implementation of Muddle; that group was composed principally of Gerald Sussman, Carl Hewitt, Chris Reeve, Dave Cressey, and later Bruce Daniels. I would therefore like to take this opportunity to thank my Muddle mentors, chiefly Chris Reeve and Bruce Daniels, for remaining civil through several months of verbal badgering. I believe that I learned more than "just another programming language" in learning Muddle, and I am grateful for this opportunity to pass on some of that knowledge. What I cannot pass on is the knowledge gained by using Muddle as a system; that I can only ask you to share.

For editing the content of this document and correcting some misconceptions, I would like to thank Chris Reeve, Bruce Daniels and especially Gerald Sussman, one of whose good ideas I finally did use.

<div align="right">

Greg Pfister
December 15, 1972

</div>

Since Greg left the fold, I have taken up the banner and updated his document. The main sources for small revisions have been the on-line file of changes to MDL, for which credit goes to Neal Ryan as well as Reeve and Daniels, and the set of on-line abstracts for interpreter Subroutines, contributed by unnamed members of the Programming Technology Division. Some new sections were written almost entirely by others: Dave Lebling wrote chapter 14 and appendix 3, Jim Michener section 14.3, Reeve chapter 19 and appendix 1, Daniels and Reeve appendix 2, Brian Berkowitz section 22.7, Tak To section 17.2.2, and Ryan section 17.1.3. Sue Pitkin did the tedious task of marking phrases in the manuscript for indexing. Pitts Jarvis and Jack Haverty advised on the use of PUB and the XGP. Many PTD people commented helpfully on a draft version.

My task has been to impose some uniformity and structure on these diverse sources (so that the result sounds less like a dozen hackers typing at a dozen consoles for a dozen days) and to enjoy some of the richness of MDL from the inside. I especially thank Chris Reeve ("the oracle") for the patience to answer questions and resolve doubts, as he no doubt has done innumerable times before.

<div align="right">

S. W. Galley
May 16, 1977

</div>

This document was prepared using the PUB system (originally from the Stanford Artificial Intelligence Laboratory) and printed on the Xerox Graphics Printer of the M.I.T. Artificial Intelligence Laboratory.

Foreword

Trying to explain MDL to an uninitiate is somewhat like trying to untie a Gordian knot. Whatever topic one chooses to discuss first, full discussion of it appears to imply discussion of everything else. What follows is a discursive presentation of MDL in an order hopefully requiring the fewest forward references. It is not perfect in that regard; however, if you are patient and willing to accept a few, stated things as "magic" until they can be explained better, you will probably not have too many problems understanding what is going on.

There are no "practice problems"; you are assumed to be learning MDL for some purpose, and your work in achieving that purpose will be more useful and motivated than artificial problems. In several cases, the examples contain illustrations of important points which are not covered in the text. Ignore examples at your peril.

This document does not assume knowledge of any specific programming language on the part of the reader. However, "computational literacy" is assumed: the reader should have written at least one program before. Also, very little familiarity with ITS or TENEX -- the time-sharing operating systems under which MDL runs -- is assumed, namely just file and user naming conventions.

Notation:

Sections marked [1] are recommended for an uninitiate's first reading, in lieu of a separate introduction or primer for MDL. [On first reading, text within brackets like these should be ignored.]

Most specifically indicated examples herein are composed of pairs of lines. The first line of a pair always ends in $ (which is how the ASCII character ESC is represented, and which always represents it); this is the input. The second line is the result of MDL's groveling over the first. If you were to type all the first lines at MDL, it would respond with all the second lines. (More exactly, the "first line" is an object in MDL (sometimes with an associated comment) followed by $, and the "second line" is everything up to but not including the next "first line".)

Anything which is written in the MDL language or which is typed on a computer console appears herein in a gothic font, as in ROOT. A metasyntactic variable -- something to be replaced in actual use by something else -- appears as *lookup:oblist*, in an italic font; often the variable will have both a meaning and a data type (as here), but sometimes one of those will be omitted, for obvious reasons.

An ellipsis (...) indicates that something uninteresting has been omitted. The character ^ means that the following character is to be "controlified": it is usually typed by holding down a console's CTRL key and striking the other key.

List of Chapters

6

List of Sections

8

12

List of Sections

14

Chapter 1. Basic Interaction

The purpose of this chapter is to provide you with that minimal amount of information needed to experiment with MDL while reading this document. It is strongly recommended that you do experiment, especially upon reaching chapter 5 (Simple Functions).

1.1. Loading MDL [1]

First, catch your rabbit. Somehow get the interpreter (the program in the file SYS: TS MDL under ITS or MDL.SAV under TENEX) running. The interpreter will first type out some news relating to MDL, if any, then type

        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 console 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. There is no ambiguity, since the program is just an incarnation of the concept "MDL".

1.2. Typing [1]

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

Typing $ (ESC) 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. ESC will be represented by the glyph $ in this document.

Typing the rubout character (DEL under ITS, control-A under TENEX) 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 CRT display consoles, rubout may "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 console is a "display" console (for example, IMLAC, ARDS, Datapoint), it first clears the screen.

Typing ^G (control-G) causes MDL to stop whatever it is doing and act as if an ERROR had occurred (section 1.4). ^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.)

Typing ^S (control-S) causes MDL to throw away what it is currently doing and return to a normal "listening" state. ^S is generally most useful for aborting infinite loops and similar terrible things. ^S destroys whatever is going on, and so it is not 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, 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. Loading a File [1]

If you have a program in MDL that you have written as an ASCII file on some device, 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). Omitted parts of the file name are taken from the default: DSK: INPUT > (ITS) or DSK: INPUT.MUD (TENEX) in the current disk directory.

Once you type $, MDL will process the text in the file (including FLOADs) exactly as if you had typed it on a console 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".

1.4. Errors — Simple Considerations [1]

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 console 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 kill 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 first line while still in the error state (before <ERRET>), MDL will print, as shown, the arguments which gave indigestion to the unhappy function:

```
<ARGS <FRAME <FRAME>>>$
[   arguments to unhappy function   ]
```

This will be explained by and by.

Chapter 2. Read, Evaluate, and Print

2.1. General [1]

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, whose output is typed on the console.

[Actually, the sequence is more like READ, TERPRI, EVAL, PRIN1, TERPRI (explained in chapter 11); MDL gives you a carriage-return line-feed when the READ is complete, that is, when all brackets are paired.]

Functionally,

> READ: printable representations --> MDL objects

> EVAL: MDL objects --> MDL objects

> PRINT: MDL objects --> printable representations

That is, READ takes ASCII text, such as is typed in at a console, 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.

2.2. Philosophy (TYPEs) [1]

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

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 the TYPE of the MDL object. Every MDL object has a TYPE, and every TYPE has its own peculiarities.

There are many different TYPEs in MDL; they will gradually be introduced below, but in the meantime here is a representative sample: SUBR (the TYPE of READ, EVAL and PRINT), FSUBR, LIST, VECTOR, FORM, FUNCTION, etc. Since every object has a TYPE, one often abbreviates "an object of TYPE *type*" by saying "a *type*".

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 "EVALs to *something*" should be taken as an abbreviation for "when given to EVAL, causes EVAL to return *something*".

As abstract entities, MDL objects are, of course, not "visible". There is, however, a standard way of representing abstract MDL objects in the real world. The standard way of representing any given TYPE of MDL object will be given below when the TYPE is introduced. These standard representations are what READ understands, and what PRINT produces.

2.3. Example (TYPE FIX) [1]

        1S
        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 console the decimal character representation of the corresponding integer.

2.4. Example (TYPE FLOAT) [1]

        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.)

2.5. Example (TYPE ATOM, PNAME) [1]

        GEORGE$
        GEORGE

This time a lot more happened.

READ noted that what was typed had no special meaning, and therefore assumed that it was the representation of an identifier, that is, 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 [a LIST of OBLISTs, available as the local value of the ATOM OBLIST]. If READ finds an ATOM in its table corresponding to the representation, that ATOM is returned as READ's value. If READ fails in the lookup, it creates a new ATOM, puts it in the table with the representation read [INSERT into <1 .OBLIST> usually], 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. One often abbreviates "object of TYPE ATOM with PNAME name" by saying "ATOM name".

EVAL, given an ATOM, returned just that ATOM.

PRINT, given an ATOM, typed out its PNAME.

At the end of this chapter, the question "what is a legal PNAME" will be considered. Further on, the methods used to attach values to ATOMs will be described.

2.6. FIXes, FLOATs, and ATOMs versus READ: Specifics

2.6.1. READ and FIXed-point Numbers

READ considers any grouping of characters which are solely digits to be a FIX, and the default radix

of the representation is decimal. 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, you will get an overflow error (unless disabled).

The radix used by READ and PRINT is changeable by the user; however, there are two formats for representations of FIXes which cause READ to use a specified radix independent of the current one. These are as follows:

(1) If a group of digits is immediately followed by a . (period), READ interprets that group as the decimal representation of a FIX. For example, 10. is always interpreted by READ as the decimal representation of ten.

(2) If a group of digits is immediately enclosed on both sides by asterisks (*), READ interprets that group as the octal representation of a FIX. For example, *10* is always interpreted by READ as the octal representation of eight.

2.6.2. 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 FLOATs.

"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.

"Scientific" notation consists of:

(1) a number,
(2) immediately followed by E or e (upper or lower case letter E),
(3) immediately followed by an exponent,

where a "number" is an arbitrarily long string of digits, with or without a decimal point (see following note); and an "exponent" is up to two digits worth of FIX. This notation represents the "number" to the "exponent" power of ten. Note: if the "number" as above would by itself be a FIX, and if the "exponent" is positive, and if the result is within the allowed range of FIXes, 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.

2.6.3. READ and PNAMEs

The question "what is a legal PNAME?" is actually not a reasonable one to ask; <u>any</u> non-empty string of <u>arbitrary</u> characters can be the PNAME of an ATOM. However, some PNAMEs are easier to type to READ than others. But even the question "what are easily typed PNAMEs?" is not too reasonable, **because:** READ **decides** that a group of characters is a PNAME by <u>default</u>; if it can't possibly be anything **else, it's a** PNAME. So, the rules governing the specification of PNAMEs 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 you are neither a perfectionist nor a masochist, skip to the next chapter.

2.6.3.1. Non-PNAMEs

A group of characters is <u>not</u> a PNAME if:

> (1) It represents a FLOAT or a FIX, as described above -- that is, it is composed wholly of digits, or digits and a single . (period), or digits and a . and the letter E or e (with optional minus signs in the right places).

> (2) It begins with a . (period).

> (3) It contains -- if typed interactively -- any of the characters which have special interactive effects: ^@, ^D, ^L, ^G, ^S, $ (ESC), rubout.

> (4) It contains a format character -- space, carriage-return, line-feed, form-feed, horizontal tab, vertical tab.

> (5) It contains a , (comma) or a # (number sign) or a ' (single quote) or a ; (semicolon) or a % (percent sign).

> (6) It contains any variety of bracket -- (or) or [or] or < or > or { or } or ".

In addition, the character \ (backslash) has a special interpretation, as mentioned below. Also, the pair of characters ! - (exclamation-point hyphen) has an extremely special interpretation, which you will reach at chapter 15.

The characters mentioned in cases 4 through 6 are "separators" -- that is, they signal to READ that whatever it was that the preceding characters represented, it's done now. They can also indicate the start of a new object's representation (all the opening "brackets" do just that).

2.6.3.2. Examples

The following examples are not in the "standard format" of "*line typed in*$ *result printed*", because they are not, in some cases, complete objects; hence, READ would continue waiting for the brackets to

be closed. In other cases, they will produce errors in EVAL if other -- currently irrelevant -- conditions are not met. Instead, the right-hand column will be used to state just what READ thought the input in the left-hand column really was.

ABC$	an ATOM of PNAME ABC
abc$	an ATOM of PNAME abc
ARBITRARILY-LONG-PNAME$	an ATOM of PNAME ARBITRARILY-LONG-PNAME
1.2345$	a FLOAT, PRINTed as 1.2345000
1.2.345$	an ATOM of PNAME 1.2.345
A.or.B$	an ATOM of PNAME A.or.B
.A.or.B$	not an ATOM, but (as explained later) a FORM containing an ATOM of PNAME A.or.B
MORE THAN ONE$	three ATOMs, with PNAMEs MORE, and THAN, and ONE
ab(cd$	an ATOM of PNAME ab, followed by the start of something else (The something else will contain an ATOM of PNAME beginning cd.)
12345A34$	an ATOM of PNAME 12345A34 (If the A had been an E, the object would have been a FLOAT.)

2.6.3.3. \ (Backslash) in ATOMs

If you have a strange, uncontrollable compulsion to have what were referred to as "separators" above as part of the PNAMEs of your ATOMs, you can do so by preceding them with the character \ (backslash). \ will also magically turn an otherwise normal FIX or FLOAT into an ATOM if it appears amongst the digits. In fact, backslash in front of any character changes it from something special to "just another character" (including the character \). It is an escape character.

When PRINT confronts an ATOM which had to be backslashed in order to be an ATOM, it will dutifully type out the required \s. They will not, however, necessarily be where you typed them; they will instead be at those positions which will cause READ the least grief. For example, PRINT will type out a PNAME which consists wholly of digits by first typing a \ and then typing the digits - no matter where you originally typed the \ (or \s).

2.6.3.4. Examples of Awful ATOMs

The following examples illustrate the amount of insanity that can be perpetrated by using \. The format of the examples is again non-standard, this time not because anything is unfinished or in error, but because commenting is needed; PRINT doesn't do it full justice.

a\ one\ and\ a\ two$ one ATOM, whose PNAME has four spaces in it

1234\56789$ an ATOM of PNAME 123456789, which PRINTs as \123456789

123\ $ an ATOM of PNAME 123*space*, which PRINTs as \123\ , with a space on the end

\\$ an ATOM whose PNAME is a single backslash

Chapter 3. Built-in Functions

3.1. Representation [1]

Up to this point, all the objects we have been concerned with have had no internal structure discernible in MDL. While the characteristics of objects with internal structure differ greatly, the way READ and PRINT handle them is uniform, to wit:

> READ, when applied to the representation of a structured object, builds and returns an object of the indicated TYPE with elements formed by applying READ to each of their representations in turn.

> PRINT, when applied to a structured object, produces a representation of the object, with its elements represented as PRINT applied to each of them in turn.

A MDL object which is used to represent the application of a function to its arguments is an object of TYPE FORM. Its printed representation is

> $<$ *func arg1 arg2 ... argn* $>$

where *func* is an object which designates the function to be applied, and *arg1* through *argn* are objects which designate the arguments or "actual parameters" or "inputs". A FORM is just a structured object which is stored and can be manipulated like a LIST (its "primitive type" is LIST -- chapter 6). The application of the function to the arguments is done by EVAL. The usual meaning of "function" (uncapitalized) in this document will be anything applicable to arguments.

3.2. Evaluation [1]

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

First, examine the *func* (first element) of the FORM. If it is an ATOM, look at its "value" (global or local, in that order -- see next chapter). 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, generate an error. Otherwise, follow its directions in evaluating or not evaluating the arguments (chapters 9 and 19) and then "apply the function" -- that is, EVAL the body of the object gotten from *func*.

3.3. Built-in Functions (TYPE SUBR, TYPE FSUBR) [1]

The built-in functions of MDL come in two varieties: those which have all their arguments EVALed before operating on them (TYPE SUBR, for "subroutine", pronounced "subber") and those which have none of their arguments EVALed (TYPE FSUBR, historically from Lisp, pronounced "effsubber"). Collectively they will be called F/SUBRs, although that term is not meaningful to the interpreter. See appendix 2 for a listing of all F/SUBRs and short descriptions. The term "Subroutine" will be used herein to mean both F/SUBRs and compiled user programs (RSUBRs and RSUBR-ENTRYs -- chapter 19).

Unless otherwise stated, <u>every</u> MDL built-in Subroutine mentioned is of TYPE <u>SUBR</u>. Also, when it is stated that an argument of a SUBR must be of a particular TYPE, note that this means that EVAL of what is there 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 'value' of the ATOM of PNAME *pname*". "The FSUBR *pname*" will be used with a similar meaning.

3.4. Examples (+ and FIX; Arithmetic) [1]

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

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

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

The SUBR FIX explicitly returns a FIXed-point number corresponding to a FLOATing-point number. FLOAT does the opposite.

```
<+ 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 FIXes gives a FIX with truncation, not rounding, of the remainder; the intermediate result remains a FIX until a FLOAT argument is encountered.

3.5. 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. (That error can be disabled, if necessary — section 16.9).

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. If RANDOM is never called with arguments, it always returns the exact same sequence of numbers, for convenience in debugging. "Debugged" programs should give RANDOM two arguments on the first call, which become the seeds for a new sequence. Popular choices of new seeds are the numbers given by TIME (which see), possibly with bits modified (chapter 18). Example ("pick a number from one to ten"):

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

30

Chapter 4. Values of Atoms

4.1. General [1]

There are two kinds of "value" which can be attached to an ATOM. An ATOM can have either, both, or neither. They interact in no way (except that alternately referring to one and then the other is inefficient). These two values are referred to as the local value and the global value of an ATOM. The terms "local" and "global" are relative to processes, not functions or programs. The SUBRs which reference the local and global values of an ATOM, and some of the characteristics of local versus global values, follow.

4.2. Global Values

4.2.1. SETG [1]

A global value can be assigned to an ATOM by the SUBR SETG ("set global"), 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 500>>$
        500

The above made the global values of both the ATOM FOO and the ATOM BAR equal to the FIXed-point number 500.

        <SETG BAR FOO>$

FOO

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

4.2.2. GVAL [1]

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 is generated.

GVAL applied to an ATOM anywhere, in any process, 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 4.2.1 were carried out in the order given, the following will evaluate as indicated:

```
,FOO$
500
<GVAL FOO>$
500
,BAR$
FOO
,,BAR$
500
```


4.2.3. Note on SUBRs and FSUBRs

The initial GVALs of the ATOMs used to refer to MDL "built-in" Subroutines are the SUBRs and FSUBRs which actually get applied when those ATOMs are referenced. If you don't like the way those supplied routines work, you are perfectly free to SETG the ATOMs to your own versions.

4.2.4. GUNASSIGN

<GUNASSIGN *atom*>

("global unassign") causes *atom* to have no assigned global value, whether or not it had one previously. The storage used for the global value can become free for other uses.

4.3. Local Values

4.3.1. SET [1]

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 like SETG.

Examples:

<SET BAR <SET FOO 100>>$
100

Both BAR and FOO have been given local values equal to the FIXed-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 might have had.

4.3.2. LVAL [1]

The SUBR used to extract the local value of an ATOM is named LVAL. 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*

The local value of an ATOM is unique within a PROCESS. SETting an ATOM in one PROCESS has no effect on its LVAL in another PROCESS, because each PROCESS has its own "control stack" (chapters 20 and 22).

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
```

4.3.3. UNASSIGN

```
<UNASSIGN atom>
```

causes *atom* to have no assigned local value, whether or not it had one previously.

4.4. VALUE

VALUE is a SUBR which takes an ATOM as an argument, and then:

(1) if the ATOM has an LVAL, returns the LVAL;

(2) if the ATOM has no LVAL but has a GVAL, returns the GVAL;

(3) if the ATOM has neither a GVAL nor an LVAL, error.

This order of seeking a value is the opposite of that used when an ATOM is the first element of a FORM. The latter will be called the G/LVAL, even though that name is not used in MDL.

Example:

```
<UNASSIGN A>$
A
<SETG A 1>$
1
<VALUE A>$
1
<SET A 2>$
```

```
2
<VALUE A>$
2
,A$
1
```

Chapter 5. Simple Functions

5.1. General [1]

The MDL equivalent of a "program" (uncompiled) is an object of TYPE FUNCTION. Actually, full-blown "programs" are usually composed of sets of FUNCTIONs, with most FUNCTIONs 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 arg1 arg2* ... >), and it always "returns" a single object, which is used as 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".

In this chapter a simple subset of the FUNCTIONs you can write is presented, namely FUNCTIONs which "act like" SUBRs with a fixed number of arguments. While this class corresponds to about 90% of the FUNCTIONs 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 SUBRs and FSUBRs, and you already know how to "use" them; you just need to be told what they do. Once you have FUNCTIONs 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.

5.2. Representation [1]

A FUNCTION is just another data object in MDL, of TYPE FUNCTION. It can be manipulated like any other data object. PRINT represents a FUNCTION like this:

    #FUNCTION (*elements*)

that is, a number sign, the ATOM FUNCTION, a left parenthesis, each of the elements of the FUNCTION, and a right parenthesis. Since PRINT represents FUNCTIONs like this, you can type them in to READ this way. (But there are a few TYPEs for which that implication is false.)

The elements of a FUNCTION can be "any number of anythings"; however, when you use a FUNCTION (apply it with a FORM), EVAL will complain if the FUNCTION does not look like

> #FUNCTION (act:atom arguments:list body)

where act is optional (section 9.8); body is at least one MDL object — any old MDL object; and, in this simple case, arguments is

> (any number of ATOMs)

that is, something READ and PRINTed as: left parenthesis, any number — including zero — of ATOMs, right parenthesis. (This is actually a normal MDL object of TYPE LIST, containing only ATOMs.)

Thus, these FUNCTIONs will cause errors — but only when used:

```
#FUNCTION ()              -- no argument LIST or body
#FUNCTION ((1) 2 7.3)     -- non-ATOM in argument LIST
#FUNCTION ((A B C D))     -- no body
#FUNCTION (<+ 1 2> A C)   -- no argument LIST
```

These FUNCTIONs will never cause errors because of format:

```
#FUNCTION (() 1 2 3 4 5)
#FUNCTION ((A) A)
#FUNCTION (()()()()()()()())
#FUNCTION ((A B C D EE F G H HIYA) <+ .A .HIYA>)
#FUNCTION ((Q) <SETG C <* .Q ,C>> <+ <MOD ,C 3> .Q>)
```

and the last two actually do something which might be useful. (The first three are rather pathological, but legal.)

5.3. Application of FUNCTIONs; Binding [1]

FUNCTIONs, like SUBRs and FSUBRs, are applied using FORMs. 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 applied to the FUNCTION, 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 value obtained</u> when the FUNCTION's body was evaluated in step (2).

-- In the above, this means return 25 as the value.

The "world" mentioned above is actually an object of TYPE ENVIRONMENT. 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.

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 completely 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 Y 0>$
0
<#FUNCTION ((X) <SET Y <* .X .X>>)  5>$
25
.Y$
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 <u>not</u> 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 or Y 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 <u>exactly</u> how a FUNCTION will <u>always</u> 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 <u>impossible</u> (1) for other people to use your FUNCTIONs, and (2) for <u>you</u> to use your FUNCTIONs a month (two weeks?) later.

5.4. Defining FUNCTIONs (FUNCTION and DEFINE) [1]

Obviously, typing #FUNCTION (...) all the time is neither reasonable nor adequate for many purposes. Normally, you just want a FUNCTION to be the GVAL of some ATOM -- the way SUBRs and FSUBRs are -- so you can use it repeatedly (and recursively). Note that you generally do not want a FUNCTION to be the LVAL of an ATOM; this has the same problems as free variables. (Of course, there are always cases where you are being clever and want the ATOM to be re-bound . . . .)

One way to "name" a FUNCTION is

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

So that

```
<SQUARE 5>$
25
<SQUARE 100>$
10000
```

Another way, which is somewhat cleaner in its typing:

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

FUNCTION is an FSUBR which simply makes a FUNCTION out of its arguments and returns the created FUNCTION.

This, however, is generally the best way:

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

The last two lines immediately above are just to prove that DEFINE did the "right thing".

DEFINE is an FSUBR which SETGs EVAL of its first argument to the FUNCTION it makes from the rest of its arguments, and then returns EVAL of its first argument. DEFINE obviously requires the least typing of the above methods, and is "best" from that standpoint. However, the real reason for using DEFINE is the following: If EVAL of DEFINE's first argument already has a GVAL, DEFINE produces an ERROR. This helps to keep you from accidently redefining things -- like MDL SUBRs and FSUBRs. The SETG constructions should be used only when you really do want to redefine something. DEFINE will be used in the rest of this document.

[Actually, if it is absolutely necessary to use DEFINE to "redefine" things, there is a "switch" which can be used: if the LVAL of the ATOM REDEFINE is T (or anything not of TYPE FALSE), DEFINE will produce no ERRORs. The normal state can be restored by evaluating <SET REDEFINE <>>. See chapter 8.]

5.5. Examples (Comments) [1]

Using SQUARE as defined above:

```
<DEFINE HYPOT (SIDE-1 SIDE-2)
        ;"This is a comment.  This FUNCTION finds the
           length of the hypotenuse of a right triangle
           of sides SIDE-1 and SIDE-2."
    <SQRT <+ <SQUARE .SIDE-1> <SQUARE .SIDE-2>>>>$
HYPOT
<HYPOT 3 4>$
5.0
```

Note that carriage-returns, line-feeds, tabs, etc. are just separators, like spaces. A comment is any single MDL object which follows a ; (semicolon). A comment can appear between any two MDL objects. A comment is totally ignored by EVAL but remembered and associated by READ with the place in the FUNCTION (or any other structured object) where it appeared. (This will become clearer after chapter 13.) The "s (double-quotes) serve to make everything between them a single MDL object, whose TYPE is STRING (chapter 7). (SQRT is the SUBR which returns the square root of its argument. It always returns a FLOAT.)

A whimsical FUNCTION:

```
<DEFINE ONE (THETA) ;"This FUNCTION always returns 1."
        <+ <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 FLOATs, and each takes its argument in radians. ATAN (arctangent) returns its value in radians. Any other trigonometric function can be compounded 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 FLOATs).

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

Two FUNCTIONs 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 LISTs 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 SETs 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
<ARGS <FRAME <FRAME>>>$
[ATM 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". Techniques for doing INC "correctly" will be covered below. Read on.

Chapter 6. Data Types

6.1. General [1]

A MDL object consists of two parts: its TYPE and its "data part" (appendix 1). The interpretation of the "data part" of an object depends of course on its TYPE. The structural organization of an object, that is, the way it is organized in storage, is referred to as its "primitive type". While there are many different TYPEs of objects in MDL, there are fewer primitive types.

All structured objects in MDL are ordered sequences of elements. As such, there are SUBRs which operate on all of them uniformly, as ordered sequences. On the other hand, the reason for having different primitive types of structured objects is that there are useful qualities of structured objects which are mutually incompatible. There are, therefore, SUBRs which do not work on all structured objects; these SUBRs exist to take full advantage of those mutually incompatible qualities. The most-commonly-used primitive types of structured objects are discussed in chapter 7, along with those special SUBRs operating on them.

It is very easy to make a new MDL object that differs from an old one only in TYPE, as long as the primitive type is unchanged. It is relatively difficult to make a new structured object that differs from an old one in primitive type, even if it has the same elements.

Before talking any more about structured objects, some information needs to be given about TYPEs in general.

6.2. Printed Representation [1]

There are many TYPEs for which MDL has no specific representation. There aren't enough different kinds of brackets. The representation used for TYPEs without any special representation is

#type representation-as-if-it-were-its-primitive-type

READ will understand that format for <u>any</u> TYPE, and PRINT will use it as a default. This

representational format will be referred to below as "# notation". It was used above to represent FUNCTIONs.

6.3. SUBRs Related to TYPEs

6.3.1. TYPE [1]

        <TYPE *any*>

returns an <u>ATOM</u> whose PNAME corresponds to the TYPE of *any*. There is no TYPE "TYPE". To type a TYPE (aren't homonyms wonderful?), just type the appropriate ATOM, like FIX or FLOAT or ATOM etc. However, in this document we will use the convention that a metasyntactic variable can have *type* for a "data type"; for example, *foo:type* means that the TYPE of *foo* is ATOM, but the ATOM must be something that the SUBR TYPE can return.

Examples:

        <TYPE 1>$
        FIX
        <TYPE 1.0>$
        FLOAT
        <TYPE +>$
        ATOM
        <TYPE ,+>$
        SUBR
        <TYPE GEORGE>$
        ATOM

6.3.2. PRIMTYPE [1]

        <PRIMTYPE *any*>

evaluates to the primitive type of *any*. The PRIMTYPE of *any* is an ATOM which also represents a TYPE. The way an object can be <u>manipulated</u> depends solely upon its PRIMTYPE; the way it is <u>evaluated</u> depends upon its TYPE.

Examples:

        <PRIMTYPE 1>$
        WORD
        <PRIMTYPE 1.0>$

```
WORD
<PRIMTYPE ,+>$
WORD
<PRIMTYPE GEORGE>$
ATOM
```

6.3.3. TYPEPRIM [1]

    <TYPEPRIM type>

returns the PRIMTYPE of an object whose TYPE is *type*. *type* is, as usual, an ATOM used to designate a TYPE.

Examples:

```
<TYPEPRIM FIX>$
WORD
<TYPEPRIM FLOAT>$
WORD
<TYPEPRIM SUBR>$
WORD
<TYPEPRIM ATOM>$
ATOM
<TYPEPRIM FORM>$
LIST
```

6.3.4. CHTYPE [1]

    <CHTYPE any type>

("change type") returns a new object that has TYPE *type* and the same "data part" as *any* (appendix 1).

```
<CHTYPE (+ 2 2) FORM>$
<+ 2 2>
```

An error is generated if the PRIMTYPE of *any* is not the same as the TYPEPRIM of *type*. An error will also be generated if the attempted CHTYPE is dangerous and/or senseless, for example, CHTYPEing a FIX to a SUBR. Unfortunately, there are few useful examples we can do at this point.

[CHTYPEing a FIX to a FLOAT or vice versa produces, in general, nonsense, since the bit formats for FIXes and FLOATs are different. The SUBRs FIX and FLOAT convert between those formats. Useful

obscurity: because of their internal representations on the PDP-10, <CHTYPE <MAX> FIX> gives the least possible FIX, and analogously for MIN.]

Passing note: "# notation" is just an instruction to READ saying "READ the representation of the PRIMTYPE normally and (literally) CHTYPE it to the specified TYPE". [Or, if the PRIMTYPE is TEMPLATE, "apply the GVAL of the TYPE name (which should be a TEMPLATE constructor) to the given elements of the PRIMTYPE TEMPLATE as arguments."]

6.4. More SUBRs Related to TYPEs

6.4.1. ALLTYPES

            <ALLTYPES>

returns a VECTOR (chapter 7) containing just those ATOMs which can currently be returned by TYPE or PRIMTYPE. This is the very "TYPE vector" (section 22.1) that the interpreter uses: look, but don't touch. No examples; try it, or see appendix 3.

6.4.2. VALID-TYPE?

            <VALID-TYPE? *atom*>

returns #FALSE () if *atom* is not the name of a TYPE, and the same object that <TYPE-C *atom*> (section 19.5) returns if it is.

6.4.3. NEWTYPE

            <NEWTYPE *atom type*>

returns *atom*, after causing it to become the representation of a brand-new TYPE whose PRIMTYPE is <TYPEPRIM *type*>. What NEWTYPE actually does is make *atom* a legal argument to CHTYPE and TYPEPRIM. (Note that names of new TYPEs can be blocked lexically to prevent collision with other names, just like any other ATOMs -- chapter 15.) Objects of a NEWTYPE-created TYPE can be generated by creating an object of the appropriate PRIMTYPE and using CHTYPE. They will be PRINTed (initially), and can be directly typed in, by the use of "# notation" as described above. EVAL of any object whose TYPE was created by NEWTYPE is initially the object itself, and, initially, you cannot APPLY something of a generated TYPE to arguments. But see below.

Examples:

```
<NEWTYPE GARGLE FIX>$
GARGLE
<TYPEPRIM GARGLE>$
WORD
<SET A <CHTYPE 1 GARGLE>>$
#GARGLE *000000000001*
<SET B #GARGLE 100>$
#GARGLE *000000000144*
<TYPE .B>$
GARGLE
<PRIMTYPE .B>$
WORD
```

6.4.4. PRINTTYPE, EVALTYPE and APPLYTYPE

<PRINTTYPE *type how*>

<EVALTYPE *type how*>

<APPLYTYPE *type how*>

all return *type*, after specifying *how* MDL is to deal with it.

These three SUBRs can be used to make newly-generated TYPEs behave in arbitrary ways, or to change the characteristics of standard MDL TYPEs. PRINTTYPE tells MDL how to print *type*, EVALTYPE how to evaluate it, and APPLYTYPE how to apply it in a FORM.

how can be either a TYPE or something that can be applied to arguments.

If *how* is a TYPE, MDL will treat *type* just like the TYPE given as *how*. *how* must have the same TYPEPRIM as *type*.

If *how* is applicable, it will be used in the following way:

For PRINTTYPE, *how* should take one argument: the object being output. *how* should output something without formatting (PRIN1-style); its result is ignored. (Note: *how* cannot use an output SUBR on *how*'s own *type*: endless recursion will result. OUTCHAN is bound during the application to the CHANNEL in use, or to a pseudo-internal channel for FLATSIZE -- chapter 11.) If *how* is the SUBR PRINT, *type* will receive no special treatment in printing, that is, it will be printed as it was in an initial MDL or immediately after its defining NEWTYPE.

For EVALTYPE, *how* should take one argument: the object being evaluated. The value returned by *how* will be used as EVAL of the object. If *how* is the SUBR EVAL, *type* will receive no special treatment in evaluation.

For APPLYTYPE, *how* should take at least one argument. The first argument will be the object being applied; the rest will be the objects it was given as arguments. The result returned by *how* will be used as the result of the application. If *how* is the SUBR APPLY, *type* will receive no special treatment in application to arguments.

If any of these SUBRs is given only one argument, that is if *how* is omitted, it returns the currently active *how* (a TYPE or an applicable object), or else #FALSE () if *type* is receiving no special treatment in that operation.

Unfortunately, these examples are fully understandable only after you have read through chapter 11.

```
<DEFINE ROMAN-PRINT (NUMB)
<COND (<OR <L=? .NUMB 0> <G? .NUMB 3999>>
        <PRINC <CHTYPE .NUMB TIME>>)
       (T
        <RCPRINT </ .NUMB 1000> '![!\M]>
        <RCPRINT </ .NUMB  100> '![!\C !\D !\M]>
        <RCPRINT </ .NUMB   10> '![!\X !\L !\C]>
        <RCPRINT     .NUMB       '![!\I !\V !\X]>)>>$
ROMAN-PRINT

<DEFINE RCPRINT (MODN V)
<SET MODN <MOD .MODN 10>>
<COND (<==? 0 .MODN>)
       (<==? 1 .MODN> <PRINC <1 .V>>)
       (<==? 2 .MODN> <PRINC <1 .V>> <PRINC <1 .V>>)
       (<==? 3 .MODN> <PRINC <1 .V>> <PRINC <1 .V>> <PRINC <1 .V>>)
       (<==? 4 .MODN> <PRINC <1 .V>> <PRINC <2 .V>>)
       (<==? 5 .MODN> <PRINC <2 .V>>)
       (<==? 6 .MODN> <PRINC <2 .V>> <PRINC <1 .V>>)
       (<==? 7 .MODN> <PRINC <2 .V>> <PRINC <1 .V>> <PRINC <1 .V>>)
       (<==? 8 .MODN>
        <PRINC <2 .V>>
        <PRINC <1 .V>>
        <PRINC <1 .V>>
        <PRINC <1 .V>>)
       (<==? 9 .MODN> <PRINC <1 .V>> <PRINC <3 .V>>)>>$
RCPRINT

<PRINTTYPE TIME FIX>  ;"fairly harmless but necessary here"$
TIME
<PRINTTYPE FIX ,ROMAN-PRINT>    ;"hee hee!"$
FIX
<+ 2 2>$
IV
```

```
1984$
MCMLXXXIV
<PRINTTYPE FIX ,PRINT>$
FIX

<NEWTYPE GRITCH LIST>    ;"a new TYPE of PRIMTYPE LIST"$
GRITCH
<EVALTYPE GRITCH>$
#FALSE ()
<EVALTYPE GRITCH LIST>  ;"evaluated like a LIST"$
GRITCH
<EVALTYPE GRITCH>$
LIST
#GRITCH (A <+ 1 2 3> !<SET A "ABC">)    ;"Type in one."$
#GRITCH (A 6 !\A !\B !\C)

<NEWTYPE HARRY VECTOR>  ;"a new TYPE of PRIMTYPE VECTOR"$
HARRY
<EVALTYPE HARRY #FUNCTION ((X) <1 .X>)>
     ;"When a HARRY is EVALed, return its first element."$
HARRY
#HARRY [1 2 3 4]$
1

<NEWTYPE WINNER LIST>    ;"a TYPE with funny application"$
WINNER
<APPLYTYPE WINNER>$
#FALSE ()
<APPLYTYPE WINNER <FUNCTION (W "TUPLE" T) (!.W !.T)>>$
WINNER
<APPLYTYPE WINNER>$
#FUNCTION ((W "TUPLE" T) (!.W !.T))
<#WINNER (A B C) <+ 1 2> q>$
(A B C 3 q)
```

The following sequence makes MDL look just like Lisp. (This example is understandable only if you know Lisp; it is included only because it is so beautiful.)

```
<EVALTYPE LIST FORM>$
LIST
<EVALTYPE ATOM ,LVAL>$
ATOM
```

So now:

```
(+ 1 2)$
3
(SET 'A 5)$
5
A$
5
```

To complete the job, of course, we would have to do some SETG's: car is 1, cdr is ,REST, and lambda is ,FUNCTION. If you really do this example, you should "undo" it before continuing:

```
<EVALTYPE 'ATOM ,EVAL>$
ATOM
<EVALTYPE LIST ,EVAL>$
LIST
```

Chapter 7. Structured Objects

This chapter discusses structured objects in general and the five basic structured PRIMTYPEs. [We defer detailed discussion of the structured PRIMTYPEs TUPLE (section 9.2) and STORAGE (section 22.2.2).]

7.1. Manipulation

The following SUBRs operate uniformly on all structured objects and generate an error if not applied to a structured object. Hereafter, *structured* represents a structured object.

7.1.1. LENGTH [1]

<LENGTH *structured*>

evaluates to the number of elements in *structured*.

7.1.2. NTH [1]

<NTH *structured fix*>

evaluates to the *fix*th element of *structured*. Error if *fix* is 0 or less, or greater than <LENGTH *structured*>. *fix* is optional, default 1.

EVAL understands the application of an object of TYPE FIX as a "shorthand" call to NTH [unless the APPLYTYPE of FIX is changed]. That is, EVAL considers the following two to be identical:

<NTH *structured fix*>   <*fix structured*>

[However, the compiler (ref 7) is happier with the former (section 9.10). The two constructs are not identical even to EVAL, if the order of evaluation is significant; for example, these two:

<NTH .X <LENGTH <SET X .Y>>>   <<LENGTH <SET X .Y>> .X>

52

are <u>not</u> identical.]

7.1.3. REST [1]

<REST *structured fix*>

evaluates to *structured* without its first *fix* elements. *fix* is optional, with 1 assumed.

Obscure but important side effect: REST actually returns *structured* "CHTYPEd" (but not through application of CHTYPE) to its PRIMTYPE. For example, REST of a FORM is a LIST. REST with an explicit second argument of 0 has no effect except for this TYPE change.

7.1.4. PUT [1]

<PUT *structured fix anything-legal*>

first makes *anything-legal* the *fix*th element of *structured*, then evaluates to *structured*. *anything-legal* is anything which can legally be an element of *structured*; often, this is synonymous with "any MDL object", but see below. Error if *fix* is 0 or less, or greater than <LENGTH *structured*>. (PUT is actually more general than this -- chapter 13.)

7.1.5. GET

<GET *structured fix*>

evaluates the same as <NTH *structured fix*>. It is more general than NTH, however (chapter 13), and is included here only for symmetry with PUT.

7.1.6. SUBSTRUC

SUBSTRUC ("substructure") facilitates the construction of structures that are composed of sub-parts of existing structures. A special case of this would be a "substring" function.

<SUBSTRUC *obj:structured rest:fix amount:fix where-to:structured*>

copies the first *amount* elements of <REST *obj rest*> into another object and returns the latter. All arguments are optional except *obj*, which must be of PRIMTYPE LIST, VECTOR, TUPLE (treated like a VECTOR), STRING, BYTES, or UVECTOR. *rest* defaults to 0, and *amount* defaults to all the elements. *where-to*, if given, receives the copied elements, starting at its beginning; it must be an object whose TYPE is the PRIMTYPE of *obj* (a VECTOR if *obj* is a TUPLE). If *where-to* is not given, a new object is

returned, of TYPE <PRIMTYPE *obj*> (a VECTOR if *obj* is a TUPLE), which <u>never</u> shares with *obj*. The copying is done in one fell swoop, not an element at a time. Note: due to an implementation restriction, if *obj* is of PRIMTYPE LIST, it must not share any elements with *where-to*.

7.2. Representation of Basic Structures

7.2.1. LIST [l]

(*element-1 element-2 ... element-N*)

represents a LIST of *N* elements.

7.2.2. VECTOR [l]

[*element-1 element-2 ... element-N*]

represents a VECTOR of *N* elements. [A TUPLE is just like a VECTOR, but it lives on the control stack.]

7.2.3. UVECTOR [l]

![*element-1 element-2 ... element-N* !]

represents a UVECTOR (uniform vector) of *N* elements. The second ! (exclamation-point) is optional for input. [A STORAGE is an archaic kind of UVECTOR that is not garbage-collected.]

7.2.4. STRING [l]

"*characters*"

represents a STRING of ASCII text. A STRING containing the character " (double-quote) is represented by placing a \ (backslash) before the double-quote inside the STRING. A \ in a STRING is represented by two consecutive backslashes.

7.2.5. BYTES

#*n* {*element-1 element-2 ... element-N*}

represents a string of *N* uniformly-sized bytes of size *n* bits.

7.2.6. TEMPLATE

> *{ element-1 element-2 ... element-N }*

represents a TEMPLATE of *N* elements when output, not input -- when input, a *#* and a TYPE must precede it.

7.3. Evaluation of Basic Structures [1]

This section and the next two describe how EVAL treats the basic structured TYPEs [in the absence of any modifying EVALTYPE calls (section 6.4.4)].

EVAL of a STRING [or BYTES or TEMPLATE] is just the original object.

EVAL acts exactly the same with LISTs, VECTORs, and UVECTORs: it generates a <u>new</u> object with elements equal to EVAL of the elements it is given. This is one of the simplest means of constructing a structure. However, see section 7.7 below.

7.4. Examples [1]

```
(1 2 <+ 3 4>)$
(1 2 7)
<SET FOO [5 <- 3> <TYPE "ABC">]>$
[5 -3 STRING]
<2 .FOO>$
-3
<TYPE <3 .FOO>>$
ATOM
<SET BAR ![("meow") (.FOO)]>$
![("meow") ([5 -3 STRING])!]
<LENGTH .BAR>$
2
<REST <1 <2 .BAR>>>$
[-3 STRING]
[<SUBSTRUC <1 <2 .BAR>> 0 2>]$
[[5 -3]]
<PUT .FOO 1 SNEAKY>          ;"Watch out for .BAR !"$
[SNEAKY -3 STRING]
.BAR$
![("meow") ([SNEAKY -3 STRING])!]
<SET FOO <REST <1 <1 .BAR>> 2>>$
```

```
"ow"
.BAR$
![("meow") ([SNEAKY -3 STRING])!]
```

7.5. Generation of Basic Structures

Since LISTs, VECTORs, UVECTORs, and STRINGs [and BYTESes] are all generated in a fairly uniform manner, methods of generating them will be covered together here. [TEMPLATEs cannot be generated by the interpreter itself; see ref 1.]

7.5.1. Direct Representation [1]

Since EVAL of a LIST, VECTOR, or UVECTOR is a new LIST, VECTOR, or UVECTOR with elements which are EVAL of the original elements, simply evaluating a representation of the object you want will generate it. (Care must be taken when representing a UVECTOR that all elements have the same TYPE.) This method of generation was exclusively used in the examples of section 7.4. Note that new STRINGs [and BYTESes] will not be generated in this manner, since the contents of a STRING are not interpreted or copied by EVAL. The same is true of any other TYPE whose TYPEPRIM happens to be LIST, VECTOR, or UVECTOR [again, assuming it neither has been EVALTYPEd nor has a built-in EVALTYPE, as do FORM and SEGMENT].

7.5.2. QUOTE [1]

QUOTE is an FSUBR of one argument which returns its argument unevaluated. READ and PRINT understand the character ' (single-quote) as an abbreviation for a call to QUOTE, the way period and comma work for LVAL and GVAL. Examples:

```
<+ 1 2>$
3
'<+ 1 2>$
<+ 1 2>
```

Any LIST, VECTOR, or UVECTOR in a program that is constant and need not have its elements evaluated should be represented directly and inside a call to QUOTE. This technique prevents the structure from being copied each time that portion of the program is executed. Examples hereafter will adhere to this dictum. (Note: one should never modify a QUOTEd object. The compiler will one day put it in read-only (pure) storage.)

7.5.3. LIST, VECTOR, UVECTOR, and STRING (the SUBRs) [1]

Each of the SUBRs LIST, VECTOR, UVECTOR, and STRING takes any number of arguments and returns an object of the appropriate TYPE whose elements are EVAL of its arguments. There are limitations on what the arguments to UVECTOR and STRING may EVAL to, due to the nature of the objects generated. See sections 7.6.5 and 7.6.6 below.

LIST, VECTOR, and UVECTOR are generally used only in special cases, since Direct Representation produces exactly the same effect (in the absence of errors), and the intention is more transparent. STRING, on the other hand, produces effects very different from literal STRINGs.

Examples:

```
<LIST 1 <+ 2 3> ABC>$
(1 5 ABC)
(1 <+ 2 3> ABC)$
(1 5 ABC)
<STRING "A" <2 "QWERT"> <REST "ABC"> "hello">$
"AWBChello"
"A <+ 2 3> (5)"$
"A <+ 2 3> (5)"
```

7.5.4. ILIST, IVECTOR, IUVECTOR, and ISTRING [1]

Each of the SUBRs ILIST, IVECTOR, IUVECTOR, and ISTRING ("implicit" or "iterated" whatever) creates and returns an object of the obvious TYPE. The format of an application of any of them is

$$< \textit{Ithing number-of-elements:fix expression:any} >$$

where *Ithing* is one of ILIST, IVECTOR, IUVECTOR, or ISTRING. An object of LENGTH *number-of-elements* is generated, whose elements are EVAL of *expression*.

expression is optional. When it is not specified, ILIST, IVECTOR, and IUVECTOR return objects filled with objects of TYPE LOSE (PRIMTYPE WORD) as place holders, a TYPE which can be passed around and have its TYPE checked, but otherwise is an illegal argument. If *expression* is not specified in ISTRING, you get a STRING made up of ^@ characters.

When *expression* is supplied as an argument, it is re-EVALuated each time a new element is generated. (Actually, EVAL of *expression* is re-EVALuated, since all of these are SUBRs.) See the last example for how this argument may be used.

[By the way, in a construct like <IUVECTOR 9 '.X>, even if the LVAL of X evaluates to itself, so that the ' could be omitted without changing the result, the compiler is much happier with the ' in place.]

IUVECTOR and ISTRING again have limitations on what *expression* may EVAL to; again, see sections 7.6.5 and 7.6.6 below.

Examples:

```
<ILIST 5 6>$
(6 6 6 6 6)
<IVECTOR 2>$
[#LOSE *000000000000* #LOSE *000000000000*]

<SET A 0>$
0
<IUVECTOR 9 '<SET A <+ .A 1>>>$
![1 2 3 4 5 6 7 8 9!]
```

7.5.5. FORM and IFORM

Sometimes the need arises to create a FORM without EVALing it or making it the body of a FUNCTION. In such cases the SUBRs FORM and IFORM ("implicit form") can be used (or QUOTE can be used). They are entirely analogous to LIST and ILIST. Example:

```
<DEFINE INC-FORM (A)
        <FORM SET .A <FORM + 1 <FORM LVAL .A>>>>$
INC-FORM
<INC-FORM FOO>$
<SET FOO <+ 1 .FOO>>
```

7.6. Unique Properties of Primitive TYPEs

7.6.1. LIST (the PRIMTYPE) [1]

An object of PRIMTYPE LIST may be considered as a "pointer chain" (appendix 1). Any MDL object may be an element of a PRIMTYPE LIST. It is easy to add and remove elements of a PRIMTYPE LIST, but the higher N is, the longer it takes to access the Nth element. The SUBRs which work only on objects of PRIMTYPE LIST are these:

7.6.1.1. PUTREST [1]

<PUTREST *head:primtype-list tail:primtype-list*>

changes *head* so that <REST *head*> is *tail* (actually <CHTYPE *tail* LIST>), then evaluates to *head*. Note that this actually changes *head*; it also changes anything having *head* as an element or a value. For example:

```
<SET BOW [<SET ARF (B W)>]>$
[(B W)]
<PUTREST .ARF '(3 4)>$
(B 3 4)
.BOW$
[(B 3 4)]
```

7.6.1.2. CONS

```
<CONS new list>
```

("construct") adds *new* to the front of *list*, without copying *list*, and returns the resulting LIST. References to *list* are not affected.

[Evaluating <CONS .E .LIST> is equivalent to evaluating (.E !.LIST) (section 7.7) but is less preferable to the compiler (ref 7).]

7.6.2. "Array" PRIMTYPEs [1]

VECTORs, UVECTORs, and STRINGs [and BYTESes and TEMPLATEs] may be considered as "arrays" (appendix 1). It is easy to access the Nth element irrespective of how large N is, and it is relatively difficult to add and delete elements. The following SUBRs can be used only with an object of PRIMTYPE VECTOR, UVECTOR, or STRING [or BYTES or TEMPLATE]. (In this section *array* represents an object of such a PRIMTYPE.)

7.6.2.1. BACK [1]

```
<BACK array fix>
```

This is the opposite of REST. It evaluates to *array*, with *fix* elements put back onto its front end, and changed to its PRIMTYPE. *fix* is optional, with default 1. If *fix* is greater than the number of elements which have been RESTed off, error. Example:

```
<SET ZOP <REST '![1 2 3 4] 3>>$
![4!]
<BACK .ZOP 2>$
![2 3 4!]
<SET S <REST "Right is might." 15>>$
" "
<BACK .S 6>$
```

"might."

7.6.2.2. TOP [1]

      <TOP *array*>

"BACKs up all the way" -- that is, evaluates to *array*, with all the elements which have been RESTed off put back onto it, and changed to its PRIMTYPE. Example:

      <TOP .ZOP>$
      ![1 2 3 4!]

7.6.3. "Vector" PRIMTYPEs

7.6.3.1. GROW

      <GROW *vu back:fix front:fix*>

adds/removes elements to/from either or both ends of *vu*, and returns the entire (TOPped) resultant object. *vu* can be of PRIMTYPE VECTOR or UVECTOR. *back* specifies a lower bound for the number of elements to be added to the <u>end</u> of *vu*; *front* specifies the same for the <u>beginning</u>. A negative *fix* specifies removal of elements.

The number of elements added to each respective end is *back* or *front* <u>increased</u> to an integral multiple of X, where X is 32 for PRIMTYPE VECTOR and 64 for PRIMTYPE UVECTOR (1 produces 32 or 64; -1 produces 0). The elements added will be LOSEs if *vu* is of PRIMTYPE VECTOR, and "empty" whatever-they-are's if *vu* is of PRIMTYPE UVECTOR. An "empty" object of PRIMTYPE WORD contains zero. An "empty" object of any other PRIMTYPE has zero in its "value word" (appendix 1) and is not safe to play with; it should be replaced via PUT.

Note that, if elements are added to the beginning of *vu*, previously-existing references to *vu* will have to use TOP or BACK to get at the added elements.

Caution: GROW is a <u>very</u> expensive operation; it <u>requires</u> a garbage collection (section 22.4) <u>every</u> time it is used. It should be reserved for <u>very</u> <u>special</u> circumstances, such as where the pattern of shared subobjects is terribly important.

Example:

      <SET A '![1]>$
      ![1!]
      <GROW .A 0 1>$
      ![0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1!]
.A$
![1!]
```

7.6.3.2. SORT

This SUBR will sort PRIMTYPEs VECTOR, UVECTOR and TUPLE (section 9.2). It works most efficiently if the sort keys are of PRIMTYPE WORD, ATOM or STRING. However, the keys may be of any TYPE, and SORT will still work. SORT acts on fixed-length records which consist of one or more contiguous elements in the structure being sorted. One element in the record is declared to be the sort key. Also, any number of additional structures can be rearranged based on how the main structure is sorted.

<SORT *pred* s1 *l1 off* s2 *l2* s3 *l3* ... *sn ln*>

where:

pred is either (see chapter 8 for information about predicates):

(1) TYPE FALSE, in which case the TYPEs of all the sort keys must be the same; they must be of PRIMTYPE WORD, STRING or ATOM; and a radix-exchange sort is used; or

(2) something applicable to two sort keys which returns TYPE FALSE if the first is not bigger than the second, in which case a shell sort is used. For example ,G? sorts numbers in ascending order, ,L? in descending order. Note: if your *pred* is buggy, the SORT may never terminate.

s1 . . . *sn* are the (PRIMTYPE) VECTORs, UVECTORs or TUPLEs being sorted, and *s1* contains the sort keys;

l1 . . . *ln* are the corresponding lengths of sort records (optional, default 1); and

off is the offset from start of record to sort key (optional, default 0).

SORT returns the sorted *s1* as a value.

Note: the SUBR SORT calls the RSUBR (chapter 19) SORTX; if the RSUBR must be loaded, you may see some output from the loader on your console.

Examples:

```
<SORT <> <SET A <IUVECTOR 500 '<RANDOM>>>>$
![...!]
```

sorts a UVECTOR of random integers.

```
<SET V [1 MONEY 2 SHOW 3 READY 4 GO]>$
[...]
<SORT <> .V 2 1>$
[4 GO 1 MONEY 3 READY 2 SHOW]

<SORT ,L? .V 2>$
[4 GO 3 READY 2 SHOW 1 MONEY]
.V$
[4 GO 3 READY 2 SHOW 1 MONEY]

<SORT <> ![2 1 4 3 6 5 8 7] 1 0 .V>$
![1 2 3 4 5 6 7 8!]
.V$
[GO 4 READY 3 SHOW 2 MONEY 1]
```

The first sort was based on the ATOMs' PNAMEs, considering records to be two elements. The second one sorted based on the FIXes. The third interchanged pairs of elements of each of its structured arguments.

7.6.4. VECTOR (the PRIMTYPE) [1]

Any MDL object may be an element of a PRIMTYPE VECTOR. A PRIMTYPE VECTOR takes two words of storage more than an equivalent PRIMTYPE LIST, but takes it all in a contiguous chunk, whereas a PRIMTYPE LIST may be physically spread out in storage (appendix 1). There are no SUBRs or FSUBRs which operate only on PRIMTYPE VECTOR.

7.6.5. UVECTOR (the PRIMTYPE) [1]

The difference between PRIMTYPEs UVECTOR and VECTOR is that every element of a PRIMTYPE UVECTOR must be of the same TYPE. A PRIMTYPE UVECTOR takes approximately half the storage of a PRIMTYPE VECTOR or PRIMTYPE LIST and, like a PRIMTYPE VECTOR, takes it in a contiguous chunk (appendix 1).

[Note: due to an implementation restriction, PRIMTYPE STRINGs, BYTESes, LOCDs (chapter 12), and objects on the control stack (chapter 22) may <u>not</u> be elements of PRIMTYPE UVECTORs.]

The "same TYPE" restriction causes an equivalent restriction to apply to EVAL of the arguments to either of the SUBRs UVECTOR or IUVECTOR. Note that attempting to say

```
![1 .A!]
```

will cause READ to produce an error, since you're attempting to put a FORM and a FIX into the same UVECTOR. On the other hand,

```
<UVECTOR 1 .A>
```

is legal, and will EVAL to the appropriate UVECTOR without error if .A EVALs to a TYPE FIX.

The following SUBRs work on PRIMTYPE UVECTORs alone.

7.6.5.1. UTYPE [1]

```
<UTYPE primtype-uvector>
```

("uniform type") evaluates to the TYPE of every element in its argument. Example:

```
<UTYPE '![A B C]>$
ATOM
```

7.6.5.2. CHUTYPE [1]

```
<CHUTYPE uv:primtype-uvector type>
```

("change uniform type") changes the UTYPE of uv to type, simultaneously changing the TYPE of all elements of uv, and returns the new, changed, uv. This works only when the PRIMTYPE of the elements of uv can remain the same through the whole process. (Exception: a uv of UTYPE LOSE can be CHUTYPEd to any type (legal in a UVECTOR of course); the resulting elements are "empty", as for GROW.)

CHUTYPE actually changes uv; hence all references to that object will reflect the change. This is quite different from CHTYPE.

Examples:

```
<SET LOST <IUVECTOR 2>>$
![#LOSE *000000000000* #LOSE *000000000000*!]
<UTYPE .LOST>$
LOSE
<CHUTYPE .LOST FORM>$
![<> <>!]
.LOST
![<> <>!]
<CHUTYPE .LOST LIST>$
![() ()!]
```

7.6.6. STRING (the PRIMTYPE) and CHARACTER [1]

The best mental image of a PRIMTYPE STRING is a PRIMTYPE UVECTOR of CHARACTERs -- where CHARACTER is the MDL TYPE for a single ASCII character. The representation of a CHARACTER, by the way, is

> !\any-ASCII-character

That is, the characters !\ (exclamation-point backslash) preceding a single ASCII character represent the corresponding object of TYPE CHARACTER (PRIMTYPE WORD). (The characters !" (exclamation-point double-quote) preceding a character are also acceptable for inputting a CHARACTER, for historical reasons.)

The SUBR ISTRING will produce an error if you give it an argument that produces a non-CHARACTER. STRING can take either CHARACTERs or STRINGs.

There are no SUBRs which uniquely manipulate PRIMTYPE STRINGs, but some are particularly useful in connection with them:

7.6.6.1. ASCII [1]

> <ASCII fix-or-character>

If its argument is of TYPE FIX, ASCII evaluates to the CHARACTER with the 7-bit ASCII code of its argument. Example: <ASCII 65> evaluates to !\A.

If its argument is of TYPE CHARACTER, ASCII evaluates to the FIXed-point number which is its argument's 7-bit ASCII code. Example: <ASCII !\Z> evaluates to 90.

[Actually, a FIX can be CHTYPEd to a CHARACTER (or vice versa) directly, but ASCII checks in the former case that the FIX is within the permissible range.]

7.6.6.2. PARSE [1]

> <PARSE string radix:fix>

PARSE applies to its argument READ's algorithm for converting ASCII representations to MDL objects and returns the first object created. The remainder of string, after the first object represented, is ignored. radix (optional, default ten) is used for converting any FIXes that occur. [See also sections 15.7.2 and 17.1.3 for additional arguments.]

7.6.6.3. LPARSE [1]

LPARSE ("list parse") is exactly like PARSE (above), except that it parses the <u>entire</u> *string* and returns a LIST of <u>all</u> objects created. If given an empty STRING or one containing only separators, LPARSE returns an empty LIST, whereas PARSE gets an error.

7.6.6.4. UNPARSE [1]

> <UNPARSE *any radix:fix*>

UNPARSE applies to its argument PRINT's algorithm for converting MDL objects to ASCII representations and returns a STRING which contains the CHARACTERs PRINT would have typed out. [However, this STRING will <u>not</u> contain any of the gratuitous carriage-returns PRINT adds to accomodate a CHANNEL's finite line-width (section 11.2.7).] *radix* (optional, default ten) is used for converting any FIXes that occur.

7.6.7. BYTES

A (PRIMTYPE) BYTES is a string of uniformly-sized bytes. The bytes can be any size between 1 and 36 bits inclusive. A BYTES is similar in some ways to a UVECTOR of FIXes and in some ways to a STRING of non-seven-bit bytes. The elements of a BYTES are always of TYPE FIX.

The SUBRs BYTES and IBYTES are similar to STRING and ISTRING, respectively, except that each of the former takes a first argument giving the size of the bytes in the generated BYTES. BYTES takes one required argument which is a FIX specifying a byte size and any number of PRIMTYPE WORDs. It returns an object of TYPE BYTES with that byte size containing the objects as elements. These objects will be ANDBed with the appropriate mask of 1-bits to fit in the byte size. IBYTES takes two required FIXes and one optional argument. It uses the first FIX to specify the byte size and the second to specify the number of elements. The third argument is repeatedly evaluated to generate FIXes that become elements of the BYTES (if it is omitted, bytes filled with zeros are generated). The analog to UTYPE is BYTE-SIZE. Examples:

```
<BYTES 3 <+ 2 2> 9 -1>$
#3 {4 1 7}
<SET A 0>$
0
<IBYTES 3 9 '<SET A <+ .A 1>>>$
#3 {1 2 3 4 5 6 7 0 1}
<IBYTES 3 4>$
#3 {0 0 0 0}
<BYTE-SIZE <BYTES 1>>$
1
```

7.6.8. TEMPLATE

A TEMPLATE is similar to a PL/I "structure" of one level: the elements are packed together and reduced in size to save storage space, while an auxiliary internal data structure describes the packing format and the elements' real TYPEs (appendix 1). The interpreter itself is not able to create objects of PRIMTYPE TEMPLATE (ref 1); however, it can apply the standard built-in Subroutines to them, with the same effects as with other "arrays".

7.7. SEGMENTs [1]

Objects of TYPE SEGMENT (whose TYPEPRIM is LIST) look very much like FORMs. SEGMENTs, however, undergo a non-standard evaluation designed to ease the construction of structured objects from elements of other structured objects.

7.7.1. Representation [1]

The representation of an object of TYPE SEGMENT is the following:

        !< func arg1 arg2 ... argn !>

where the second ! (exclamation-point) is optional, and *func* and *arg1* through *argn* are any legal consituents of a FORM (that is, anything). The pointed brackets can be implicit, as in the period and comma notation for LVAL and GVAL.

All of the following are SEGMENTs:

        !<3 .FOO>   !.FOO   !,FOO

7.7.2. Evaluation [1]

A SEGMENT is evaluated in exactly the same manner as a FORM, with the following three exceptions:

    (1) It had better be done inside an EVAL of a structure, else error. (See special case of FORMs in section 7.7.5 below.)

    (2) It had better EVAL to a structured object, else error.

    (3) What actually gets inserted into the structure being built are the elements of the structure returned by the FORM-like evaluation.

7.7.3. Examples [1]

```
<SET ZOP '![2 3 4]>$
![2 3 4!]
<SET ARF (B 3 4)>$
(B 3 4)
(.ARF !.ZOP)$
((B 3 4) 2 3 4)
![!.ZOP !<REST .ARF>!]$
![2 3 4 3 4!]

<SET S "STRUNG.">$
"STRUNG."
(!.S)$
(!\S !\T !\R !\U !\N !\G !\.)

<SET NIL ()>$
()
[!.NIL]$
[]
```

7.7.4. Note on Efficiency [1]

Most of the cases in which it is possible to use SEGMENTs require EVAL to generate an entire new object. Naturally, this uses up both storage and time. However, there is one case which it is possible to handle without copying, and EVAL uses it. When the structure being built is a PRIMTYPE LIST, and the segment value of a PRIMTYPE LIST is the last (rightmost) element being concatenated, that last PRIMTYPE LIST is not copied. This case is similar to CONS and is the principle reason why PRIMTYPE LISTs have their structures more easily varied than PRIMTYPE VECTOR or UVECTOR.

Examples:

```
.ARF$
(B 3 4)
```

This does not copy ARF:

```
(1 2 !.ARF)$
(1 2 B 3 4)
```

These do:

```
(1 !.ARF 2)            ;"not last element"$
(1 B 3 4 2)
```

```
[1 2 !.ARF]              ;"not PRIMTYPE LIST"$
[1 2 B 3 4]
(1 2 !.ARF !<REST '(1)>) ;"still not last element"$
(1 2 B 3 4)
```

Note the following, which occurs because copying does not take place:

```
<SET DOG (A !.ARF)>$
(A B 3 4)
<PUT .ARF 1 "BOWOW">$
("BOWOW" 3 4)
.DOG$
(A "BOWOW" 3 4)
<PUT .DOG 3 "WOOF">$
(A "BOWOW" "WOOF" 4)
.ARF$
("BOWOW" "WOOF" 4)
```

Since ARF was not copied, it was literally part of DOG. Hence, when an element of ARF was changed, DOG was changed. Similarly, when an element of DOG which ARF shared was changed, ARF was changed too.

7.7.5. SEGMENTs in FORMs [1]

When a SEGMENT appears as an element of a FORM, the effect is approximately the same as if the elements of EVAL of the SEGMENT were in the FORM. Example:

```
<SET A '![1 2 3 4]>$
![1 2 3 4!]
<+ !.A 5>$
15
```

Note: the elements of the structure segment-evaluated in a FORM are not re-evaluated if the thing being applied is a SUBR. Thus if .A were (1 2 <+ 3 4> 5), the above example would produce an error; you can't add up FORMs.

You could perform the same summation of 5 and the elements of A by using

```
<EVAL <CHTYPE (+ !.A 5) FORM>>
```

(Note that EVAL must be explicitly called as a SUBR; if it were not so called, you would just get the FORM <+ 1 2 3 4 5> -- not its "value".) However, the latter is more expensive both in time and in storage; when you use the SEGMENT directly in the FORM, a new FORM is, in fact, not generated as it is in the latter case. (The elements are put on "the control stack" with the other arguments.)

7.8. Self-referencing Structures

It is possible for a structured object to "contain" itself, either as a subset or as an element, as an element of a structured element, etc. Such an object cannot be PRINTed, because recursion begins and never terminates. Warning: if you try the examples in this section with a live MDL, be sure you know how to use ^S (section 1.2) to save PRINT from endless agony. (Certain constructs with ATOMs can give PRINT similar trouble: see chapters 12 and 15.)

7.8.1. Self-subset

<PUTREST *head:primtype-list tail:primtype-list*>

If *head* is a subset of *tail*, that is, if <REST *tail fix*> is the same object as <REST *head* 0> for some *fix*, then both *head* and *tail* will be "circular" (and thus self-referencing) after the PUTREST. Example:

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

7.8.2. Self-element

<PUT *s1:structured fix s2:structured*>

If *s1* is the same object as *s2*, then it will "contain" itself (and thus be self-referencing) after the PUT. Examples:

```
<SET S <LIST 1 2 3>>        ;"or VECTOR"$
(1 2 3)
<PUT .S 3 .S>$
(1 2 (1 2 (1 2 (1 2 ...
<SET U ![![]]>$
![![]!]
<PUT .U 1 .U>$
![![![![![![...
```

Test your reaction time or your console's bracket-maker. Amaze your friends.

Chapter 8. Truth

8.1. Truth Values [1]

MDL represents "false" with an object of a particular TYPE: TYPE FALSE (unsurprisingly). TYPE FALSE is structured; its PRIMTYPE is LIST. Thus, you can give reasons or excuses by making them elements of a FALSE. (Again, EVALing a FALSE neither copies it nor EVALs its elements, so it is not necessary to QUOTE a FALSE appearing in a program.) Objects of TYPE FALSE are represented in "# notation":

        #FALSE *list-of-its-elements*

The empty FORM evaluates to the empty FALSE:

        <>$
        #FALSE ( )

Anything which is not FALSE, is, reasonably enough, true. In this document the "data type" *false-or-any* in metasyntactic variables means that the only significant attribute of the object in that context is whether its TYPE is FALSE or not.

8.2. Predicates [1]

There are numerous MDL F/SUBRs which can return a FALSE or a true. See appendix 2 to find them all. Most return either #FALSE ( ) or the ATOM with PNAME T. (The latter is for historical reasons, namely Lisp.) Some predicates which are meaningful now are described next.

8.2.1. Arithmetic [1]

        <0? *fix-or-float*>

evaluates to T only if its argument is identically equal to 0 or 0.0.

```
<1? fix-or-float>
```

evaluates to T only if its argument is identically equal to 1 or 1.0.

```
<G? n:fix-or-float m:fix-or-float>
```

evaluates to T only if *n* is algebraically greater than *m*. L=? is the Boolean complement of G?; that is, it is T only if *n* is not algebraically greater than *m*.

```
<L? n:fix-or-float m:fix-or-float>
```

evaluates to T only if *n* is algebraically less than *m*. G=? is the Boolean complement of L?.

8.2.2. Equality and Membership [1]

```
<==? e1:any e2:any>
```

evaluates to T only if *e1* is the <u>same</u> <u>object</u> as *e2* (appendix 1). Two objects that look the same when PRINTed may not be ==?. Two FIXes of the same "value" are "the same object"; so are two FLOATs of <u>exactly</u> the same "value". Empty objects of PRIMTYPE LIST (and no other structured PRIMTYPE) are ==? if their TYPEs are the same. Example:

```
<==? <SET X "RANDOM STRING"> <TOP <REST .X 6>>>$
T
<==? .X "RANDOM STRING">$
#FALSE ()
```

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

```
<=? e1:any e2:any>
```

evaluates to T if *e1* and *e2* have the same TYPE and are structurally equal – that is, they "look the same", their printed representations are the same. =? is much slower than ==?. =? should be used only when its characteristics are necessary; they are not in any comparisons of unstructured objects. ==? and =? always return the same value for FIXes, FLOATs, ATOMs, etc. (Mnemonically, ==? tests for "more equality" than =?; in fact, it tests for actual physical identity.)

Example, illustrating non-copying of a SEGMENT in Direct Representation of a LIST:

```
<SET A '(1 2 3)>$
(1 2 3)
<==? .A (!.A)>$
T
<==? .A <SET B <LIST !.A>>>$
```

```
#FALSE ()
<=? .A .B>$
T
```

N=? is the Boolean complement of =?.

```
<MEMBER object:any structured>
```

runs down *structured* from first to last element, comparing each element of *structured* with *object*. If it finds an element of *structured* which is =? to *object*, it returns <REST *structured i*> (which is of TYPE <PRIMTYPE *structured*>), where the (*i*+1)th element of *structured* is =? to *object*. That is, the first element of what it returns is the <u>first</u> element of *structured* that is =? to *object*.

If no element of *structured* is =? to *object*, MEMBER returns #FALSE ().

The search is more efficient if *structured* is of PRIMTYPE VECTOR (or UVECTOR, if possible) than if it is of PRIMTYPE LIST. As usual, if *structured* is constant, it should be QUOTEd.

If *object* and *structured* are of PRIMTYPE STRING [or BYTES], MEMBER does a substring search. Example:

```
<MEMBER "PART" "SUM OF PARTS">$
"PARTS"
```

```
<MEMQ object:any structured>
```

("member quick") is exactly the same as MEMBER, except that the comparison test is ==? .

```
<STRCOMP s1 s2>
```

("string comparison") can be given either two STRINGs or two ATOMs as arguments. In the latter case the PNAMEs are used. It actually isn't a predicate, since it can return three possible values: 0 if *s1* is =? to *s2*; 1 if *s1* sorts alphabetically after *s2*; and -1 if *s1* sorts alphabetically before *s2*. "Alphabetically" means, in this case, according to the numeric order of ASCII, with the standard alphabetizing rules.

[A predicate suitable for an ascending SORT (which see) is <G? <STRCOMP .ARG1 .ARG2> 0>.]

8.2.3. Boolean Operators [1]

```
<NOT e:false-or-any>
```

evaluates to T only if *e* evaluates to a FALSE, and to #FALSE () otherwise.

```
<AND e1 e2 ... eN>
```

AND is an FSUBR. It evaluates its arguments from first to 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? is the SUBR equivalent to AND, that is, all its arguments are evaluated before any of them is tested.

```
<OR e1 e2 ... eN>
```

OR is an FSUBR. 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? is the SUBR equivalent to OR.

8.2.4. Object Properties [1]

```
<TYPE? any type-1 ... type-N>
```

evaluates to *type-i* only if <==? *type-i* <TYPE *any*>> is true. It is faster and gives more information than ORing tests for each TYPE. If the test fails for all *type-is*, TYPE? returns #FALSE ( ).

```
<APPLICABLE? e>
```

evaluates to T only if *e* is of a TYPE that can legally be applied to arguments in a FORM, that is, be (EVAL of) the first element of a FORM being evaluated (appendix 3).

```
<MONAD? e>
```

evaluates to #FALSE ( ) only if NTH and REST (with nonzero second argument) can be performed on its argument without error. An unstructured or empty structured object will cause MONAD? to return T.

```
<STRUCTURED? e>
```

evaluates to T only if *e* is a structured object. It is not the inverse of MONAD?, since each returns T if its argument is an empty structure.

```
<EMPTY? structured>
```

evaluates to T only if its argument, which must be a structured object, has no elements.

```
<LENGTH? structured fix>
```

evaluates to <LENGTH *structured*> only if that is less than or equal to *fix*; otherwise, it evaluates to #FALSE ( ).

[If *structured* is a circular PRIMTYPE LIST, LENGTH? will return a value, whereas LENGTH will execute forever. To see if you can do <REST *structured* <+ 1 *fix*>> without error, do the test <NOT <LENGTH? *structured* *fix*>>. Mnemonically, you can think of the first two letters of LENGTH? as signifying the "less than or equal to" sense of the test.]

8.3. COND [1]

The MDL Subroutine which is most used for varying evaluation depending on a truth value is the FSUBR COND ("conditional"). A call to COND has this format:

        <COND *clause-1:list* ... *clause-N:list*>

where *N* is at least one.

COND always returns the result of the <u>last</u> evaluation it performs. The following rules determine the order of evaluations performed.

    (1) Evaluate the first element of each clause (from first to 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, EVALing 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 of the current clause and returns the last thing it evaluates. If it can't find a non-FALSE, it returns the last FALSE it saw.

8.3.1. Examples [1]

```
<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
```

8.4. Shortcuts with Conditionals

8.4.1. AND and OR as Short CONDs

Since AND and OR are FSUBRs, they can be used as miniature CONDs. 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. Of course, if *action(s)* are more than one thing, you must be careful that none but the last returns false or true, respectively. Watch out especially for TERPRI (chapter 11). Examples:

<AND <ASSIGNED? FLAG> .FLAG <FCN .ARG>>

applies FCN only if someone else has SET FLAG to true. (ASSIGNED? is true if its argument ATOM has an LVAL.) No error can occur in the testing of FLAG because of the order of evaluation.

<AND <SET C <OPEN "READ" "A FILE">> <LOAD .C> <CLOSE .C>>

effectively FLOADs the file (chapter 11) without the possibility of getting an error if the file cannot be opened.

8.4.2. 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 LIST element is an AND that returns a FALSE for the test. 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>>)>
```

A variation is to make the last AND argument into the test for the COND clause. (That is, the third and fourth clauses in the above example can be combined.) Of course, you must be careful that no other AND argument evaluates to a FALSE; most Subroutines do not return a FALSE without a very good reason for it. (A notable exception is TERPRI (which see).) Even safer is to use PROG (section 10.1) instead of AND.

Another variation is to increase the nesting with a new COND after the unconditional part. At least 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 could 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>>)>)>
```

Chapter 9. Functions

This chapter could be named "fun and games with argument LISTs". Its purpose is to explain the more complicated things which can be done with FUNCTIONs, and this involves, basically, explaining all the various tokens which can appear in the argument LIST of a FUNCTION. Topics are covered in what is approximately an order of increasing complexity. This order has little to do with the order in which tokens can actually appear in an argument LIST, so what an argument LIST "looks like" overall gets rather lost in the shuffle. To alleviate this problem, section 9.9 is a summary of everything that can go into an argument LIST, in the correct order. If you find yourself getting lost, please refer to that summary.

9.1. "OPTIONAL" [1]

MDL provides very convenient means for allowing optional arguments. The STRING "OPTIONAL" (or "OPT" -- they're totally equivalent) in the argument LIST allows the specification of optional arguments with default values. The syntax of the "OPTIONAL" part of the argument LIST is as follows:

> "OPTIONAL" *al-1 al-2 ... al-N*

First, there is the STRING "OPTIONAL". Then there is any number of either ATOMs or two-element LISTs, intermixed, one per optional argument. The first element of each two-element LIST must be an ATOM; this is the dummy variable. The second element is an arbitrary MDL expression. If there are required arguments, they must come before the "OPTIONAL".

When EVAL is binding the variables of a FUNCTION and sees "OPTIONAL", the following happens:

If an explicit argument was given in the position of an optional one, the explicit argument is bound to the corresponding dummy ATOM.

If there is no explicit argument and the ATOM stands alone, that is, it is not the first element of a two-element LIST, that ATOM becomes "bound", but no local value is assigned to it [see below]. A local value can be assigned to it by using SET.

If there is no explicit argument and the ATOM is the first element of a two-element LIST, the MDL expression in the LIST with the ATOM is evaluated and bound to the ATOM.

[Until an ATOM is assigned, any attempt to reference its LVAL will produce an error. The predicate SUBRs BOUND? and ASSIGNED? can be used to check for such situations. BOUND? returns T if its argument is currently bound via an argument LIST or has ever been SET while not bound via an argument LIST. The latter kind of binding is called "top-level binding", because it is done outside all active argument-LIST binding. ASSIGNED? will return #FALSE () if its argument is either unassigned or unbound. By the way, there are two predicates for global values similar to BOUND? and ASSIGNED?, namely GBOUND? and GASSIGNED?. Each returns T only if its argument, which (as in BOUND? and ASSIGNED?) must be an ATOM, has a global value "slot" (chapter 22) or a global value, respectively.]

Example:

```
<DEFINE INC1 (A "OPTIONAL" (N 1)) <SET .A <+ ..A .N>>>$
INC1
<SET B 0>$
0
<INC1 B>$
1
<INC1 B 5>$
6
```

Here we defined another (not quite working) increment FUNCTION. It now takes an optional argument specifying how much to increment the ATOM it is given. If not given, the increment is 1. Now, 1 is a pretty simple MDL expression; there is no reason why the optional argument cannot be complicated -- for example, a call to a FUNCTION which reads a file on an I/O device.

9.2. TUPLEs

9.2.1. "TUPLE" and TUPLE (the TYPE) [1]

There are also times when you want to be able to have an arbitrary number of arguments. You can always do this by defining the FUNCTION as having a structure as its argument, with the arbitrary number of arguments as elements of the structure. This can, however, lead to inelegant-looking FORMs and extra garbage to be collected. The STRING "TUPLE" appearing in the argument LIST allows you to avoid that. It must follow explicit and optional dummy arguments (if there are any of either) and must be followed by an ATOM.

The effect of "TUPLE" appearing in an argument LIST is the following: any arguments left in the

FORM, after satisfying explicit and optional arguments, are EVALed and made sequential elements of an object of TYPE and PRIMTYPE TUPLE. The TUPLE is then bound to the ATOM following "TUPLE" in the argument LIST. If there were no arguments left by the time the "TUPLE" was reached, an empty TUPLE is bound to the ATOM.

An object of TYPE TUPLE is exactly the same as a VECTOR except that a TUPLE is not held in garbage-collected storage. It is instead held with ATOM bindings in a control stack. This does not affect manipulation of the TUPLE within the function generating it or any function called within that one; it can be treated just like a VECTOR. Note, however, that a TUPLE ceases to exist when the function which generated it returns. Returning a TUPLE as a value is a good way to generate an error. (A copy of a TUPLE can easily be generated by segment-evaluating the TUPLE into something; that copy can be returned.) The predicate LEGAL? returns #FALSE () if it is given a TUPLE generated by an APPLICABLE object which has already returned, and T if it is given a TUPLE which is still "good".

[Note to linear readers: you now are equipped to understand the NEWTYPE examples in section 6.4.4 completely.]

Example:

```
<DEFINE NTHARG (N "TUPLE" T)
                ;"Get all but first argument into T."
        <COND (<==? 1 .N> 1)
                ;"If N is 1, return 1st arg, i.e., .N,
                 i.e., 1.  Note that <1? .N> would be
                 true even if .N were 1.0."
            (<L? <LENGTH .T> <SET N <- .N 1>>>
            #FALSE ("DUMMY"))
                ;"Check to see if there is an Nth arg,
                 and make N a good index into T while
                 you're at it.
                 If there isn't an Nth arg, complain."
            (ELSE <NTH .T .N>)>>
```

NTHARG, above, takes any number of arguments. Its first argument must be of TYPE FIX. It returns EVAL of its Nth argument, if it has an Nth argument. If it doesn't, it returns #FALSE ("DUMMY"). (The ELSE is not absolutely necessary in the last clause. If the Nth argument is a FALSE, the COND will return that FALSE.) Exercise for the reader: NTHARG will generate an error if its first argument is not FIX. Where and why? (How about <NTHARG 1.5 2 3>?) Fix it.

9.2.2. TUPLE (the SUBR) and ITUPLE

These SUBRs are the same as VECTOR and IVECTOR, except that they build TUPLEs (that is, vectors on

the control stack). They can be used only at top level in an "OPTIONAL" list or "AUX" list (see below). The clear advantage of TUPLE and ITUPLE ("implicit tuple") is in storage-management efficiency. They produce no garbage, since they are flushed automatically upon function return.

Examples:

```
<DEFINE F (A B "AUX" (C <ITUPLE 10 3>)) ...>
```

creates a 10-element TUPLE and SETs C to it.

```
<DEFINE H ("OPTIONAL" (A <ITUPLE 10 '<I>>)
                      "AUX" (B <TUPLE !.A 1 2 3>))
            ...>
```

These are valid uses of TUPLE and ITUPLE. However, the following is not a valid use of TUPLE, because it is not called at top level of the "AUX":

```
<DEFINE NO (A B "AUX" (C <REST <TUPLE !.A>>)) ...>
```

However, the desired effect could be achieved by

```
<DEFINE OK (A B "AUX" (D <TUPLE !.A>) (C <REST .D>)) ...>
```


9.3. "AUX" [1]

"AUX" (or "EXTRA" -- they're totally equivalent) are STRINGs which, placed in an argument LIST, serve to dynamically allocate temporary variables for the use of a Function.

"AUX" must appear in the argument LIST after any information about explicit arguments. It is followed by ATOMs or two-element LISTs as if it were "OPTIONAL". ATOMs in the two-element LISTs are bound to EVAL of the second element in the LIST. Atoms not in such LISTs are initially unassigned; they are explicitly given "no" LVAL.

All binding specified in an argument LIST is done sequentially from first to last, so initialization expressions for "AUX" (or "OPTIONAL") can refer to objects which have just been bound. For example, this works:

```
<DEFINE AUXEX ("TUPLE" T
                  · "AUX" (A <LENGTH .T>) (B <* 2 .A>))
          ![.A .B]>$
AUXEX
<AUXEX 1 2 "FOO">$
![3 6!]
```

9.4. QUOTEd arguments [1]

If an ATOM in an argument LIST which is to be bound to a required or optional argument is surrounded by a call to QUOTE, that ATOM is bound to the unevaluated argument. Example:

```
<DEFINE Q2 (A 'B) (.A .B)>$
Q2
<Q2 <+ 1 2> <+ 1 2>>$
(3 <+ 1 2>)
```

9.5. "ARGS" [1]

The indicator "ARGS" can appear in an argument LIST with precisely the same syntax as "TUPLE". However, "ARGS" causes the ATOM following it to be bound to a LIST of the remaining unevaluated arguments.

"ARGS" does not cause any copying to take place. It simply gives you

```
<REST application:form fix>
```

with an appropriate fix. The TYPE change to LIST is a result of the REST. Since the LIST shares all its elements with the original FORM, PUTs into the LIST will change the calling program, however dangerous that may be.

Examples:

```
<DEFINE QIT (N "ARGS" L) <.N .L>>$
QIT
<QIT 2 <+ 3 4> <LENGTH ,QALL> FOO>$
<LENGTH ,QALL>

<DEFINE FUNCT1 ("ARGS" ARGL-AND-BODY)
        <CHTYPE .ARGL-AND-BODY FUNCTION>>$
FUNCT1
<FUNCT1 (A B) <+ .A .B>>$
#FUNCTION ((A B) <+ .A .B>)
```

The last example is a perfectly valid equivalent of the FSUBR FUNCTION.

9.6. "CALL"

The indicator "CALL" is an ultimate "ARGS". If it appears in an argument LIST, it must be followed by an ATOM and must be the only thing used to gather arguments. "CALL" causes the ATOM which follows it to become bound to the actual FORM that is being evaluated -- that is, you get the "function call" itself. Since "CALL" binds to the FORM itself, and not a copy, PUTs into that FORM will change the calling code.

"CALL" exists as a Catch-22 for argument manipulation. If you can't do it with "CALL", it can't be done.

9.7. EVAL and "BIND"

Obtaining unevaluated arguments, for example, via QUOTE and "ARGS", very often implies that you wish to evaluate them at some point. You can do this by explicitly calling EVAL, which is a SUBR. Example:

```
<SET F '<+ 1 2>>$
<+ 1 2>
<EVAL .F>$
3
```

EVAL can take a second argument, of TYPE ENVIRONMENT (or others, see section 20.8 below). An ENVIRONMENT consists basically of a state of ATOM bindings; it is the "world" mentioned in chapter 5. Now, since binding changes the ENVIRONMENT, if you wish to use EVAL within a FUNCTION, you probably want to get hold of the environment which existed before that FUNCTION's binding took place. The indicator "BIND", which must, if it is used, be the first thing in an argument LIST, provides this information. It binds the ATOM immediately following it to the ENVIRONMENT existing "at call time" -- that is, just before any binding is done for its FUNCTION. Example:

```
<SET A 0>$
0
<DEFINE WRONG ('B "AUX" (A 1)) <EVAL .B>>$
WRONG
<WRONG .A>$
1
<DEFINE RIGHT ("BIND" E 'B "AUX" (A 1)) <EVAL .B .E>>$
RIGHT
<RIGHT .A>$
0
```

82

9.7.1. Local Values versus ENVIRONMENTs

SET, LVAL, VALUE, BOUND?, ASSIGNED?, and UNASSIGN all take a final optional argument which has not previously been mentioned: an ENVIRONMENT (or other TYPEs, see section 20.8 below). If this argument is given, the SET or LVAL is done in the ENVIRONMENT specified. LVAL cannot be abbreviated by . (period) if it is given an explicit second argument.

This feature is just what is needed to cure the INC bug mentioned in chapter 5. A "correct" INC can be defined as follows:

```
<DEFINE INC ("BIND" OUTER ATM)
        <SET .ATM <+ 1 <LVAL .ATM .OUTER>> .OUTER>>
```

9.8. ACTIVATION, "NAME", "ACT", AGAIN, and RETURN [1]

EVALuation of a FUNCTION, after the argument LIST has been taken care of, normally consists of EVALuating each of the objects in the body in the order given, and returning the value of the last thing EVALed. If you want to vary this sequence, you need to know, at least, where the FUNCTION begins. Actually, EVAL normally hasn't the foggiest idea of where its current FUNCTION began. "Where'd I start" information is bundled up with a TYPE called ACTIVATION. In "normal" FUNCTION EVALuation, ACTIVATIONs are not generated; one can be generated, and bound to an ATOM, in either of the two following ways:

(1) Put an ATOM immediately before the argument LIST. The ACTIVATION of the Function will be bound to that ATOM.

(2) As the last thing in the argument LIST, insert either of the STRINGs "NAME" or "ACT" and follow it with an ATOM. The ATOM will be bound to the ACTIVATION of the Function.

In this document "Function" (capitalized) will designate anything that can generate an ACTIVATION; besides TYPE FUNCTION, this class includes the FSUBRs PROG, BIND, and REPEAT, yet to be discussed.

Each ACTIVATION refers explicitly to a particular evaluation of a Function. For example, if a recursive FUNCTION generates an ACTIVATION, a new ACTIVATION referring explicitly to each recursion step is generated on every recursion.

Like TUPLEs, ACTIVATIONs are held in a control stack. Unlike TUPLEs, there is no way to get a copy of an ACTIVATION which can usefully be returned as a value. (This is a consequence of the fact that ACTIVATIONs refer to evaluations; when the evaluation is finished, the ACTIVATION no longer exists.) ACTIVATIONs can be tested, like TUPLEs, by LEGAL? for legality. They are used by the SUBRs AGAIN and RETURN.

AGAIN can take one argument: an ACTIVATION. It means "start doing this again", where "this" is specified by the ACTIVATION. Specifically, AGAIN causes EVAL to return to where it started working on the <u>body</u> of the Function in the evaluation specified by the ACTIVATION. The evaluation is not redone completely; in particular, no re-binding (of arguments, "AUX" variables, etc.) is done.

RETURN can take two arguments: an arbitrary expression and an ACTIVATION, in that order. It causes the Function evaluation whose ACTIVATION it is given to terminate and return EVAL of RETURN's first argument. That is, RETURN means "quit doing this and return that", where "this" is the ACTIVATION -- its second argument -- and "that" is the expression -- its first argument. Example:

```
<DEFINE MY+ ("TUPLE" T "AUX" (M 0) "NAME" NM)
        <COND (<EMPTY? .T> <RETURN .M .NM>)>
        <SET M <+ .M <1 .T>>>
        <SET T <REST .T>>
        <AGAIN .NM>>$
MY+
<MY+ 1 3 <LENGTH "FOO">>$
7
<MY+>$
0
```

Note: suppose an ACTIVATION of one Function (call it F1) is passed to another Function (call it F2) -- for example, via an application of F2 within F1 with F1's ACTIVATION as an argument. If F2 RETURNs to F1's ACTIVATION, F2 <u>and</u> F1 terminate immediately, and <u>F1</u> returns the RETURN's first argument. Good for error exits. AGAIN can clearly pull a similar trick. In the following example, F1 computes the sum of F2 applied to each of its arguments; F2 computes the product of the elements of its structured argument, but it aborts if it finds an element that is not a number.

```
<DEFINE F1 ACT ("TUPLE" T "AUX" (T1 .T))
        <COND (<NOT <EMPTY? .T1>>
                <PUT .T1 1 <F2 <1 .T1> .ACT>>
                <SET T1 <REST .T1>>
                <AGAIN .ACT>)
              (ELSE <+ !.T>)>>$
F1


<DEFINE F2 (S A "AUX" (S1 .S))
        <REPEAT MY-ACT ((PRD 1))
          <COND (<NOT <EMPTY? .S1>>
                <COND (<NOT <TYPE? <1 .S1> FIX FLOAT>>
                        <RETURN #FALSE ("NON-NUMBER") .A>)
                      (ELSE <SET PRD <* .PRD <1 .S1>>>)>
                <SET S1 <REST .S1>>)
                (ELSE <RETURN .PRD>)>>>$
F2
```

```
<F1 '(1 2) '(3 4)>$
14
<F1 '(T 2) '(3 4)>$
#FALSE ("NON-NUMBER")
```

9.9. Argument List Summary

The following is a listing of all the various tokens which can appear in the argument LIST of a FUNCTION, in the order in which they can occur. Short descriptions of their effects are included. **All** of them are <u>optional</u> -- that is, any of them (in any position) can be left out or included -- but the order in which they appear <u>must</u> be that of this list. "QUOTEd ATOM", "matching object", and "2-list" are defined below.

(1) "BIND"

> must be followed by an ATOM. It binds that ATOM to the ENVIRONMENT which existed when the FUNCTION was applied.

(2) ATOMs and QUOTEd ATOMs (any number)

> are required arguments. QUOTEd ATOMs are bound to the matching object. ATOMs are bound to EVAL of the matching object in the ENVIRONMENT existing when the FUNCTION was applied.

(3) "OPTIONAL" or "OPT" (they're equivalent)

> is followed by any number of ATOMs, QUOTEd ATOMs, or 2-lists. These are optional arguments. If a matching object exists, an ATOM -- either standing alone or the first element of a 2-list -- is bound to EVAL of the object, performed in the ENVIRONMENT existing when the FUNCTION was applied. A QUOTEd ATOM -- alone or in a 2-list -- is bound to the matching object itself. If no such object exists, ATOMs and QUOTEd ATOMs are left unbound, and the first element of each 2-list is bound to EVAL of the corresponding second element. (This EVAL is done in the new ENVIRONMENT of the Function as it is being constructed.)

(4) "ARGS" (and <u>not</u> "TUPLE")

> must be followed by an ATOM. The ATOM is bound to a LIST of <u>all</u> the remaining arguments, <u>unevaluated</u>. (If there are no more arguments, the LIST is empty.) This LIST is actually a REST of the FORM applying the FUNCTION. If "ARGS" appears in the argument LIST, "TUPLE" should not appear.

(4) **"TUPLE"** (and <u>not</u> "ARGS")

must be followed by an ATOM. The ATOM is bound to a TUPLE ("VECTOR on the control stack") of all the remaining arguments, <u>evaluated</u> in the environment existing when the FUNCTION was applied. (If no arguments remain, the TUPLE is empty.) If "TUPLE" appears in the argument LIST, "ARGS" should not appear.

(5) **"AUX"** or **"EXTRA"** (they're equivalent)

is followed by any number of ATOMs or 2-lists. These are auxiliary variables, bound away from the previous environment for the use of this Function. ATOMs are bound in the ENVIRONMENT of the Function, but they are unassigned; the first element of each 2-list is both bound and assigned to EVAL of the corresponding second element. (This EVAL is done in the new ENVIRONMENT of the Function as it is being constructed.)

(6) **"NAME"** or **"ACT"** (they're equivalent)

must be followed by an ATOM. The ATOM is bound to the ACTIVATION of the current evaluation of the Function.

<u>ALSO</u> -- in place of sections (2) (3) <u>and</u> (4), you can have

(2-3-4) **"CALL"**

which must be followed by an ATOM. The ATOM is bound to the FORM which caused application of this FUNCTION.

The special terms used above mean this:

"QUOTEd ATOM" -- a two-element FORM whose first element is the ATOM QUOTE, and whose second element is any ATOM. (Can be typed -- and will be PRINTed -- as '*atom*.)

"Matching object" -- that element of a FORM whose position in the FORM matches the position of a required or optional argument in an argument LIST.

"2-list" -- a two-element LIST whose first element is an ATOM (or QUOTEd ATOM; see below) and whose second element can be anything but a SEGMENT. EVAL of the second element is assigned to a new binding of the first element (the ATOM) as the "default value" in "OPTIONAL" or the "initial value" in "AUX". In the case of "OPTIONAL", the first element of a 2-list can be a QUOTEd ATOM; in this case, an argument which is supplied is not EVALed, but if it is not supplied the second element of the LIST <u>is</u> EVALed and assigned to the ATOM.

9.10. APPLY [1]

Occasionally there is a valid reason for the first element of a FORM not to be an ATOM. For example, the object to be applied to arguments may be chosen at run time, or it may depend on the arguments in some way. While EVAL is perfectly happy in this case to EVALuate the first element and go on from there, the compiler (ref 7) can generate more efficient code if it knows whether the result of the evaluation will (1) always be of TYPE FIX, (2) always be an applicable (non-FIX) object that evaluates all its arguments, or (3) neither. The easiest way to tell the compiler if (1) or (2) is true is to use the ATOM NTH (section 7.1.2) in case (1) or APPLY in case (2) as the first element of the FORM.

        `<APPLY object arg-1 ... arg-N>`

evaluates *object* and all the *arg-is* and then applies the former to all the latter. An error occurs if *object* evaluates to something not applicable, or to an FSUBR, or to a FUNCTION (or user Subroutine -- chapter 19) with "ARGS" or "CALL" or QUOTEd arguments.

Example:

```
<APPLY <NTH .ANALYZERS
            <LENGTH <MEMQ <TYPE .ARG> .ARGTYPES>>>
       .ARG>
```

calls a function to analyze .ARG. Which function is called depends on the TYPE of the argument; this represents the idea of a dispatch table.

9.11. CLOSURE

        `<CLOSURE function a1 ... aN>`

where *function* is a FUNCTION, and *a1* through *aN* are any number of ATOMs, returns an object of TYPE CLOSURE. This can be applied like any other function, but, whenever it is applied, the ATOMs given in the call to CLOSURE are <u>first</u> bound to the VALUEs they had when the CLOSURE was generated, then the *function* is applied as normal. This is a "poor man's funarg".

A CLOSURE is useful when a FUNCTION must have state information remembered between calls to it, especially in these two cases: when the LVALs of external state ATOMs might be compromised by other programs, or when more than one distinct sequence of calls are active concurrently. Example of the latter: each object of a structured NEWTYPE might have an associated CLOSURE that coughs up one element at a time, remembering between calls how far it got. Often only one ATOM will be included in the CLOSURE, with a value in the CLOSURE that is a structure containing all the relevant information.

Chapter 10. Looping

10.1. PROG and REPEAT [1]

PROG and REPEAT are almost identical FSUBRs which make it possible to vary the order of EVALuation arbitrarily -- that is, to have "jumps". The syntax of PROG ("program") is

<PROG act:atom aux:list body>

where

act is an optional ATOM, which is bound to the ACTIVATION of the PROG.

aux is a LIST which looks exactly like that part of a FUNCTION's argument LIST which follows an "AUX", and serves exactly the same purpose. It is not optional. If you need no temporary variables or "ACT", make it ( ).

body is a non-zero number of arbitrary MDL expressions.

The syntax of REPEAT is identical, except that, of course, REPEAT is the first element of the FORM, not PROG.

10.1.1. Basic EVALuation [1]

Upon entering a PROG, an ACTIVATION is always generated. If there is an ATOM in the right place, the ACTIVATION is also bound to that ATOM. The variables in the aux (if any) are then bound as indicated in the aux. All of the expressions in body are then EVALuated in their order of occurrence. If nothing untoward happens, you leave the PROG upon evaluating the last expression in body, returning the value of that last expression.

PROG thus provides a way to package together a group of things you wish to do, in a somewhat more limited way than can be done with a FUNCTION. But PROGs are generally used for their other properties.

REPEAT acts in all ways <u>exactly</u> like a PROG whose last expression is <AGAIN>. The only way to leave a REPEAT is to explicitly use RETURN (or GO with a TAG -- section 10.5).

10.1.2. AGAIN and RETURN in PROG and REPEAT [1]

Within a PROG or REPEAT, you always have a defined ACTIVATION, whether you bind it to an ATOM or not. [In fact the interpreter binds it to the ATOM LPROG\ !-INTERRUPTS ("last PROG"). The FSUBR BIND is identical to PROG except that BIND does not bind that ATOM, so that AGAIN and RETURN with no ACTIVATION argument will not refer to it. This feature could be useful within MACROs.]

If AGAIN is used with no arguments, it uses the ACTIVATION of the closest surrounding PROG or REPEAT <u>within the current function</u> (error if none) and re-starts the PROG or REPEAT without rebinding the *aux* variables, just the way it works in a FUNCTION. With an argument, it can of course re-start any Function (PROG or REPEAT or FUNCTION) within which it is embedded at run time.

As with AGAIN, if RETURN is given no ACTIVATION argument, it uses the ACTIVATION of the closest surrounding PROG or REPEAT within the current function and causes that PROG or REPEAT to terminate and return RETURN's first argument. If RETURN is given <u>no</u> arguments, it causes the closest surrounding PROG or REPEAT to return the ATOM T. Also like AGAIN, it can, with an ACTIVATION argument, terminate any Function within which it is embedded at run time.

10.1.3. Examples [1]

Examples of the use of PROG are difficult to find, since it is almost never necessary, and it slows down the interpreter (chapter 24). PROG can be useful as a point of return from the middle of a computation, or inside a COND (which see), but we won't exemplify these uses. Instead, what follows is an example of a typically poor use of PROG which has been observed among Lisp programmers using MDL. Then, the same thing is done using REPEAT. In both cases, the example FUNCTION just adds up all its arguments and returns the sum. (The SUBR GO is discussed in section 10.5.)

```
;"Lisp style"
     <DEFINE MY+ ("TUPLE" TUP)
          <PROG (SUM)
                  <SET SUM 0>
          LP      <COND (<EMPTY? .TUP> <RETURN .SUM>)>
                  <SET SUM <+ .SUM <1 .TUP>>>
                  <SET TUP <REST .TUP>>
                  <GO LP>>>
```

```
;"MDL style"
    <DEFINE MY+ ("TUPLE" TUP)
        <REPEAT ((SUM 0))
            <COND (<EMPTY? .TUP> <RETURN .SUM>)>
            <SET SUM <+ .SUM <1 .TUP>>
            <SET TUP <REST .TUP>>>>>
```

Of course, neither of the above is optimal MDL code for this problem, since MY+ can be written using SEGMENT evaluation as

```
    <DEFINE MY+ ("TUPLE" TUP) <+ !.TUP>>
```

There are, of course, lots of problems which can't be handled so simply, and lots of uses for REPEAT.

10.2. MAPF and MAPR: Basics [1]

MAPF ("map first") and MAPR ("map rest") are two SUBRs which take care of a majority of cases which require loops over data. The basic idea is the following:

Suppose you have a LIST (or other structure) of data, and you want to apply a particular function to each element. That is exactly what MAPF does; you give it the function and the structure, and it applies the function to each element of the structure, starting with the first.

On the other hand, suppose you want to change each element of a structure according to a particular algorithm. This can be done only with great pain using MAPF, since you don't have easy access to the structure inside the function; you have only the structure's elements. MAPR solves the problem by applying a function to RESTs of a structure: first to <REST structure 0>, then to <REST structure 1>, etc. Thus, the function can change the structure by changing its argument, for example, by a <PUT argument 1 something>. It can even PUT a new element farther down the structure, which will be seen by the function on subsequent applications.

Now suppose, in addition to applying a function to a structure, you want to record the results -- the values returned by the function -- in another structure. Both MAPF and MAPR can do this: they both take an additional function as an argument, and, when the looping is over, apply the additional function to all the results, and then return the result of that application. Thus, if the additional function is ,LIST, you get a LIST of the previous results; if it is ,VECTOR, you get a VECTOR of results; etc.

Finally, it might be the case that you really want to loop a function over more than one structure simultaneously. For instance, consider creating a LIST whose elements are the element-by-element sum of the contents of two other LISTs. Both MAPF and MAPR allow this; you can, in fact, give each of them any number of structures full of arguments for your looping function.

This was all mentioned because MAPF and MAPR appear to be complex when seen baldly, due to the fact that the argument descriptions must take into account the general case. Simpler, degenerate cases are usually the ones used.

10.2.1. MAPF [1]

<MAPF *finalf loopf s1 s2 ... sN*>

where (after argument evaluation)

finalf is something applicable that evaluates all its arguments, or a FALSE;

loopf is something applicable to *N* arguments that evaluates all its arguments; and

s1 through *sN* are structured objects (any TYPE)

does the following:

(1) First, it applies *loopf* to *N* arguments: the first element of each of the structures. Then it RESTs each of the structures, and does the application again, looping until <u>any</u> of the structures runs out of elements. Each of the values returned by *loopf* is recorded in a TUPLE.

(2) Then, it applies *finalf* to all the recorded values simultaneously, and returns the result of that application. If *finalf* is a FALSE, the recorded values are "thrown away" (actually never recorded in the first place) and the MAPF returns only the last value returned by *loopf*. If any of the *si* structures is empty, so that *loopf* is never invoked, *finalf* is applied to <u>no</u> arguments; if *finalf* is a FALSE, MAPF returns #FALSE ().

10.2.2. MAPR [1]

<MAPR *finalf loopf s1 s2 ... sN*>

acts just like MAPF, but, instead of applying *loopf* to the elements of the structures, it applies it to RESTs of the structures.

10.2.3. Examples [1]

Make the element-wise sum of two LISTs:

```
<MAPF ,LIST ,+ '(1 2 3 4) '(10 11 12 13)>$
(11 13 15 17)
```

Change a UVECTOR to contain double its values:

```
<SET UV '![5 6 7 8 9]>$
![5 6 7 8 9!]
<MAPR <>
        #FUNCTION ((L) <PUT .L 1 <* <1 .L> 2>>)
        .UV>$
![18!]
.UV$
![10 12 14 16 18!]
```

Create a STRING from CHARACTERs:

```
<MAPF ,STRING 1 '["MODELLING" "DEVELOPMENT" "LIBRARY"]>$
"MDL"
```

Sum the squares of the elements of a UVECTOR:

```
<MAPF ,+ #FUNCTION ((N) <* .N .N>) '![3 4]>$
25
```

A parallel assignment FUNCTION (Note that the arguments to MAPF are of different lengths.):

```
<DEFINE PSET ("TUPLE" TUP)
        <MAPF <>
              ,SET
              .TUP
              <REST .TUP </ <LENGTH .TUP> 2>>>>$
PSET
<PSET A B C 1 2 3>$
3
.A$
1
.B$
2
.C$
3
```

Note: it is easy to forget that finalf must evaluate its arguments, which precludes the use of an FSUBR. It is primarily for this reason that the SUBRs AND? and OR? were invented. As an example, the predicate =? could have been defined this way:

```
<DEFINE =? (A B)
        <COND (<MONAD? .A> <==? .A .B>)
              (<AND <NOT <MONAD? .B>>
                    <==? <TYPE .A> <TYPE .B>>
                    <==? <LENGTH .A> <LENGTH .B>>>
               <MAPF ,AND? ,=? .A .B>)>>
```

[By the way, the following shows how to construct a value that has the same TYPE as an argument.

```
<DEFINE MAP-NOT (S)
 <COND (<MEMQ <PRIMTYPE .S> '![LIST VECTOR UVECTOR STRING]>
        <CHTYPE <MAPF ,<PRIMTYPE .S> ,NOT .S>
               <TYPE .S>>)>>>
```

It works because the ATOMs that name the common STRUCTURED PRIMTYPEs (LIST, VECTOR, UVECTOR and STRING) have as GVALs the corresponding SUBRs to build objects of those TYPEs.]

10.3. More on MAPF and MAPR

10.3.1. MAPRET

MAPRET is a SUBR that enables the *loopf* being used in a MAPR or MAPF (and lexically within it, that is, not separated from it by a function call) to return from zero to any number of values as opposed to just one. For example, suppose a MAPF of the following form is used:

```
<MAPF ,LIST <FUNCTION (E) ...> ...>
```

Now suppose that the programmer wants to add no elements to the final LIST on some calls to the FUNCTION and add many on other calls to the FUNCTION. To accomplish this, the FUNCTION simply calls MAPRET with the elements it wants added to the LIST. More generally, MAPRET causes its arguments to be added to the final TUPLE of arguments to which the *finalf* will be applied.

Warning: MAPRET is guaranteed to work only if it is called from an explicit FUNCTION which is the second argument to a MAPF or MAPR. In other words, the second argument to MAPF or MAPR must be #FUNCTION (...) or <FUNCTION ...> if MAPRET is to be used.

Example: the following returns a LIST of all the ATOMs in an OBLIST (chapter 15):

```
<DEFINE ATOMS (OB)
        <MAPF ,LIST
              <FUNCTION (BKT) <MAPRET !.BKT>>
              .OB>>
```

10.3.2. MAPSTOP

MAPSTOP is the same as MAPRET, except that, after adding its arguments, if any, to the final TUPLE, it forces the application of *finalf* to occur, whether or not the structured objects have run out of elements. Example: the following copies the first ten (or all) elements of its argument into a LIST:

```
<DEFINE FIRST-TEN (STRUC "AUX" (I 10))
    <MAPF ,LIST
          <FUNCTION (E)
              <COND (<0? <SET I <- .I 1>>> <MAPSTOP .E>)>
              .E>
          .STRUC>>
```

10.3.3. MAPLEAVE

MAPLEAVE is analogous to RETURN, except that it works in (lexically within) MAPF or MAPR instead of PROG or REPEAT. It flushes the accumulated TUPLE of results and returns its argument (optional, default T) as the value of the MAPF or MAPR. (It finds the MAPF/R that should return in the current binding of the ATOM LMAP\ !-INTERRUPTS ("last map").) Example: the following finds and returns the first non-zero element of its argument, or #FALSE ( ) if there is none:

```
<DEFINE FIRST-NO (STRUC)
        <MAPF <>
              <FUNCTION (X)
                <COND (<N==? .X 0> <MAPLEAVE .X>)>>
              .STRUC>>
```

10.3.4. Only two arguments

If MAPF or MAPR is given only two arguments, the iteration function *loopf* is applied to no arguments each time, and the looping continues indefinitely until a MAPLEAVE or MAPSTOP is invoked. This feature gives MAPF and MAPR all the features of STACKFORM (below). Example: the following returns a LIST of the integers from one less than its argument to zero.

```
<DEFINE LNUM (N)
        <MAPF ,LIST
              <FUNCTION ()
                <COND (<0? <SET N <- .N 1>>> <MAPSTOP 0>)
                      (ELSE .N)>>>>
```

10.4. STACKFORM

STACKFORM is archaic, due to improvements in the implementation of MAPF/R (above). In fact,

        `<STACKFORM function arg pred>`

is exactly equivalent to

        `<MAPF function`
                `<FUNCTION () <COND (pred arg) (T <MAPSTOP>)>>>`

In fact MAPF/R is more powerful, because MAPRET, MAPSTOP, and MAPLEAVE provide flexibility not available with STACKFORM.

This FSUBR is used to apply a function to a variable number of arguments, as determined by an arbitrary predicate. This could also be done by constructing the appropriate FORM and EVALing it. STACKFORM is more efficient: it builds the "FORM" on the control stack (section 22.1), which produces no "garbage". This is where STACKFORM obtains its name. One principle use involves processing input characters, in cases where you don't know how many characters are going to arrive. The example below demonstrates this, using SUBRs which are more fully explained in chapter 11. Another example can be found in chapter 13.

An application of STACKFORM looks like this:

        `<STACKFORM function arg pred>`

where
    function must EVAL to a function which evaluates all its arguments;
    arg is an arbitrary expression; and
    pred is another arbitrary expression which should be capable of returning a FALSE when EVALed.

Evaluation of an application of a STACKFORM proceeds as follows (it is vaguely like a "while" loop):

(1) Evaluate *function* and place the result on the control stack.

(2) Evaluate *pred*, and then:

    (2.1) If *pred* evaluated to non-FALSE, evaluate *arg* and place the result on the control stack. Then go back to the start of (2).

    (2.2) If *pred* evaluated to a FALSE, apply the stacked *function* to the stacked *args* and return the result.

Example: the following FUNCTION reads characters from the current input channel until an $ (ESC) is read, and then returns what was read as one STRING. (The SUBR READCHR reads one character from the input channel and returns it. NEXTCHR returns the next CHARACTER which READCHR will return -- chapter 11.)

```
<DEFINE RDSTR ()
        <STACKFORM ,STRING
                   <READCHR>
                   <NOT <==? <NEXTCHR> <ASCII 27>>>>>$
RDSTR

<PROG () <READCHR> ;"Flush the ESC ending this input."
        <RDSTR>>$
ABC123<+ 3 4>$"ABC123<+ 3 4>"
```

10.5. GO and TAG

GO is provided in MDL for people who can't recover from a youthful experience with Basic, Fortran, PL/I, etc. The SUBRs previously described in this chapter are much more tasteful for making good, clean, "structured" programs. GO just bollixes things.

GO is a SUBR which allows you to break the normal order of evaluation and re-start just before any top-level expression in a PROG or REPEAT. It can take two TYPEs of arguments: ATOM or TAG.

Given an ATOM, GO searches the *body* of the immediately surrounding PROG or REPEAT within the current Function, starting after *aux*, for an occurrence of that ATOM at the top level of *body*. (This search is effectively a MEMQ.) If it doesn't find the ATOM, error. If it does, evaluation is resumed at the expression following the ATOM.

The SUBR TAG generates and returns objects of TYPE TAG. This SUBR takes one argument: an ATOM which would be a legal argument for a GO. An object of TYPE TAG contains sufficient information to allow you to GO to any top-level position in a PROG or REPEAT from within any function called inside the PROG or REPEAT. GO with a TAG is vaguely like AGAIN with an ACTIVATION; it allows you to "go back" to the middle of any PROG or REPEAT which called you. Also like ACTIVATIONs, TAGs into a PROG or REPEAT can no longer be used after the PROG or REPEAT has returned. LEGAL? can be used to see if a TAG is still valid.

10.6. Looping versus Recursion

Since any program in MDL can be called recursively, champions of "pure Lisp" or somesuch may be tempted to implement any repetitive algorithm using recursion. 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".

Chapter 11. Input/Output

The transmission of information between an object in MDL and an external storage device can be done in three ways. Historically, the first way was to convert an object into a string of characters, or vice versa. The transformation is nearly one-to-one (although some MDL objects, for example TUPLEs, cannot be input in this way) and is similar in style to Fortran's formatted I/O. It is what READ and PRINT do.

The second way is used for the contents of MDL objects rather than the objects themselves. Here an image of numbers or characters within an object is transmitted, similar in style to Fortran's unformatted I/O. Usually imaged I/O cannot be properly done from/to a console.

The third way is to dump an object in a clever format so that it can be reproduced exactly when input the next time. Exact reproduction means that any sharing between structures or self-reference is preserved; only the garbage collector itself can do I/O in this way.

11.1. Conversion I/O

All conversion-I/O SUBRs in MDL take an optional argument which directs their attention to a specific I/O channel. This section will describe SUBRs without their optional arguments. In this situation, they all refer to a default channel, initially the console running the MDL. When given an optional argument, that argument follows any arguments indicated here. Some of these SUBRs also have additional optional arguments, relevant to conversion, discussion of which will be deferred until later.

11.1.1. Input

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

11.1.1.1. 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 2. See also sections 11.3, 15.7.1, and 17.1.3 for optional arguments.

11.1.1.2. READCHR

> <READCHR>

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

11.1.1.3. NEXTCHR

> <NEXTCHR>

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

11.1.2. 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 SUBRs 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 (section 11.2.7).

11.1.2.1. 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 2.

11.1.2.2. PRIN1

> <PRIN1 any>

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

11.1.2.3. 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 (chapter 15) which would otherwise be necessary.

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

11.1.2.4. TERPRI

> <TERPRI>

("terminate printing") outputs a carriage-return line-feed and then returns #FALSE ( )!

11.1.2.5. CRLF

> <CRLF>

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

11.1.2.6. FLATSIZE

> <FLATSIZE *any max:fix radix:fix*>

does not actually cause any output to occur and does not take a CHANNEL argument. Instead, it compares *max* with the number of characters PRIN1 would take to print *any*. If *max* is less than the number of characters needed (including the case where *any* is self-referencing), FLATSIZE returns #FALSE ( ); otherwise, it returns the number of characters needed to PRIN1 *any*. *radix* (optional, default ten) is used for converting any FIXes that occur.

This SUBR is especially useful in conjunction with (section 11.2.7) those elements of a CHANNEL which specify the number of characters per output line and the current position on an output line.

11.2. CHANNEL (the TYPE)

I/O channels are dynamically assigned in MDL, and are represented by an object of TYPE CHANNEL, which is of PRIMTYPE VECTOR. The format of a CHANNEL will be explained later, in section 11.2.7. First, how to generate and use them.

11.2.1. OPEN

        `<OPEN mode file-spec>`

or

        `<OPEN mode name1 name2 dev dir>`

OPEN is a SUBR which creates and returns a CHANNEL. All its arguments must be of TYPE STRING, and _all_ are optional. The preceding statement is false when the device is "INT" or "NET"; see sections 11.9 and 11.11. If the attempted opening of an ITS or TENEX I/O channel fails, OPEN returns #FALSE (_reason:string file-spec:string status:fix_), where the _reason_ and the _status_ are supplied by the operating system, and the _file-spec_ is the name of the file that MDL was trying to open.

The first argument to OPEN tells the mode, as understood by ITS, in which the CHANNEL will be used, for a typical device:

"READ"	unit ASCII input (default)
"PRINT"	unit ASCII output
"READB"	block image input (not from a console)
"PRINTB"	block image output (not to a console)
"PRINTO"	block image update in place (only on "DSK", file must exist)
"DISPLAY"	graphics

The above modes are significant only under ITS, and only for devices about which MDL has no special knowledge or which it does not treat specially. The programmer's choice of mode is usually determined by which SUBRs will be used on the CHANNEL. The following table tells which SUBRs can be used with which modes, where OK indicates an allowed use:

"READ"	"PRINT"	"READB"	"PRINTB" "PRINTO"	"DISPLAY"	mode / SUBRs
OK		OK			READ READCHR NEXTCHR READSTRING FILECOPY FILE-LENGTH LOAD
	OK		OK*		PRINT PRIN1 PRINC IMAGE TERPRI FILECOPY PRINTSTRING BUFOUT RENAME
		OK			READB GC-READ
			OK		PRINTB GC-DUMP
OK		OK	OK		ACCESS
OK	OK	OK	OK		RESET
OK	OK				ECHOPAIR
OK					TTYECHO TYI
				OK	DISPLAY ERASE

* PRINTing (or PRIN1ing) an RSUBR (chapter 19) on a "PRINTB" or "PRINTO" CHANNEL has special effects.

The next one to four arguments to OPEN specify the file involved. If only one STRING is used, it can contain the entire specification, according to standard operating-system syntax. Otherwise, the string(s) are interpreted as follows:

name1 is the first file name, that part to the left of the space (ITS) or period (TENEX). Default: <VALUE NM1>, if any, otherwise "INPUT".

name2 is the second file name, that part to the right of the space (ITS) or period (TENEX). Default: <VALUE NM2>, if any, otherwise ">" (ITS) or "MDL" and highest version number (TENEX).

dev is the device name. Default: <VALUE DEV>, if any, otherwise "DSK".

dir is the disk-directory name. Default: <VALUE SNM>, if any, otherwise the "working-directory" name as defined by the operating system.

Examples:

<OPEN "PRINT" "TPL:"> opens a conversion-output CHANNEL to the TPL device under ITS.

<OPEN "PRINT" "DUMMY" "NAMES" "TPL"> does the same.

<OPEN "PRINT" "TPL"> opens a CHANNEL to the file DSK:TPL > (ITS) or DSK:TPL.MDL (TENEX).

<OPEN "READ" "FOO" ">" "DSK" "GUEST"> opens a conversion-input CHANNEL to that file.

<OPEN "READ" "GUEST;FOO"> does the same under ITS.

11.2.2. OPEN-NR

OPEN-NR is the same as OPEN, except that the date of last reference of the opened file is not changed.

11.2.3. CHANNEL (the SUBR)

CHANNEL is called exactly like OPEN, but it always returns an unopened CHANNEL, which can later be opened by RESET (below) just as if it had once been open.

11.2.4. CLOSE

        <CLOSE *channel*>

closes *channel* and returns its argument, with its "state" changed to "closed". If *channel* is for output, all buffered output is written out first. No harm is done if *channel* is already CLOSEd.

11.2.5. CHANLIST

        <CHANLIST>

returns a LIST whose elements are all the currently open CHANNELs. The first two elements are usually ,INCHAN and ,OUTCHAN (see below). A CHANNEL not referenced by anything except <CHANLIST> will be CLOSEd during garbage collection.

11.2.6. INCHAN and OUTCHAN

The default channel for input SUBRs is the local value of the ATOM INCHAN. The default channel for output SUBRs is the local value of the ATOM OUTCHAN.

You can direct I/O to a CHANNEL by SETting 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. Thus the CHANNEL being used is available for READ macros (section 17.1) and PRINTTYPEs (section 6.4.4).)

By the way, a good trick for playing with INCHAN and OUTCHAN within a function is to use the ATOMs INCHAN and OUTCHAN as "AUX" variables, re-binding their local values to the CHANNEL you want. When you leave, of course, the old LVALs are restored (which is the whole point). The ATOMs must be declared SPECIAL (chapter 14) for this trick to compile correctly.

INCHAN and OUTCHAN also have global values, initially the CHANNELs directed at the console running MDL. Initially, INCHAN's and OUTCHAN's local and global values are the same.

11.2.7. Contents of CHANNELs

The contents of an object of TYPE CHANNEL are accessed by the I/O SUBRs each time such a SUBR is used. If you change the contents of a CHANNEL (for example, with PUT), the next use of that CHANNEL will be changed appropriately. Some elements of CHANNELs, however, should be played with seldom, if ever, and only at your peril. These are marked below with an * (asterisk). Caveat user.

There follows a table of the contents of a CHANNEL, the TYPE of each element, and an interpretation. The format used is the following:

element-number: type interpretation

11.2.7.1. Output CHANNELs

The contents of a CHANNEL used for output are as follows:

-1: LIST	transcript channel(s) (see below)
* 0: varies	device-dependent information
* 1: FIX	channel number (ITS) or jfn (TENEX), 0 for display or internal or closed
* 2: STRING	mode
* 3: STRING	first file name argument
* 4: STRING	second file name argument
* 5: STRING	device name argument
* 6: STRING	directory name argument
* 7: STRING	real first file name
* 8: STRING	real second file name
* 9: STRING	real device name
*10: STRING	real directory name
*11: FIX	various status bits
*12: FIX	PDP-10 instruction used to do one I/O operation
13: FIX	number of characters per line of output
14: FIX	current character position on a line
15: FIX	number of lines per page
16: FIX	current line number on a page
17: FIX	access pointer for file-oriented devices
18: FIX	radix for FIX conversion
19: FIX	information needed when outputting to the display

N.B.: The elements of a CHANNEL below number 1 are usually invisible but are obtainable via <NTH <TOP *channel*> *fix*>, for some appropriate *fix*.

The transcript-channels slot has this meaning: if this slot contains a LIST of CHANNELs, then anything input or output on the original CHANNEL is output on these CHANNELs. Caution: do not use a CHANNEL as its own transcript channel; you probably won't live to tell about it.

11.2.7.2. Input CHANNELs

The contents of the elements up to number 12 of a CHANNEL used for input are the same as that for output. The remaining elements are as follows ((same) indicates that the use is the same as that for output):

13: varies	object evaluated when end of file is reached
*14: FIX	one "look-ahead" character, used by READ
*15: FIX	PDP-10 instruction executed waiting for input
16: LIST	queue of buffers for input from a console
17: FIX	access pointer for file-oriented devices (same)
18: FIX	radix for FIX conversion (same)
19: STRING	buffer for input

11.3. End-of-File "Routine"

As mentioned above, an explicit CHANNEL is the first optional argument of all SUBRs used for conversion I/O. The second optional argument for conversion-input SUBRs 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 QUOTEd FORM which applies a function of yours. The default value of this argument is a call to ERROR. Note: the CHANNEL has been CLOSEd by the time this argument is evaluated.

Example: the following FUNCTION counts the occurrences of a character in a file, according to its arguments. The file names, device, and directory are optional, with the usual defaults.

```
<DEFINE COUNT-CHAR
        (CHAR "TUPLE" FILE "AUX" (CNT 0) (CHN <OPEN "READ" !.FILE>))
      <COND (.CHN                ;"If CHN is FALSE, bad OPEN: return the FALSE
                                 so result can be tested by another FUNCTION."
            <REPEAT ()
                <AND <==? .CHAR <READCHR .CHN '<RETURN>>>
                    <SET CNT <+ 1 .CNT>>>>
                ;"Until EOF, keep reading and testing a character at a time."
        .CNT                     ;"Then return the count.")>>
```

11.4. Imaged I/O

11.4.1. Input

11.4.1.1. READB

> <READB *buffer:uvector-or-storage channel eof:any*>

The *channel* must be open in "READB" mode. READB will read as many 36-bit binary words as necessary to fill the *buffer* (whose UTYPE must be of PRIMTYPE WORD), unless it hits the end of file. READB returns the number of words actually read, as a FIXed-point number. This will normally be the length of the *buffer*, unless the end of file was read, in which case it will be less, and only the beginning of *buffer* will have been filled (SUBSTRUC may help). An attempt to READB again, after *buffer* is not filled, will evaluate the end-of-file routine *eof*, which is optional, default a call to ERROR.

11.4.1.2. READSTRING

> <READSTRING *buffer:string channel stop:fix-or-string eof*>

is the STRING analog to READB, where *buffer* and *eof* are as in READB, and *channel* is any input CHANNEL (default .INCHAN). *stop* tells when to stop inputting: if a FIX, read this many CHARACTERs (default: fill up *buffer*); if a STRING, stop reading if any CHARACTER in this STRING is read (don't include this CHARACTER in final STRING).

11.4.2. Output

11.4.2.1. PRINTB

> <PRINTB *buffer:uvector-or-storage channel*>

This call writes the entire contents of the *buffer* into the specified channel open in "PRINTB" or "PRINTO" mode. It returns *buffer*.

11.4.2.2. PRINTSTRING

> <PRINTSTRING *buffer:string channel count:fix*>

is analogous to READSTRING. It outputs *buffer* on *channel*, either the whole thing or the first *count* characters, and returns the number of characters output.

11.4.2.3. IMAGE

        <IMAGE *fix channel*>

is a rather special-purpose SUBR. When any conversion-output routine outputs an ASCII control character (with special exceptions like carriage-returns, line-feeds, etc.), it actually outputs two characters: ^ (circumflex), followed by the upper-case character which has been control-shifted. IMAGE, on the other hand, always outputs the real thing: that ASCII character whose ASCII 7-bit code is *fix*. It is guaranteed not to give any gratuitous line-feeds or such. *channel* is optional, with default .OUTCHAN, and its slots for current character position (number 14) and current line number (16) are not updated. IMAGE returns *fix*.

11.5. Dumped I/O

11.5.1. Output: GC-DUMP

        <GC-DUMP *any printb:channel-or-false*>

dumps *any* on *printb* in a clever format so that GC-READ (below) can reproduce *any* exactly, including sharing. *any* cannot live on the control stack, nor can it be of PRIMTYPE PROCESS or LOCD or ASOC (which see). *any* is returned as a value.

If *printb* is a CHANNEL, it must be open in "PRINTB" or "PRINTO" mode. If *printb* is a FALSE, GC-DUMP instead returns a UVECTOR (of UTYPE PRIMTYPE WORD) that contains what it would have output on a CHANNEL. This UVECTOR can be PRINTBed anywhere you desire, but, if it is changed <u>in any way</u>, GC-READ will not be able to input it. Probably the only reason to get it is to check its length before output.

Except for the miniature garbage collection required, GC-DUMP is about twice as fast as PRINT, but the amount of external storage used is two or three times as much.

11.5.2. Input: GC-READ

        <GC-READ *readb:channel eof:any*>

returns one object from the *channel*, which must be open in "READB" mode. The file must have been produced by GC-DUMP. *eof* is optional. GC-READ is about ten times faster than READ.

11.6. 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. RESTOREing a SAVE file is much faster than re-READing the objects it contains.

Since a SAVE file does not contain all extant MDL objects, only the impure and PURIFYed (section 22.9.2) ones, a change to the interpreter has the result of making all previous SAVE files unusable. To prevent errors from arising from this, the interpreter has a version number, which is incremented whenever changes are installed. The current version number is printed out on initially starting up the program and is available as the GVAL of the ATOM MUDDLE. This version number is written out as the very first part of each SAVE file. If RESTORE attempts to re-load a SAVE file whose version number is not the same as the interpreter being used, an ERROR is produced. If desired, the version number of a SAVE file can be obtained by doing a READ of that file. Only that initial READ will work; the rest of the file is not ASCII.

11.6.1. SAVE

                     ⟨SAVE *file-spec:string gc?:false-or-any*⟩

or

                     ⟨SAVE *name1 name2 dev dir gc?:false-or-any*⟩

saves the entire state of your MDL away in the file specified by its arguments, and then returns "SAVED". All STRING arguments are optional, with defaults "MUDDLE", "SAVE", "DSK", and ⟨VALUE SNM⟩. *gc?* is optional and, if supplied and of TYPE FALSE, causes no garbage collection to occur before SAVEing. (FSAVE is an alias for SAVE that may be seen in old programs.)

If .SNM is the null STRING (""), it will not be a part of the SAVE file, that is, RESTORE will not change .SNM. This mechanism is handy for "public" SAVE files, which should not point the user at a particular disk directory. Under ITS, the file is actually written with the name _MUDS_ > and renamed to the argument(s) only when complete, to prevent losing a previous SAVE file if a crash occurs. Under TENEX, version numbers provide the same safety.

Example:

```
<DEFINE SAVE-IT ("OPTIONAL"
                (FILE '("PUBLIC" "SAVE" "DSK" "GUEST"))
                "AUX" (DIR ,SNM))
       <SETUP>
       <SETG SNM "">
       <COND (<=? "SAVED" <SAVE !.FILE>>   ;"See below."
              <SETG SNM .DIR>
              <CLEANUP>
              "Saved.")
             (T
              <TERPRI>
              <PRINC "Amazing program at your service.">
              <TERPRI>
              <START-RUNNING>)>>
```

11.6.2. RESTORE

        ⟨RESTORE *file-spec*⟩

or

        ⟨RESTORE *name1 name2 dev dir*⟩

<u>replaces</u> the entire current state of your MDL with that SAVEd in the file specified. All arguments are optional, with defaults as in SAVE.

RESTORE completely replaces the contents of the MDL, including the state of execution existing when the SAVE was done and the state of all open I/O CHANNELs. If a file which was open when the SAVE was done does not exist when the RESTORE is done, you will get a message to that effect.

A RESTORE <u>never</u> returns (unless it gets an error): it causes a SAVE done some time ago to return <u>again</u> (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 RESTOREation.

11.7. Other I/O Functions

11.7.1. LOAD

        ⟨LOAD *input:channel oblist*⟩

eventually returns "DONE". First, however, it READs and EVALs every MDL object in the file pointed to by *input*, and then CLOSEs *input*. Any occurrences of rubout, ^@, ^D, ^L, etc., in the file are given no special meaning; they are simply ATOM constituents.

oblist is optional, used to specify a LIST of OBLISTs for the READ. Its default is .OBLIST (chapter 15).

11.7.2. FLOAD

<FLOAD *file-spec oblist*>

or

<FLOAD *name1 name2 dev dir oblist*>

("file load") acts just like LOAD, except that it takes arguments (with defaults) like OPEN, OPENs the CHANNEL itself for reading, and CLOSEs the CHANNEL when done. *oblist* is optional, as in LOAD. If the OPEN fails, an ERROR occurs, giving the reason for failure.

11.7.3. SNAME

<SNAME *string*> ("system name", a hangover from ITS) is identical in effect to <SETG SNM *string*>, that is, it causes the default *dir* argument to all SUBRs which want file specifications to become its argument, in the absence of a local value for SNM, and it returns its argument.

<SNAME> is the same in effect as <GVAL SNM>, that is, it returns the current default *dir*.

11.7.4. ACCESS

<ACCESS *channel fix*>

returns *channel*, after making the next character or binary word (depending on the mode of *channel*, which should not be "PRINT") which will be input from or output to *channel* the (*fix*+1)st one from the beginning of the file. *channel* must be open to a randomly accessible device ("DSK", "USR", etc.). A *fix* of 0 positions *channel* at the beginning of the file.

11.7.5. FILE-LENGTH

<FILE-LENGTH *input:channel*>

returns a FIX, the length of the file open on *input*. This information is supplied by the operating system, and it may not be available, for example, with the "NET" device (section 11.11). If *input*'s mode is "READ", the length is in characters (rounded up to a multiple of five); if "READB", in binary words. If ACCESS is applied to *input* and this length or more, then the next input operation will detect the end of file.

11.7.6. FILECOPY

<FILECOPY *input:channel output:channel*>

copies characters from *input* to *output* until the end of file on *input* (thus closing *input*) and returns the number of characters copied. Both arguments are optional, with respective defaults .INCHAN and .OUTCHAN. The operation is essentially a READSTRING - PRINTSTRING loop. Neither CHANNEL need be freshly OPENed, and *output* need not be immediately CLOSEd. Restriction: internally a <FILE-LENGTH *input*> is done, which must succeed; thus FILECOPY might lose if *input* is a "NET" CHANNEL.

11.7.7. RESET

<RESET *channel*>

returns *channel*, after "resetting" it. Resetting a CHANNEL is like OPENing 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*).

11.7.8. BUFOUT

<BUFOUT *output:channel*>

causes all internal MDL buffers for *output* to be written out and returns its argument. This is helpful if the operating system or MDL is flaky and you want to attempt to minimize your losses. The output may be padded with up to four extra spaces, if *output*'s mode is "PRINT".

11.7.9. RENAME

RENAME is for renaming and deleting files. It takes three kinds of arguments:
(a) two file names, in either single- or multi-STRING format, separated by the ATOM TO,

(b) one file name in either format, or

(c) a CHANNEL and a file name in either format (not under TENEX).

Omitted file-name parts use the OPEN defaults. 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. Defaults are as for OPEN. For example:

```
<RENAME "FOO 3" TO "BAR">        ;"Rename FOO 3 to BAR >."
```

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

```
<RENAME "FOO FOO DSK:HARRY;">  ;"Delete file FOO FOO from
                                HARRY's directory."
```

In case (c) the CHANNEL must be open in either "PRINT" or "PRINTB" mode, and a rename while open for writing is attempted. The real-name slots in the CHANNEL are updated to reflect any successful change.

11.8. Console CHANNELs

MDL behaves like the ITS version of the text editor Teco with respect to typing in carriage-return, in that it automatically adds a line-feed. In order to type in a lone carriage-return, a carriage-return followed by a rubout must be typed. Also PRINT, PRIN1 and PRINC do not automatically add a line-feed when a carriage-return is output. This enables overstriking on a console that lacks backspacing capability. It also means that what goes on a console and what goes in a file are more likely to look the same.

Under ITS, MDL can start up without a console, give control of the console away to an inferior job or get it back while running. Doing a RESET on either of the console channels causes MDL to find out if it now has the console; if it does, the console is reopened and the current screen size and device parameters are updated. If it doesn't have the console, an internal flag is set, causing output to the console to be ignored and attempted input from the console to make the job go to sleep.

Under ITS, there are some peculiarities associated with pseudo-consoles ("STY" and "STn" devices). If the CHANNEL given to READCHR is open in "READ" mode to a pseudo-console, and if no input is available, READCHR returns -1, TYPE FIX. If the CHANNEL given to READSTRING is open in "READ" mode to a pseudo-console, reading also stops if and when no more characters are available, that is, when READCHR would return -1.

11.8.1. ECHOPAIR

        ⟨ECHOPAIR *console-in:channel console-out:channel*⟩

returns its first argument, after making the two CHANNELs "know about each other" so that rubout, ^@, ^D and ^L on *console-in* will cause the appropriate output on *console-out*.

11.8.2. TTYECHO

        ⟨TTYECHO *console-input:channel pred*⟩

turns the echoing of typed characters on *channel* off or on, according to whether or not *pred* is of TYPE FALSE, and returns *channel*. It is useful in conjunction with TYI (below) for a program that wants to do character input and echoing in its own fashion.

11.8.3. TYI

        ⟨TYI *console-input:channel*⟩

returns one CHARACTER from *channel* (optional, default .INCHAN) when it is typed, rather than after $ (ESC) is typed, as is the case with READCHR. The following example echos input characters as their ASCII values, until a carriage-return is typed:

```
<REPEAT ((FOO <TTYECHO .INCHAN <>>))
    <AND <==? 13 <PRINC <ASCII <TYI .INCHAN>>>>
        <RETURN <TTYECHO .INCHAN T>>>>
```

11.9. Internal CHANNELs

If the device specified in an OPEN is "INT", a CHANNEL is created which does not refer to any I/O device outside MDL. In this case, the mode must be "READ" or "PRINT", and there is another argument, which must be a function.

For a "READ" CHANNEL, the function must take no arguments. Whenever a CHARACTER is desired from this CHANNEL, the function will be applied to no arguments and must return a CHARACTER. This will occur once per call to READCHR using this CHANNEL, and several times per call to READ. Under ITS, the function can signal that its "end-of-file" has been reached by returning ⟨CHTYPE *777777000003* CHARACTER⟩ (-1 in left half, control-C in right), which is the standard ITS end-of-file signal. Under TENEX, the function should return either that or ⟨CHTYPE *777777000032* CHARACTER⟩ (-1 and control-Z), the latter being the TENEX end-of-file signal.

For a "PRINT" CHANNEL, the function must take one argument, which will be a CHARACTER. It can dispose of its argument in any way it pleases. The value returned by the function is ignored.

Example: <OPEN "PRINT" "INT:" ,FCN> opens an internal output CHANNEL with ,FCN as its character-gobbler.

11.10. "DISPLAY" CHANNELs

"DISPLAY" CHANNELs are used to display graphical information, of TYPE PICTURE. The manipulation of useful PICTUREs is covered in ref 3. The following two SUBRs are not available under TENEX.

11.10.1. DISPLAY

> <DISPLAY *display:channel picture*>

displays *picture* on the device associated with *channel*. If *picture* is not given, the display on the device is refreshed, which is useful only for display consoles.

11.10.2. ERASE

> <ERASE *display:channel picture*>

removes *picture* from the display on *channel*. If *picture* is not given, all PICTUREs displayed on *channel* will be removed, leaving an empty display.

11.11. The "NET" Device: the ARPA Network

The "NET" device is different in many ways from conventional devices. Under ITS, it is the only device besides "INT" that does not take all strings as its arguments to OPEN, and it must take an additional optional argument to specify the byte size of the socket. The format of a call to open a network socket is

> <OPEN *mode:string local-socket:fix foreign-socket:fix* "NET" *foreign-host:fix byte-size:fix*>

where:

> *mode* is the mode of the desired CHANNEL. This must be either "READ", "PRINT", "READB" or "PRINTB".

local-socket is the local socket number. If it is -1, the operating system will generate a unique local socket number. If it is not, under TENEX, the socket number is fork-relative.

foreign-socket is the foreign socket number. If it is -1, this is an OPEN for "listening".

foreign-host is the foreign host number. If it is an OPEN for listening, this argument is ignored.

byte-size is the optional byte size. For "READ" or "PRINT" this must be either 7 (default) or 8. For "READB" or "PRINTB", it can be any integer from 1 to 36 (default).

Under TENEX, OPEN can instead be given a STRING argument of the form "NET:...". In this case the local socket number can be directory-relative.

Like any other OPEN, either a CHANNEL or a FALSE is returned. Once open, a network CHANNEL can be used like any other CHANNEL, except that FILE-LENGTH, ACCESS, RENAME, etc., cannot be done. The "argument" first-name, second-name, and directory-name slots in the CHANNEL are used for local socket, foreign socket, and foreign host (as specified in the call to OPEN), respectively. The corresponding "real" slots are used somewhat differently. If a channel is OPENed with local socket -1, the "real" first-name slot will contain the unique socket number generated by the operating system. If a listening socket is OPENed, the foreign socket and host numbers of the answering host are stored in the "real" second-name and directory-name slots of the CHANNEL when the Request For Connection is received.

An interrupt (chapter 21) can be associated with a "NET"-device CHANNEL, so that a program will know that the CHANNEL has or needs data, according to its *mode*.

There also exist several special-purpose SUBRs for the "NET" device. These are described next.

11.11.1. NETSTATE

<NETSTATE *network:channel*>

returns a UVECTOR of three FIXes. The first is the state of the connection, the second is a code specifying why a connection was closed, and the last is the number of bits available on the connection for input. The meaning of the state and close codes are installation-dependent and so are not included here.

11.11.2. NETACC

<NETACC *network:channel*>

accepts a connection to a socket that is open for listening and returns its argument. It will return a FALSE if the connection is in the wrong state.

11.11.3. NETS

<NETS *network:channel*>

returns its argument, after forcing any system-buffered network output to be sent. ITS normally does this every half second anyway. TENEX does not do it unless and until NETS is called. NETS is similar to BUFOUT for normal CHANNELs, except that even operating-system buffers are emptied now.

Chapter 12. Locatives

There is in MDL a facility for obtaining and working directly with objects which roughly correspond to "pointers" in assembly language or "lvals" in BCPL or PAL. In MDL, these are generically known as <u>locatives</u> (from "location") and are of several TYPEs, as mentioned below. Locatives exist to provide efficient means for altering structures: direct replacement as opposed to re-copying.

Locatives <u>always</u> refer to elements in structures. It is not possible to obtain a locative to something (for example, an ATOM) which is not part of any structure. It is possible to obtain a locative to any element in any structured object in MDL -- even to associations (chapter 13) and to the values of ATOMs, structurings which are normally "hidden".

In the following, the object occupying the structured position to which you have obtained a locative will be referred to as the object <u>pointed to</u> by the locative.

12.1. Obtaining Locatives

12.1.1. LLOC

    <LLOC atom env>

returns a locative (TYPE LOCD, "locative to iDentifier") to the LVAL of *atom* in *env*. If *atom* is not bound in *env*, error. *env* is optional, with default being the current ENVIRONMENT. The locative returned by LLOC is <u>independent of future re-bindings</u> of *atom*. That is, IN (see below) of that locative will return the same thing even if *atom* is re-bound to something else; SETLOC (see below) will affect only that particular binding of *atom*.

Since bindings are kept on a stack (tra la), any attempt to use a locative to an LVAL which has become unbound will fetch up an error. (It breaks just like a TUPLE....) LEGAL? can, once again, be used to see if a LOCD is valid. Caution: <SET A <LLOC A>> creates a self-reference and can make PRINT very unhappy.

12.1.2. GLOC

<GLOC *atom pred*>

returns a locative (TYPE LOCD) to the GVAL of *atom.* If *atom* has no GVAL slot, an ERROR occurs, unless *pred* (optional) is given and not FALSE, in which case a slot is created (chapter 22). Caution: <SETG A <GLOC A>> creates a self-reference and can make PRINT very unhappy.

12.1.3. AT

<AT *structured fix*>

returns a locative to the *fix*th element in *structured.* *fix* is optional, default 1. The exact TYPE of the locative returned depends on the PRIMTYPE of *structured*: LOCL for LIST, LOCV for VECTOR, LOCU for UVECTOR, LOCS for STRING, LOCB for BYTES, LOCT for TEMPLATE, and LOCA for TUPLE. If *fix* is greater than <LENGTH *structured*> or less than 1, error. The locative is unaffected by applications of REST, BACK, TOP, GROW, etc. to *structured.*

12.1.4. GETPL and GETL

<GETPL *item:any indicator:any default:any*>

returns a locative (TYPE LOCAS) to the association of *item* under *indicator.* (See chapter 13 for information about associations.) If no such association exists, GETPL returns EVAL of *default. default* is optional, with default #FALSE ().

GETPL corresponds to GETPROP amongst the association machinery. There also exists GETL, which corresponds to GET, returning either a LOCAS or a locative to the *indicator*th element of a structured *item.* GETL is like AT if *item* is a structure and like GETPL if not.

12.2. LOCATIVE?

This SUBR is a predicate that tells whether or not its argument is a locative. It is cheaper than <MEMQ <PRIMTYPE *arg*> '![LOCD LOCL ...]>.

12.3. Using Locatives

The following two SUBRs provide the means for working with locatives. They are independent of the specific TYPE of the locative. The notation *locative* indicates anything which could be returned by LLOC, GLOC, AT, GETPL or GETL.

12.3.1. IN

        <IN *locative*>

returns the object to which *locative* points. The only way you can get an error using IN is when *locative* points to an LVAL which has become unbound from an ATOM. This is the same as the problem in referencing TUPLEs as mentioned in section 9.2, and it can be avoided by first testing <LEGAL? *locd*>.

Example:

        <SET A 1>$
        1
        <IN <LLOC A>>$
        1

12.3.2. SETLOC

        <SETLOC *locative* *any*>

returns *any*, after having made *any* the contents of that position in a structure pointed to by *locative*. The structure itself is not otherwise disturbed. Error if *locative* is to a non-LEGAL? LVAL or if you try to put an object of the wrong TYPE into a PRIMTYPE UVECTOR, STRING, BYTES, or TEMPLATE.

Example:

        <SET A (1 2 3)>$
        (1 2 3)
        <SETLOC <AT .A 2> HI>$
        HI
        .A$
        (1 HI 3)

12.4. Note on Locatives

You may have noticed that locatives are, strictly speaking, unnecessary; you can do everything locatives allow by appropriate use of, for example, SET, LVAL, PUT, NTH, etc. What locatives provide is generality.

Basically, how you obtained a locative is irrelevant to SETLOC and IN; thus the same program can play with GVALs, LVALs, objects in explicit structures, etc., without being bothered by what function it should use to do so. This is particularly true with respect to locatives to LVALs; the fact that they are independent of changes in binding can save a lot of fooling around with EVAL and ENVIRONMENTs.

Chapter 13. Association (Properties)

There is an "associative" data storage and retrieval system embedded in MDL which allows the construction of data structures with arbitrary selectors. It is used via the SUBRs described in this chapter.

13.1. Associative Storage

13.1.1. PUTPROP

　　　　<PUTPROP *item:any indicator:any value:any*>

("put property") returns *item*, having associated *value* with *item* under the indicator *indicator*.

13.1.2. PUT

　　　　<PUT *item:any indicator:any value:any*>

is identical to PUTPROP, except that, if *item* is structured <u>and</u> *indicator* is of TYPE FIX, it does <SETLOC <AT *item indicator*> *value*>. In other words, an element with an integral selector is stored in the structure itself, instead of in association space. PUT (like AT) will get an error if *indicator* is out of range; PUTPROP will not.

13.1.3. Removing Associations

If PUTPROP is used <u>without</u> its *value* argument, it removes any association existing between its *item* argument and its *indicator* argument. If an association did exist, using PUTPROP in this way returns the *value* which was associated. If no association existed, it returns #FALSE ( ).

PUT, with arguments which refer to association, can be used in the same way.

If either *item* or *indicator* cease to exist (that is, no one was pointing to them, so they were garbage-collected), and no locatives to the association exist, then the association between them ceases to exist (is garbage-collected).

13.2. Associative Retrieval

13.2.1. GETPROP

<GETPROP *item:any indicator:any exp:any*>

("get property") returns the *value* associated with *item* under *indicator*, if any. If there is no such association, GETPROP returns EVAL of *exp* (that is, *exp* gets EVALed both at call time and later).

exp is optional. If not given, GETPROP returns #FALSE ( ) if it cannot return a *value*.

Note: *item* and *indicator* in GETPROP must be the <u>same MDL objects</u> used to establish the association; that is, they must be ==? to the objects used by PUTPROP or PUT.

13.2.2. GET

<GET *item:any indicator:any exp:any*>

is the inverse of PUT, using NTH or GETPROP depending on the test outlined in section 13.1.2. *exp* is optional and used as in GETPROP.

13.3. Examples of Association

```
<SET L '(1 2 3 4)>$
(1 2 3 4)
<PUT .L FOO "L is a list.">$
(1 2 3 4)
<GET .L FOO>$
"L is a list."
<PUTPROP .L 3 '![4]>$
(1 2 3 4)
<GETPROP .L 3>$
![4!]
<GET .L 3>$
3
```

```
<SET N 0>$
0
<PUT .N .L "list on a zero">$
0
<GET .N '(1 2 3 4)>$
#FALSE ()
```

The last example failed because READ generated a new LIST -- not the one which is L's LVAL. However,

```
<GET 0 .L>$
"list on a zero"
```

works because <==? .N 0> is true.

To associate something with the Nth position in a structure, as opposed to its Nth element, associate it with <REST *structure* *N-1*>, as in the following:

```
<PUT <REST .L 2> PERCENT 0.3>$
(3 4)
<GET <2 .L> PERCENT>$
#FALSE ()
<GET <REST .L 2> PERCENT>$
0.30000000
```

Remember comments?

```
<SET N '![A B C ;"third element" D E]>$
![A B C D E!]
<GET <REST .N 2> COMMENT>$
"third element"
```

The ' in the <SET N ... > is to keep EVAL from generating a new UVECTOR ("Direct Representation"), which would not have the comment on it (and which would be a needless duplicate). A "top-level" comment -- one attached to the entire object returned by READ -- is PUT on the CHANNEL in use, since there is no position in any structure for it. If no top-level comment follows the object, READ removes the value (<PUT *channel* COMMENT>); so anybody that wants to see a top-level comment must look for it after each READ.

If you need to have a structure with selectors in more than one dimension (for example, a sparse matrix that does not deserve to be linearized), associations can be cascaded to achieve the desired result. In effect an extra level of association maps two indicators into one. For example, to associate *value* with *item* under *indicator-1* and *indicator-2* simultaneously:

```
<PUTPROP indicator-1 indicator-2 T>
```

```
<PUTPROP item <GETPL indicator-1 indicator-2> value>
```

13.4. Examining Associations

Associations (created by PUT and PUTPROP) are chained together in a doubly-linked list, internal to MDL. The order of associations in the chain is their order of creation, newest first. There are several SUBRs for examining the chain of associations. ASSOCIATIONS returns the first association in the chain, or #FALSE ( ) if there are none. NEXT takes an association as an argument and returns the next association in the chain, or #FALSE ( ) if there are no more. ITEM, INDICATOR and AVALUE all take an association as an argument and return the item, indicator and value, respectively. Associations print as:

```
#ASOC (item indicator value)
```

(sic: only one S). Example: the following gathers all the existing associations into a LIST.

```
<PROG ((A <ASSOCIATIONS>))
 <COND (<NOT .A> '())
       (T (.A !<MAPF ,LIST
                   <FUNCTION () <COND (<SET A <NEXT .A>> .A)
                                      (T <MAPSTOP>)>>>)))>
```

Chapter 14. Data-type Declarations

In MDL, it is possible to declare the permissible range of "types" and/or structures that an ATOM's values or a function's arguments or value may have. This is done using a special TYPE, the DECL ("declaration"). A DECL is of PRIMTYPE LIST but has a complicated internal structure. DECLs are used by the interpreter to find TYPE errors in function calling and by the compiler to generate more efficient code.

There are two kinds of DECLs. The first kind of DECL is the most common. It is called the ATOM DECL and is used most commonly to specify the type/structure of the LVALs of the ATOMs in the argument LIST of a FUNCTION or *aux* LIST of a PROG or REPEAT. This DECL has the form:

> #DECL (*atoms:list Pattern* ...)

where the pairing of a LIST of ATOMs and a "Pattern" can be repeated indefinitely. This declares the ATOMs in a *list* to be of the type/structure specified in the following *Pattern*. The special ATOM VALUE, if it appears, declares the result of a FUNCTION call or PROG or REPEAT evaluation to satisfy the Pattern specified. An ATOM DECL is useful in only one place: immediately following the argument LIST of a FUNCTION, PROG or REPEAT. It normally includes ATOMs in the argument LIST and ATOMs whose LVALs are otherwise used in the Function body.

The second kind of DECL is rarely seen by the casual MDL user, except in appendix 2. It is called the RSUBR DECL. It is used to specify the type/structure of the arguments and result of an RSUBR or RSUBR-ENTRY (chapter 19). It is of the following form:

> #DECL ("VALUE" *Pattern Pattern* ...)

where the STRING "VALUE" precedes the specification of the type/structure of the value of the call to the RSUBR, and the remaining *Patterns* specify the arguments to the RSUBR in order. The full specification of the RSUBR DECL will be given in section 14.8. The RSUBR DECL is useful in only one place: as an element of an RSUBR or RSUBR-ENTRY.

14.1. Patterns

The simplest possible Pattern is to say that a value is exactly some other object, by giving that object, QUOTEd. For example, to declare that a variable is a particular ATOM:

    #DECL ((X) 'T)

declares that .X is always the ATOM T. When variables are DECLed as "being" some other object in this way, the test used is =?, and not ==?. The distinction is usually not important, as ATOMs, which are most commonly used in this construction, are ==? to each other if =? anyway.

It is more common to want to specify that a value must be of a given TYPE. This is done with the simplest non-specific Pattern, a TYPE name. For example,

    #DECL ((X) FIX (Y) FLOAT)

declares .X to be of TYPE FIX, and .Y of TYPE FLOAT. In addition to the names of all of the built-in and created TYPEs, such as FIX, FLOAT and LIST, a few "compound" type names are allowed:

ANY allows any TYPE.

STRUCTURED allows any structured TYPE, such as LIST, VECTOR, FALSE, CHANNEL, etc. (appendix 3).

LOCATIVE allows any locative TYPE, such as are returned by LLOC, GLOC, AT, and so on (chapter 12).

APPLICABLE allows any applicable TYPE, such as FUNCTION, SUBR, FIX (!), etc. (appendix 3).

Any other ATOM can be used to stand for a more complex construct, if an association is established on that ATOM and the ATOM DECL. A common example is to <PUT NUMBER DECL '<OR FIX FLOAT>> (see below), so that NUMBER can be used as a "compound type name".

The single TYPE name can be generalized slightly, allowing anything of a given PRIMTYPE, using the following construction:

    #DECL ((X) <PRIMTYPE WORD> (Y) <PRIMTYPE LIST>)

This construction consists of a two-element FORM, where the first element is the ATOM PRIMTYPE, and the second the name of a primitive type.

The next step is to specify the elements of a structure. This is done in the simplest way as follows:

    < structured:type Pattern Pattern ...>

where there is a one-to-one correspondence between the *Patterns* and the elements of the structure. For example:

        #DECL ((X) <VECTOR FIX FLOAT>)

declares .X to be a VECTOR having <u>at least</u> two elements, the first of which is a FIX and the second a FLOAT. It is often convenient to allow additional elements, so that only the elements being used in the local neighborhood of the DECL need to be declared. To disallow additional elements, a SEGMENT is used instead of a FORM (the "excl-ed" brackets make it look more emphatic). For example:

        #DECL ((X) !<VECTOR FIX FLOAT>)

declares .X to be a VECTOR having <u>exactly</u> two elements, the first of which is a FIX and the second a FLOAT. Note that the *Patterns* given for elements can be any legal Pattern:

        #DECL ((X) <VECTOR <VECTOR FIX FLOAT>> (Y) <<PRIMTYPE LIST> LIST>)

declares .X to be a VECTOR containing another VECTOR of at least two elements, and .Y to be of PRIMTYPE LIST, containing a LIST. In the case of a BYTES, the individual elements cannot be declared (they must be FIXes anyway), only the size and number of the bytes:

        #DECL ((B) <BYTES 7 3>)

declares .B to be a BYTES with BYTE-SIZE 7 and at least three elements.

It is possible to say that some number of elements of a structure satisfy a given Pattern (or sequence of Patterns). This is called an "NTH construction".

        [*number:fix Pattern Pattern ...]

states that the sequence of *Patterns* which is REST of the VECTOR is repeated the *number* of times given. For example:

        #DECL ((X) <VECTOR [3 FIX] FLOAT> (Y) <LIST [3 FIX FLOAT]>)

.X is declared to contain three FIXes and a FLOAT, perhaps followed by other elements. .Y is declared to repeat the sequence FIX-FLOAT three times. Note that there may be more repetitions of the sequence in .Y (but not in .X): the DECL specifies only the first six elements.

For indefinite repetition, the same construction is used, but, instead of the number of repetitions of the sequence of Patterns, the ATOM REST is given. This allows any number of repetitions, from zero on up. For example:

        #DECL ((X) <VECTOR [REST FIX]> (Y) <LIST [3 FIX] [REST FIX]>

A "REST construction" can contain any number of Patterns, just like an NTH construction:

> #DECL ((X) <VECTOR [REST FIX FLOAT LIST]>)

declares that .X is a VECTOR wherein the sequence FIX-FLOAT-LIST repeats indefinitely. It does not declare that <LENGTH .X> is an even multiple of three; the VECTOR can end at any point.

A variation on REST is OPT (or OPTIONAL), which is similar to REST except that the construction is scanned once at most instead of indefinitely, and further undeclared elements can follow. For example:

> #DECL ((X) <VECTOR [OPT FIX]>)

declares that .X is a VECTOR which is empty or whose first element is a FIX. Only a REST construction can follow an "OPT construction".

Note that the REST construction must always be the last element of the structure declaration, as it gives a Pattern for the rest of the structure. Thus, the REST construction is different from all others in that it has an unlimited range. No matter how many times the Pattern it gives is RESTed off of the structure, the remainder of the structure still has that Pattern.

This exhausts the possible single Patterns that can be given in a declaration. However, there is also a compound Pattern defined. It allows specification of several possible Patterns for one value:

> <OR *Pattern Pattern* ... >

Any non-compound Pattern can be included as one of the elements of the compound Pattern. Finally, compound Patterns can be used as Patterns for elements of structures, and so on.

> #DECL ((X) <OR FIX FLOAT>
>   (Y) <OR FIX <UVECTOR [REST <OR FIX FLOAT>]>>)

The OR construction can be extended to any level of ridiculousness, but the higher the level of complexity and compoundedness the less likely the compiler will find the DECL useful.

At the highest level, any Pattern at top level in an ATOM DECL can be enclosed in the construction

> < *speciality:atom Pattern* >

which explicitly declares the speciality of the ATOM(s) in the preceding LIST. *speciality* can be either SPECIAL or UNSPECIAL. Speciality is important only when the program is to be compiled. The word comes from the control stack, which is called "special" in Lisp because the garbage collector finds objects on it and modifies their internal pointers when storage is compacted. (An internal stack is used within the interpreter and is not accessible to programs -- section 22.1.) In an interpreted

program all local values are initially SPECIAL, because all bindings are put on the control stack (but see SPECIAL-MODE below). When the program is compiled, only values declared SPECIAL (which may or may not be the default) remain in bindings on the control stack. All others are taken care of simply by storing objects on the control stack: the ATOMs involved are not needed and are not created on loading. So, a program that SETs an ATOM's local value for another program to pick up must declare that ATOM to be SPECIAL. If it doesn't, the ATOM's binding will go away during compiling, and the program that needs to access the ATOM will get a no-value error or access an erroneous binding. Usually only ATOMs which have the opposite speciality from that of the current SPECIAL-MODE are explicitly declared. The usual SPECIAL-MODE is UNSPECIAL, so typically only SPECIAL declarations use this construction:

>   #DECL ((X) <SPECIAL <VECTOR FIX FIX>>)

explicitly declares X to be SPECIAL.

14.2. Examples

>   #DECL ((Q) <OR VECTOR CHANNEL>)

declares .Q to be either a VECTOR or a CHANNEL.

>   #DECL ((P Q R S) <PRIMTYPE LIST>)

declares .P, .Q, .R, and .S all to be of PRIMTYPE LIST.

>   #DECL ((F) <FORM [3 ANY]>)

declares .F to be a FORM whose length is at least three, containing objects of any old TYPE.

>   #DECL ((LL) <<PRIMTYPE LIST> [4 <LIST [REST FIX]>]>)

declares .LL to be of PRIMTYPE LIST, and to have at least four elements, each of which are LISTs of unspecified length (possibly empty) containing FIXes.

>   #DECL ((VV) <VECTOR FIX ATOM CHARACTER>)

declares .VV to be a VECTOR with at least three elements. Those elements are, in order, of TYPE FIX, ATOM, and CHARACTER.

>   #DECL ((LH) <LIST ATOM [REST FLOAT]>)

declares .LH to be a LIST whose first element is an ATOM and the rest of whose elements are FLOATs. It also says that .LH is at least one element long.

```
#DECL ((FOO) <LIST [REST 'T FIX]>)
```

declares .FOO to be a LIST whose odd-positioned elements are the ATOM T and whose even-positioned elements are FIXes.

```
<MAPR <>
        <FUNCTION (X)
          #DECL ((X) <VECTOR [1 FIX]>)
          <PUT .X 1 0>>
        .FOO>
```

declares .X to be a VECTOR containing at least one FIX. The more restrictive [REST FIX] would take excessive checking time by the interpreter, because the REST of the VECTOR would be checked on each iteration of the MAPR. In this case both DECLs are equally powerful, because checking the first element of all the RESTs of a structure eventually checks all the elements. Also, since the FUNCTION refers only to the first element of X, this is as much declaration as the compiler can effectively use. (If this VECTOR always contains only FIXes, it should be a UVECTOR instead, for space efficiency. Then a [REST FIX] DECL would make the interpreter check only the UTYPE. If the FIXes cover a small non-negative range, then a BYTES might be even better, with a DECL of <BYTES n 0>.)

```
<DEFINE FACT (N)
        #DECL ((N) <UNSPECIAL FIX>)
        <COND (<0? .N> 1) (ELSE <* .N <FACT <- .N 1>>>)>>
```

declares .N to be of TYPE FIX and UNSPECIAL. This speciality declaration ensures that, independent of SPECIAL-MODE during compiling, .N gets compiled into a fast control-stack access.

```
<PROG ((L (0))
        #DECL ((L VALUE) <UNSPECIAL <LIST [REST FIX]>>
               (N) <UNSPECIAL FIX>)
        <COND (<0? .N> <RETURN .L>)>
        <SET L (<+ .N <1 .L>> !.L)>
        <SET N <- .N 1>>>
```

The above declares L and N to be UNSPECIAL, says that .N is a FIX, and says that .L, along with the value returned, is a LIST of any length composed entirely of FIXes.

14.3. The DECL Syntax

This section gives quasi-BNF productions for the MDL DECL syntax. In the following table MDL type-specifiers are distinguished *in this way*.

```
decl    ::=    #DECL (declprs)

declprs ::=    (atlist) pattern | declprs declprs

atlist  ::=    atom | atom atlist

pattern ::=    pat | <UNSPECIAL pat> | <SPECIAL pat>

pat     ::=    unit | <OR unit ... unit>

unit    ::=    type | <PRIMTYPE type> | atom | 'any
               | ANY | STRUCTURED | LOCATIVE | APPLICABLE
               | <struc elts> | <<OR struc ... struc> elts>
               | !<struc elts> | !<<OR struc ... struc> elts>
               | <BYTES fix fix> | !<BYTES fix fix>

struc   ::=    structured-type | <PRIMTYPE structured-type>

elts    ::=    pat | pat elts
               | [fix pat ... pat]
               | [fix pat ... pat] elts
               | [opt pat ... pat] | [REST pat ... pat]
               | [opt pat ... pat] [REST pat ... pat]

opt     ::=    OPT | OPTIONAL
```

14.4. Good DECLs

There are some rules of thumb concerning "good" DECLs. A "good" DECL is one that is minimally offensive to the DECL-checking mechanism and the compiler, but that gives the maximum amount of information. It is simple to state what gives offense to the compiler and DECL-checking mechanism: complexity. For example, a large compound DECL like:

```
#DECL ((X) <OR FIX LIST UVECTOR FALSE>)
```

is a DECL that the compiler will find totally useless. It might as well be ANY. The more involved the OR, the less information the compiler will find useful in it. For example, if the function takes <OR LIST VECTOR UVECTOR>, maybe you should really say STRUCTURED. Also, a very general DECL indicates a very general program, which is not likely to be efficient when compiled (of course there is a tradeoff here). Narrowing the DECL to one PRIMTYPE gives a great gain in compiled efficiency, to one TYPE still more.

Another situation to be avoided is the ordinary large DECL, even if it is perfectly straightforward. If you have created a structure which has a very specific DECL and is used all over your code, it might be better as a NEWTYPE (see below). The advantage of a NEWTYPE over a large explicit DECL is twofold. First, the entire structure must be checked only when it is created, that is, CHTYPEd from its PRIMTYPE. As a full DECL, it is checked completely on entering each function and on each reassignment of ATOMs DECLed to be it. Second, the amount of storage saved in the DECLs of FUNCTIONs and so on is large, not to mention the effort of typing in and keeping up to date several instances of the full DECL.

14.5. Global DECLs

14.5.1. GDECL and MANIFEST

There are two ways to declare GVALs for the DECL-checking mechanism. These are through the FSUBR GDECL ("global declaration") and the SUBR MANIFEST.

        <GDECL atoms:list Pattern ...>

GDECL allows the type/structure of global values to be declared in much the same way as local values. Example:

        <GDECL (X) FIX (Y) <LIST FIX>>

declares ,X to be a FIX, and ,Y to be a LIST containing at least one FIX.

        <MANIFEST atom atom ...>

MANIFEST takes as arguments ATOMs whose GVALs are declared to be constants. It is used most commonly to indicate that certain ATOMs are the names of offsets in structures. For example:

        <SETG X 1>
        <MANIFEST X>

allows the compiler to confidently open-compile applications of X (getting the first element of a structure), knowing that ,X will not change. Any sort of object can be a MANIFEST value; if it does not get embedded in the compiled code, it is included in the RSUBR's "reference vector", for fast access. However, as a general rule, structured objects should not be made MANIFEST: the SETG will survive in the compiled version (for the use of new uncompiled programs), but uses of GVAL will instead refer to a distinct copy of the object in each RSUBR that does a GVAL. A structured object should instead be GDECLed.

An attempt to SETG a MANIFEST ATOM will cause an ERROR, unless either:

(1) the ATOM was previously globally unassigned;

(2) the old value is ==? to the new value; or

(3) .REDEFINE is not FALSE.

14.5.2. MANIFEST? and UNMANIFEST

        <MANIFEST? *atom*>

returns T if *atom* is MANIFEST, #FALSE ( ) otherwise.

        <UNMANIFEST atom atom ...>

removes the MANIFEST of the global value of each of its arguments so that the value can be changed.

14.5.3. GBOUND?

        <GBOUND? *atom*>

("globally bound?") returns T if *atom* has a global value slot (that is, if it has ever been SETGed, MANIFEST, GDECLed, or GLOCed (chapter 12) with a true second argument), #FALSE ( ) otherwise.

14.6. NEWTYPE (again)

NEWTYPE gives the programmer another way to DECL objects. The third (and optional) argument of NEWTYPE is a QUOTEd Pattern. If given, it will be saved as the value of an association (chapter 13) using the name of the NEWTYPE as the item and the ATOM DECL as the indicator, and it will be used to check any object that is about to be CHTYPEd to the NEWTYPE. For example:

        <NEWTYPE COMPLEX-NUMBER VECTOR '<<PRIMTYPE VECTOR> FLOAT FLOAT>>

creates a new TYPE, with its first two elements declared to be FLOATs. If later someone types:

        #COMPLEX-NUMBER [1.0 2]

an ERROR will result (the second element is not a FLOAT). The Pattern can be replaced by doing another NEWTYPE for the same TYPE, or by putting a new value in the association. Further examples:

        <NEWTYPE FOO LIST '<<PRIMTYPE LIST> FIX FLOAT [REST ATOM]>>

causes FOOs to contain a FIX and a FLOAT and any number of ATOMs.

```
<NEWTYPE BAR LIST>

<SET A #BAR (#BAR () 1 1.2 GRITCH)>

<NEWTYPE BAR LIST '<<PRIMTYPE LIST> BAR [REST FIX FLOAT ATOM]>>
```

This is an example of a recursively DECLed TYPE. Note that <1 .A> does not satisfy the DECL, because it is empty, but it was CHTYPEd before the DECL was associated with BAR. Now, even <CHTYPE <1 .A> <TYPE <1 .A>>> will cause an error.

In each of these examples, the <<PRIMTYPE ...> ...> construction was used, in order to permit CHTYPEing an object into itself. See what happens otherwise:

```
<NEWTYPE OOPS LIST '<LIST ATOM FLOAT>>$
OOPS
<SET A <CHTYPE (E 2.71828) OOPS>>$
#OOPS (E 2.71828)
```

Now <CHTYPE .A OOPS> will cause an error. Unfortunately, you must

```
<CHTYPE <CHTYPE .A LIST> OOPS>$
#OOPS (E 2.71828)
```

14.7. Controlling DECL Checking

There are several SUBRs and FSUBRs in MDL that are used to control and interact with the DECL-checking mechanism.

14.7.1. DECL-CHECK

This entire complex checking mechanism can get in the way during debugging. As a result, the most commonly used DECL-oriented SUBR is DECL-CHECK. It is used to enable and disable the entire DECL-checking mechanism.

```
<DECL-CHECK false-or-any>
```

If its single argument is non-FALSE, DECL checking is turned on; if it is FALSE, DECL checking is turned off. The previous state is returned as a value. If no argument is given, DECL-CHECK returns the current state. In an initial MDL DECL checking is on.

When DECL checking is on, the DECL of an ATOM is checked each time it is SET, the arguments and

results of calls to FUNCTIONs, RSUBRs, and RSUBR-ENTRYs are checked, and the values returned by PROG and REPEAT are checked. The same is done for SETGs and, in particular, attempts to change MANIFEST global values. Attempts to CHTYPE an object to a NEWTYPE (if the NEWTYPE has the optional DECL) are also checked. When DECL checking is off, none of these checks is performed.

14.7.2. SPECIAL-CHECK and SPECIAL-MODE

        `<SPECIAL-CHECK` *false-or-any*`>`

controls whether or not SPECIAL checking is performed at run time by the interpreter. It is initially off. Failure to declare an ATOM to be SPECIAL when it should be will produce buggy compiled code.

        `<SPECIAL-MODE` *speciality:atom*`>`

sets the default mode (for ATOMs not declared either way) and returns the previous default, or the current default if no argument is given. The initial default is UNSPECIAL.

14.7.3. GET-DECL and PUT-DECL

GET-DECL and PUT-DECL are used to examine and change the current DECL (of either the global or the local value) of an ATOM.

        `<GET-DECL` *locd*`>`

returns the DECL Pattern (if any, otherwise #FALSE ( )) associated with the global or local value slot of an ATOM. For example:

```
<PROG (X)
      #DECL ((X) <OR FIX FLOAT>)
      ...
      <GET-DECL <LLOC X>>
      ...>
```

would return `<OR FIX FLOAT>` as the result of the application of GET-DECL. Note that because of the use of LLOC (or GLOC, for globals) the ATOM being examined must be bound, otherwise you will get an error! This can be gotten around by testing first with BOUND? (or GBOUND?, or by giving GLOC a second argument which is not FALSE).

If the slot being examined is the global slot and the value is MANIFEST, then the ATOM MANIFEST is returned. If the value being examined is not DECLed, #FALSE ( ) is returned.

        `<PUT-DECL` *locd* '*Pattern*`>`

makes *Pattern* be the DECL for the value and returns *locd*. If <DECL-CHECK> is true, the current value must satisfy the new Pattern. PUT-DECL is normally used in debugging, to change the DECL of an object to correspond to changes in the program. Note that it is not legal to PUT-DECL a "Pattern" of MANIFEST or #FALSE ( ).

14.7.4. DECL?

```
<DECL? any 'Pattern>
```

specifically checks *any* against *Pattern*. For example:

```
<DECL? '[1 2 3] '<VECTOR [REST FIX]>>$
T
<DECL? '[1 2.0 3.0] '<VECTOR [REST FIX]>>$
#FALSE ()
```

14.8. The RSUBR DECL

The RSUBR DECL is similar to the ATOM DECL, except that the declarations are of argument positions and value rather than of specific ATOMs. Patterns can be preceded by STRINGs which further describe the argument (or value).

The simplest RSUBR DECL is for an RSUBR or RSUBR-ENTRY (chapter 19) which has all of its arguments evaluated and returns a DECLed value. For example:

```
#DECL ("VALUE" FIX FIX FLOAT)
```

declares that there are two arguments, a FIX and a FLOAT, and a result which is a FIX. While the STRING "VALUE" is not constrained to appear at the front of the DECL, it does appear there by custom. It need not appear at all, if the result is not to be declared, but (again by custom) in this case it is usually declared ANY.

If any arguments are optional, the STRING "OPTIONAL" (or "OPT") is placed before the Pattern for the first optional argument:

```
#DECL ("VALUE" FIX FIX "OPTIONAL" FLOAT)
```

If any of the arguments is not to be evaluated, it is preceded by the STRING "QUOTE":

```
#DECL ("VALUE" FIX "QUOTE" FORM)
```

declares one argument, which is not EVALed.

If the arguments are to be evaluated and gathered into a TUPLE, the Pattern for it is preceded by the STRING "TUPLE":

        #DECL ("VALUE" FIX "TUPLE" <TUPLE [REST FIX]>)

If the arguments are to be unevaluated and gathered into a LIST, or if the calling FORM is the only "argument", the Pattern is preceded by the appropriate STRING:

        #DECL ("VALUE" FIX "ARGS" LIST)

        #DECL ("VALUE" FIX "CALL" <PRIMTYPE LIST>)

In every case the special indicator STRING is followed by a Pattern which describes the argument, even though it may sometimes produce fairly ludicrous results, since the Pattern for "TUPLE" always must be a TUPLE; for "ARGS", a LIST; and for "CALL", a FORM or SEGMENT.

Chapter 15. Lexical Blocking

Lexical, or static, blocking is another means of preventing identifier collisions in MDL. (The first was dynamic blocking -- binding and ENVIRONMENTs.) By using a subset of the MDL lexical blocking facilities, the "block structure" of such languages as Algol, PL/I, SAIL, etc., can be simulated, should you wish to do so.

15.1. Basic Considerations

Since what follows appears to be rather complex, a short discussion of the basic problem lexical blocking solves and MDL's basic solution will be given first.

ATOMs are identifiers. It is thus essential that whenever you type an ATOM, READ should respond with the unique identifier you wish to designate. The problem is that it is unreasonable to expect the PNAMEs of all ATOMs to be unique. When you use an ATOM A in a program, do you mean the A you typed two minutes ago, the A you used in another one of your programs, or the A used by some library program?

Dynamic blocking (pushdown of LVALs) solves many such problems. However, there are some which it does not solve -- such as state variables (whether impure or pure). Major problems with a system having only dynamic blocking usually arise only when attempts are made to share large numbers of significant programs among many people.

The solution used in MDL is basically as follows: READ must maintain at least one table of ATOMs to guarantee any uniqueness. So, allow many such tables, and make it easy for the user to specify which one is wanted. Such a table is an object of TYPE OBLIST ("object list"). All the complication which follows arises out of a desire to provide a powerful, easily used method of working with OBLISTs, with good defaults.

15.2. OBLISTs

An OBLIST is of PRIMTYPE UVECTOR with UTYPE LIST; the LISTs hold ATOMs. (The ATOMs are ordered by a hash coding on their PNAMEs; each LIST is a hashing bucket.) What follows is information about OBLISTs as such.

15.2.1. OBLIST Names

Every normally constituted OBLIST has a name. The name of an OBLIST is an ATOM associated with the OBLIST under the indicator OBLIST. Thus,

        <GETPROP *oblist* OBLIST>

or

        <GET *oblist* OBLIST>

returns the name of *oblist*.

Similarly, every name of an OBLIST is associated with its OBLIST, again under the indicator OBLIST, so that

        <GETPROP *oblist-name:atom* OBLIST>

or

        <GET *oblist-name:atom* OBLIST>

returns the OBLIST whose name is *oblist-name*.

Since there is nothing special about the association of OBLISTs and their names, the name of an OBLIST can be changed by use of PUTPROP, both on the OBLIST and its name. It is not wise to change the OBLIST association without changing the name association, as you are likely to confuse READ and PRINT terribly.

You can also use PUT or PUTPROP to remove the association between an OBLIST and its name completely. If you want the OBLIST to go away (be garbage collected), and you want to keep its name around, this must be done; otherwise the association will force it to stay, even if there are no other references to it. (If you have no references to either the name or the OBLIST (an ATOM -- including a TYPE name -- points to its OBLIST), both of them -- and their association -- will go away without your having to remove the association, of course.) It is not recommended that you remove the name of an OBLIST without having it go away, since then ATOMs in that OBLIST will PRINT the same as if they were in no OBLIST -- which is defeating the purpose of this whole exercise.

15.2.2. MOBLIST

<MOBLIST *atom fix*>

("make oblist") creates and returns a new OBLIST, containing no ATOMs, whose name is *atom*, unless there already exists an OBLIST of that name, in which case it returns the existing OBLIST. *fix* is the size of the OBLIST created -- the number of hashing buckets. *fix* is optional (ignored if the OBLIST already exists), with default 13. If specified, *fix* should be a prime number, since that allows the hashing to work better.

15.2.3. OBLIST?

<OBLIST? *atom*>

returns #FALSE ( ) if *atom* is not in any OBLIST. If *atom* is in an OBLIST, it returns that OBLIST.

15.3. READ and OBLISTs

15.3.1. Trailers

READ can be explicitly told to look up an ATOM in a particular OBLIST by giving the ATOM a <u>trailer</u>. A trailer consists of the characters !- (exclamation-point dash) following the ATOM, immediately followed by the name of the OBLIST. For example,

A!-OB

specifies the unique ATOM of PNAME A which is in the OBLIST whose name is the ATOM OB.

Note that the name of the OBLIST must follow the !- with <u>no</u> separators (like space, tab, carriage-return, etc.). There is a default name (section 15.5) which types and is typed as !-*separator*.

Trailers can be used recursively:

B!-A!-OB

specifies the unique ATOM of PNAME B which is in the OBLIST whose name is the unique ATOM of PNAME A which is in the OBLIST whose name is OB. (Whew!) The recursion is terminated via the defaults described below.

If an ATOM with a given PNAME is not found in the OBLIST specified by a trailer, a new ATOM with that PNAME is created and inserted into that OBLIST.

If an OBLIST whose name is given in a trailer does not exist, READ creates one, of length 13 buckets.

15.3.2. READ and Defaults

If trailer notation is not used (the "normal" case), and for an ATOM that terminates a trailer, READ looks up the PNAME of the ATOM in a LIST of OBLISTs, with default the LVAL of the ATOM OBLIST. This lookup starts with <1 .OBLIST> and continues until .OBLIST is exhausted. If the ATOM is not found, READ usually inserts it into <1 .OBLIST>. (It is possible to force READ to use a different element of the LIST of OBLISTs for new insertions. If the ATOM DEFAULT is in that LIST, the OBLIST following that ATOM will be used.)

15.4. PRINT and OBLISTs

When PRINT is given an ATOM to output, it outputs as little of the trailer as is necessary to specify the ATOM uniquely to READ. That is, if the ATOM is the first ATOM of that PNAME which READ would find in its normal lookup in the current .OBLIST, no trailer is output. Otherwise, !- is output and the name of the OBLIST is recursively PRINled.

Warning: there are obscure cases, which do not occur in normal practice, for which the PRINT trailer recursion does not terminate. For instance, if an ATOM must have a trailer printed, and the name of the OBLIST is an ATOM in that very same OBLIST, death. Any similar circular case will also give PRINT a hernia.

15.5. Initial State

In an initial MDL, .OBLIST contains two OBLISTs. <1 .OBLIST> initially contains no ATOMs, and <2 .OBLIST> contains all the ATOMs whose GVALs are SUBRs or FSUBRs, as well as OBLIST, DEFAULT, T, etc. It is difficult to lose track of the latter; the specific trailer !-separator will always cause reference to that OBLIST. In addition, the SUBR ROOT, which takes no arguments, always returns that OBLIST.

The name of <ROOT> is ROOT; this ATOM is in <ROOT> and would cause infinite PRINT recursion were it not for the use of !-separator. The name of the initial <1 .OBLIST> is INITIAL (really INITIAL!-).

The ATOM OBLIST also has a GVAL ,OBLIST is initially the same as .OBLIST; however, ,OBLIST is not affected by the SUBRs used to manipulate the OBLIST structure. It is instead used only when errors occur.

In the case of an ERROR, the current .OBLIST is checked to see if it is "reasonable" -- that is, contains nothing of the wrong TYPE. (It is reasonable, but not standard, for .OBLIST to be a single OBLIST instead of a LIST of them.) If it is reasonable, that value stays current. Otherwise, OBLIST is SET to ,OBLIST. Note that changes made to the OBLISTs on ,OBLIST -- for example, new ATOMs added -- remain. If even ,OBLIST is unreasonable, OBLIST is SET and SETGed to its initial value. <ERRET> (section 16.4) always assumes that .OBLIST is unreasonable.

Three other OBLISTs exist in a virgin MDL; their names and purposes are as follows:

ERRORS!- contains ATOMs whose PNAMEs are used as error messages. It is returned by <ERRORS>.

INTERRUPTS!- is used by the interrupt system (section 21.2.1). It is returned by <INTERRUPTS>.

MUDDLE!- is used infrequently by the interpreter when loading compiled programs to fix up references to locations within the interpreter.

The preloading of compiled programs may create other OBLISTs in an initialized MDL (ref 4).

15.6. BLOCK and ENDBLOCK

These SUBRs are analogous to begin and end in Algol, etc., in the way they manipulate static blocking (and in no other way).

        <BLOCK list-of-oblists>

returns its argument after "pushing" the current LVAL of the ATOM OBLIST and making its argument the current LVAL. You usually want <ROOT> to be an element of list-of-oblists, normally its last.

        <ENDBLOCK>

"pops" the LVAL of the ATOM OBLIST and returns the resultant LIST of OBLISTs.

Note that this OBLIST "pushing" and "popping" is entirely independent of functional application, binding, etc.

15.7. SUBRs Associated with Lexical Blocking

15.7.1. READ (again)

                    <READ *channel eof-routine lookup*>

This is a fuller call to READ. *lookup* is an OBLIST or a LIST of them, used as stated in section 15.3 to look up ATOMs and insert them in OBLISTs. If it is not specified, .OBLIST is used. See also sections 11.1.1.1, 11.3, and 17.1.3 for other arguments.

15.7.2. PARSE and LPARSE (again)

                    <PARSE *string radix:fix lookup*>

as was previously mentioned, applies READ's algorithm to *string* and returns the first MDL object resulting. This includes looking up prospective ATOMs on *lookup*, if given, or .OBLIST. LPARSE can be called in the same way. See also sections 7.6.6.2 and 17.1.3 for other arguments.

15.7.3. LOOKUP

                    <LOOKUP *string oblist*>

returns the ATOM of PNAME *string* in the OBLIST *oblist*, if there is such an ATOM; otherwise, it returns #FALSE (). If *string* would PARSE into an ATOM anyway, LOOKUP is faster, although it looks in only one OBLIST instead of a LIST of them.

15.7.4. ATOM

                    <ATOM *string*>

creates and returns a spanking new ATOM of PNAME *string* which is guaranteed not to be on any OBLIST.

An ATOM which is not on any OBLIST is PRINTed with a trailer of !-#FALSE ( ).

15.7.5. REMOVE

                    <REMOVE *string oblist*>

removes the ATOM of PNAME *string* from *oblist* and returns that ATOM. If there is no such ATOM, REMOVE returns #FALSE ( ). Also,

<REMOVE *atom*>

removes *atom* from its OBLIST, if it is on one. It returns *atom* if it was on an OBLIST; otherwise it returns #FALSE ( ).

15.7.6. INSERT

<INSERT *string-or-atom oblist*>

creates an ATOM of PNAME *string*, inserts it into *oblist* and returns it. If there is already an ATOM with the same PNAME as *atom* in *oblist*, error. The standard way to avoid the error and always get your *atom* is

<OR <LOOKUP *string oblist*> <INSERT *string oblist*>>

As with REMOVE, INSERT can also take an ATOM as its first argument; this ATOM must not be on any OBLIST -- it must have been REMOVEd, or just created by ATOM -- else error. The OBLIST argument is never optional. If you would like the new ATOM to live in the OBLIST that READ would have chosen, you can <PARSE *string*> instead.

15.7.7. PNAME

<PNAME *atom*>

returns a STRING (newly created) which is *atom*'s PNAME ("printed name"). If trailers are not needed, PNAME is much faster than UNPARSE on *atom*. (In fact UNPARSE has to go all the way through the PRINT algorithm twice, the first time to see how long a STRING is needed.)

15.7.8. SPNAME

SPNAME ("shared printed name") is identical to PNAME, except that the STRING it returns shares storage with *atom* (appendix 1), which is more efficient if the STRING will not be modified. PUTting into such a STRING will cause an ERROR.

15.8. Example: Another Solution to the INC Problem

What follows is an example of the way OBLISTs are "normally" used to provide "externally available" ATOMs and "local" ATOMs which are not so readily available externally. Ref 8 describes a systematic way to accomplish the same thing and more.

```
<MOBLIST INCO 1>
            ;"Create an OBLIST to hold your external symbols.
            Its name is INCO!-INITIAL!- ."

INC!-INCO
            ;"Put your external symbols into that OBLIST.
            If you have many, just write them successively."

<BLOCK (<MOBLIST INCI!-INCO 1> <GET INCO OBLIST> <ROOT>)>
            ;"Create a local OBLIST, naming it INCI!-INCO, and set up .OBLIST for
            reading in your program.  The OBLIST INCO is included in the BLOCK so
            that as your external symbols are used, they will be found in the
            right place.  Note that the ATOM INCO is not in any OBLIST of the
            BLOCK; therefore, trailer notation of !-INCO will not work within the
            current BLOCK-ENDBLOCK pair."

<DEFINE INC      ;"INC is found in the INCO OBLIST."
        (A)      ;"A is not found and is therefore put into INCI by READ."
        #DECL ((VALUE A) <OR FIX FLOAT>)
        <SET .A <+ ..A 1>>>      ;"All other ATOMs are found in the ROOT."
<ENDBLOCK>
```

This example is rather trivial, but it contains all the issues, of which there are three.

The first idea is that you should create two OBLISTs, one to hold ATOMs which are to be known to other users (INCO), and the other to hold internal ATOMs which are not normally of interest to others (INCI). The case above has one ATOM in each category.

Second, INCO is explicitly used without trailers so that surrounding BLOCKs and ENDBLOCKs will have an effect on it. Thus INCO will be in the OBLIST desired by the user; INC will be in INCO, and the user can access it by saying INC!-INCO; INCI will also be in INCO, and can be accessed in the same way; finally, A is really A!-INCI!-INCO. The point of all this is to structure the nesting of OBLISTs.

Finally, if for some reason (like saving storage space) you wish to throw INCI away, you can follow the ENDBLOCK with

```
<REMOVE "INCI" <GET INCO OBLIST>>
```

and thus remove all references to it. The ability to do such pruning is one reason for structuring OBLIST references.

Note that, even after removing INCI, you can "get A back" -- that is, be able to type it in -- by saying something of the form

```
<INSERT <1 <1 ,INC!-INCO>> <1 .OBLIST>>
```

thereby grabbing A out of the structure of INC and re-inserting it into an OBLIST. However, this resurrects the name collision caused by <INCI-INCO A>.

Chapter 16. Errors, Frames, etc.

16.1. LISTEN

This SUBR takes any number of arguments. It first checks the LVALs of INCHAN, OUTCHAN, and OBLIST for reasonability and console usability. In each case, if the value is unreasonable, the ATOM is rebound to the corresponding GVAL, if reasonable, or to an invented reasonable value. LISTEN then does <TTYECHO .INCHAN T> and <ECHOPAIR .INCHAN .OUTCHAN>. Next, it PRINTs its arguments, then PRINTs

    LISTENING-AT-LEVEL / PROCESS p

where / is an integer (FIX) which is incremented each time LISTEN is called recursively, and p is an integer identifying the PROCESS (chapter 20) in which the LISTEN was EVALed. LISTEN then does <APPLY <VALUE REP>>, if there is one, and if it is APPLICABLE. If not, it applies the SUBR REP (without making a new FRAME — see below). This SUBR drops into an infinite READ-EVAL-PRINT loop, which can be left via ERRET (section 16.4).

The standard LISTEN loop has two features for getting a handle on objects that you have typed in and MDL has typed out. If the ATOM L-INS has a local value that is a LIST, LISTEN will keep recent inputs (what READ returns) in it, most recent first. Similarly, if the ATOM L-OUTS has a local value that is a LIST, LISTEN will keep recent outputs (what EVAL returns) in it, most recent first. The keeping is done before the PRINTing, so that ^S does not defeat its purpose. The user can decide how much to keep around by setting the length of each LIST. Even if L-OUTS is not used, the atom LAST-OUT is always SET to the last object returned by EVAL in the standard LISTEN loop. Example:

```
<SET L-INS (RECENTEST RECENTER RECENT)>$
(RECENTEST RECENTER RECENT)
.L-INS$
(.L-INS RECENTEST RECENTER)
<SET FOO 69>$
69
<SET FIXIT <2 .L-INS>>          ;"grab the last input"$
<SET FOO 69>
```

```
.L-INS$
(.L-INS <SET FIXIT <2 .L-INS>> <SET FOO 69>)
<PUT .FIXIT 3 105>$
<SET FOO 105>
<EVAL .FIXIT>$
105
.L-INS$
(.L-INS <EVAL .FIXIT> <PUT .FIXIT 3 105>)
.FOO$
105
```

16.2. ERROR

This SUBR is the same as LISTEN, except that (1) it generates an interrupt (chapter 21), if enabled, and (2) it PRINTs *ERROR* before PRINTing its arguments.

When any SUBR or FSUBR detects an anomalous condition (for example, its arguments are of the wrong TYPE), it calls ERROR with at least two arguments, including:

(1) an ATOM whose PNAME describes the problem, normally from the OBLIST ERRORS!- (appendix 4),

(2) the ATOM that names the SUBR or FSUBR, and

(3) any other information of interest,

and then returns whatever the call to ERROR returns. Exception: a few (for example DEFINE) will take further action that depends on the value returned. This nonstandard action is specified in the error message (first ERROR argument).

16.3. FRAME (the TYPE)

A FRAME is the object placed on a PROCESS's control stack (chapter 20) whenever a SUBR, FSUBR, RSUBR, or RSUBR-ENTRY (chapter 19) is applied. (These objects are herein collectively called "Subroutines".) It contains information describing what was applied, plus a TUPLE whose elements are the arguments to the Subroutine applied. If any of the Subroutine's arguments are to be evaluated, they will have been by the time the FRAME is generated.

A FRAME is an anomalous TYPE in the following ways:

(1) It cannot be typed in. It can be generated only by an application.

(2) It does not type out in any standard format, but rather as #FRAME followed by the PNAME of the Subroutine applied.

16.3.1. ARGS

      `<ARGS frame>`

("arguments") returns the argument TUPLE of *frame*.

16.3.2. FUNCT

      `<FUNCT frame>`

("function") returns the ATOM whose G/LVAL is being applied in *frame*.

16.3.3. FRAME (the SUBR)

      `<FRAME frame>`

returns the FRAME stacked <u>before</u> *frame* or, if there is none, it will generate an ERROR. The lowest FRAME that can be returned without error has a FUNCT of TOPLEVEL. If called with no arguments, FRAME returns the topmost FRAME used in an application of ERROR or LISTEN, which was bound by the interpreter to the ATOM LERR\ !-INTERRUPTS ("last error").

16.3.4. Examples

Say you have gotten an ERROR. You can now type at ERROR's LISTEN loop and get things EVALed. For example,

      `<FUNCT <FRAME>>$`
      `ERROR`
      `<FUNCT <FRAME <FRAME>>>$`
      the-name-of-the-Subroutine-which-called-ERROR:atom
      `<ARGS <FRAME <FRAME>>>$`
      the-arguments-to-the-Subroutine-which-called-ERROR:tuple

16.4. ERRET

      `<ERRET any frame>`

This SUBR ("error return") (1) causes the control stack to be stripped down to the level of *frame*, and (2) <u>then</u> returns *any*. The net result is that the application which generated *frame* is forced to return

any. Additional side effects that would have happened in the absence of an error may not have happened.

The second argument to ERRET is optional, with default the FRAME of the last invocation of ERROR or LISTEN.

If ERRET is called with no arguments, it drops you all the way down to the bottom of the control stack -- before the level-1 LISTEN loop -- and then calls LISTEN. As always, LISTEN first ensures that MDL is receptive.

Examples:

```
<* 3 <+ a 1>>$

*ERROR*
ARG-WRONG-TYPE
+
LISTENING-AT-LEVEL 2 PROCESS 1
<ARGS <FRAME <FRAME>>>$
[a 1]
<ERRET 5>$        ;"This causes the + to return 5."
15                ;"finally returned by the *"
```

Note that when you are in an ERROR, the most recent set of bindings is still in effect. This means that you can examine values of dummy variables while still in the error state. For example,

```
<DEFINE F (A "AUX" (B "a string"))
        #DECL ((VALUE) LIST (A) STRUCTURED (B) STRING)
        (.B <REST .A 2>) ;"Return this LIST." >$
F
<F '(1)>$

*ERROR*
OUT-OF-BOUNDS
REST .
LISTENING-AT-LEVEL 2 PROCESS 1
.A$
(1)
.B$
"a string"
<ERRET '(5)>  ; "Make the REST return (5)."$
("a string" (5))
```

16.5. RETRY

<pre>   <RETRY <i>frame</i>></pre>

causes the control stack to be stripped down just beyond <i>frame</i>, and then causes the Subroutine call that generated <i>frame</i> to be done again. <i>frame</i> is optional, with default the FRAME of the last invocation of ERROR or LISTEN. RETRY differs from AGAIN in that (1) it is not intended to be used in programs; (2) it can retry any old <i>frame</i> (any Subroutine call), whereas AGAIN requires an ACTIVATION (PROG or REPEAT or "ACT"); and (3) if it retries the EVAL of a FORM that makes an ACTIVATION, it will cause rebinding in the argument LIST, thus duplicating side effects.

16.6. UNWIND

UNWIND is an FSUBR that takes two arguments, usually FORMs. It EVALs the first one, and, if the EVAL returns normally, the value of the EVAL call is the value of UNWIND. If, however, during the EVAL a non-local return attempts to return below the UNWIND FRAME in the control stack, the second argument is EVALed, its value is ignored, and the non-local return is completed. The second argument is evaluated in the environment that was present when the call to UNWIND was made. This facility is useful for cleaning up data bases that are in inconsistent states and for closing temporary CHANNELs that may be left around. FLOAD sets up an UNWIND to close its CHANNEL if the user attempts to ERRET without finishing the FLOAD. Example:

```
<DEFINE CLEAN ACT ("AUX" (C <OPEN "READ" "A FILE">))
        #DECL ((C) <OR CHANNEL FALSE> ...)
        <COND (.C
               <UNWIND <PROG () ... <CLOSE .C>>
                       <CLOSE .C>>)>>
```

16.7. Control-G (^G)

Typing control-G (^G, <ASCII 7>) at MDL causes it to act just as if an error had occurred in whatever was currently being done. You can then examine the values of variables as above, continue by applying ERRET to one argument (which is ignored), RETRY a FRAME lower on the control stack, or flush everything by applying ERRET to no arguments.

16.8. Control-S (^S)

Typing control-S (^S, <ASCII 19>) at MDL causes it to stop what is happening and return to the FRAME .LERR\ !-INTERRUPTS, returning the ATOM T.

16.9. OVERFLOW

<OVERFLOW *false-or-any*>

There is one error that can be disabled: numeric overflow and underflow caused by the arithmetic SUBRs (+, -, *, /). The SUBR OVERFLOW takes one argument: if it is of TYPE FALSE, under/overflow errors are disabled; otherwise they are enabled. The initial state is enabled. OVERFLOW returns T or #FALSE ( ), reflecting the previous state. Calling it with no argument returns the current state.

Chapter 17. Macro-operations

17.1. READ Macros

17.1.1. % and %%

The tokens % and %% are interpreted by READ in such a way as to give a "macro" capability to MDL similar to PL/I's.

Whenever READ encounters a single % -- anywhere, at any depth of recursion -- it immediately, without looking at the rest of the input, evaluates the object following the % The result of that evaluation is used by READ in place of the object following the % That is, % means "don't really READ this, use EVAL of it instead." % is often used in files in front of calls to ASCII, BITS (which see), etc., although when the FUNCTION is compiled the compiler will do the evaluation if the arguments are constant. Also seen is %.INCHAN, read as the CHANNEL in use during LOAD or FLOAD; for example, <PUT %.INCHAN 18 8> causes succeeding FIXes to be read as octal.

Whenever READ encounters %%, it likewise immediately evaluates the object following the %%. However, it completely ignores the result of that evaluation. Side effects of that evaluation remain, of course.

Example:

```
<DEFINE SETUP () <SET A 0>>$
SETUP
<DEFINE NXT () <SET A <+ .A 1>>>$
NXT
[%%<SETUP> %<NXT> %<NXT> (%%<SETUP>) %<NXT>]$
[1 2 () 1]
```

17.1.2. LINK

                        `<LINK exp:any string oblist>`

creates an object of TYPE LINK, PRIMTYPE ATOM. A LINK looks vaguely like an ATOM; it has a PNAME (the *string* argument), resides in an OBLIST (the *oblist* argument) and has a "value" (the *exp* argument). A LINK has the strange property that, whenever it is encountered by READ (that is, its PNAME is read, just like an ATOM, possibly with OBLIST trailers), READ substitutes the LINK's "value" for the LINK immediately. The effect of READing a LINK's PNAME is exactly the same as the effect of reading its "value".

The *oblist* argument is optional, with default `<1 .OBLIST>`. LINK returns its first argument. The LINK is created via INSERT, so an error results if there is already an ATOM or LINK in *oblist* with the same PNAME.

The primary use of LINKs is in interactive work with MDL; expressions which are commonly used, but annoyingly long to type, can be "linked" to PNAMEs which are shorter. The canonical example is the following:

                        `<LINK '<ERRET> "^E" <ROOT>>`

which links the ATOM of PNAME ^E in the ROOT OBLIST to the expression `<ERRET>`.

17.1.3. Program-defined Macro-characters

During READing from an input CHANNEL or PARSEing a STRING, any character can be made to have a special meaning. A character can cause an arbitrary routine to be invoked, which can then return any number of elements to be put into the object being built by READ, PARSE, or LPARSE. Translation of characters is also possible. This facility was designed for those persons who want to use the MDL reader to do large parts of their input but have to modify its actions for some areas; for example, one might want to treat left and right parentheses as tokens, rather than as delimiters indicating a LIST.

17.1.3.1. READ (finally)

Associated with READ is an ATOM, READ-TABLE!-, whose local value, if any, must be a VECTOR of elements, one for each character up to and including all characters to be treated specially. Each element indicates, if not 0, the action to be taken upon READ's encounter with that character. A similar VECTOR, the local value of PARSE-TABLE!-, if any, is used to find the action to take for characters encountered when PARSE or LPARSE is applied to a STRING.

These tables can have up to 256 elements, one for each ASCII character and one for each possible exclamation-point/ASCII-character pair. In MDL, the exclamation-point is used as a method of

expanding the ASCII character set, and an exclamation-point/character pair is treated as one logical character when not reading a STRING.

The element corresponding to a character is <NTH *table* <+ 1 <ASCII *char*>>>. The element corresponding to an exclamation-point/ASCII-character pair is <NTH *table* <+ 129 <ASCII *char*>>>. The table can be shorter than 256 elements, in which case it is treated as if it were 256 long with all the elements above its actual length 0.

An element of the tables must satisfy one of the following DECL Patterns:

'0 indicates that no special action is to be taken when this character is encountered.

CHARACTER indicates that the encountered character is to be translated into the given CHARACTER whenever it appears, except when as an object of TYPE CHARACTER, or in a STRING, or immediately following a \.

FIX indicates that the character is to be given the same treatment as the character with the ASCII value of the FIX. This allows you to cause other characters to be treated in the same way as A-Z for example. The same exceptions apply as described above.

<LIST FIX> indicates the same thing, except that the character does not by itself cause a break. Therefore, if it occurs when reading an ATOM or number, it will be treated as part of that ATOM or number.

APPLICABLE (to one argument) indicates that the character is to be a break character. Whenever it is encountered, the reading of the current object is finished, and the corresponding element of the table is APPLYed to the ASCII CHARACTER. (If READ is called during the application, the end-of-file slot of the CHANNEL temporarily contains a special kind of ACTIVATION (TYPE READA) so that end-of-file can be signalled properly to the original READ. Isn't that wonderful?) The value returned is taken to be what was read, unless an object of TYPE SPLICE is returned. If so, the elements of this object, which is of PRIMTYPE LIST, are spliced in at the point where MDL is reading. An empty SPLICE allows one to return nothing. If a structured object is not being built, and a SPLICE is returned, elements after the first will be ignored. A SPLICE during reading is similar to a SEGMENT during evaluating, except that, in some sense, a SPLICE says "expand me", whereas the structure containing a SEGMENT says "I will expand you".

<LIST APPLICABLE> indicates the same thing, except that the character does not by itself cause a break. Therefore, if it occurs when reading an ATOM or number, it will be treated as part of that ATOM or number.

READ takes an additional optional argument, which is what to use instead of the local value of the ATOM READ-TABLE as the VECTOR of read-macro characters. If this argument is supplied, READ-TABLE is rebound to it within the call to READ. READ takes from zero to four arguments. The fullest call to READ is thus:

> <READ *channel eof-routine lookup read-table:vector*>

The other arguments are explained in sections 11.1.1.1, 11.3, and 15.7.1.

ERROR and LISTEN rebind READ-TABLE to the GVAL of READ-TABLE, if any, else UNASSIGN it.

17.1.3.2. Examples

Examples of each of the different kinds of entries in macro tables:

```
<SET READ-TABLE <IVECTOR 256 0>>$
[...]

<PUT .READ-TABLE <+ 1 <ASCII !\a>> !\A>
        ;"CHARACTER: translate a to A."$
[...]
abc$
Abc


<PUT .READ-TABLE <+ 1 <ASCII !\%>> <ASCII !\A>>
        ;"FIX: make % just a normal ASCII character."$
[...]
A%BC$
A\%BC


<PUT .READ-TABLE <+ 1 <ASCII !\,>> (<ASCII !\,>)>
        ;"<LIST FIX>: make comma no longer a break
          character, but still special if at a break."$
[...]
A,B$
A\,B
;"That was an ATOM with PNAME A,B ."
',B$
,B
;"That was the FORM <GVAL B> ."


<PUT .READ-TABLE <+ 1 <ASCII !\:>>
    #FUNCTION ((X) <LIST COLON <READ>>)>
        ;"APPLICABLE: make a new thing like ( < and [ ."$
[...]
B:A$
B
(COLON A)
:::FOO$
(COLON (COLON (COLON FOO)))
```

```
<PUT .READ-TABLE <+ 1 <ASCII !\:>>
    '(#FUNCTION ((X) <LIST COLON <READ>>)))>
        ;"<LIST APPLICABLE>: like above, but not a break
        now."$
[...]
B:A$
B:A
;"That was an ATOM."
:::FOO$
(COLON (COLON (COLON FOO)))
```

17.1.3.3. PARSE and LPARSE (finally)

<PARSE *string radix lookup parse-table:vector look-ahead:character*>

is the fullest call to PARSE. PARSE can take from zero to five arguments. If PARSE is given no arguments, it returns the first object parsed from the local value of the STRING PARSE-STRING and additionally SETs PARSE-STRING to the STRING having those CHARACTERs which were parsed RESTed off. If PARSE is given a STRING to parse, the ATOM PARSE-STRING is rebound to the STRING within that call. If the *parse-table* argument is given to PARSE, PARSE-TABLE is rebound to it within that call to PARSE. Finally, PARSE can take a *look-ahead* CHARACTER, which is treated as if it were logically concatenated to the front of the *string* being parsed. Other arguments are described in sections 7.6.6.2 and 15.7.2.

LPARSE is exactly like PARSE, except that it tries to parse the whole STRING, returning a LIST of the objects created.

17.2. EVAL Macros

An EVAL macro provides the convenience of a FUNCTION without the overhead of calling, SPECIALs, etc. in the compiled version. A special-purpose function that is called often by FUNCTIONs that will be compiled is a good candidate for an EVAL macro.

17.2.1. DEFMAC and EXPAND

DEFMAC ("define macro") is syntactically exactly the same as DEFINE. However, instead of creating a FUNCTION, DEFMAC creates a MACRO. A MACRO is of PRIMTYPE LIST and in fact has a FUNCTION (or other APPLICABLE TYPE) as its single element.

A MACRO can itself be applied to arguments. A MACRO is applied in a funny way, however: it is

EVALed twice. The first EVAL causes the MACRO's element to be applied to the MACRO's arguments. Whatever that application returns (usually another FORM) is also EVALed. The result of the second EVALuation is the result of applying the MACRO. EXPAND is used to perform the first EVAL without the second.

To avoid complications, the first EVAL (by EXPAND, to create the object to be EVALed the second time around) is done at top level. The result of this policy is that two syntactically identical invocations of a MACRO always return the same expansion to be EVALed in the second step. The first EVAL generates two extra FRAMEs: one for a call to EXPAND, and one for a call to EVAL the MACRO application in a top-level environment.

Example:

```
<DEFMAC INC (ATM "OPTIONAL" (N 1))
        #DECL ((VALUE) FORM (ATM) ATOM (N) <OR FIX FLOAT>)
        <FORM SET .ATM <FORM + <FORM LVAL .ATM> .N>>>$
INC
,INC$
#MACRO (#FUNCTION ((ATM "OPTIONAL" (N 1)) ...))
<SET X 1>$
1
<INC X>$
2
.X$
2
<EXPAND '<INC X>>$
<SET X <+ .X 1>>
```

Perhaps the intention is clearer if PARSE and % are used:

```
<DEFMAC INC (ATM "OPTIONAL" (N 1))
        #DECL (...)
        <PARSE "<SET %.ATM <+ %.ATM %.N>>">>
```

MACROs really exhibit their advantages when they are compiled. The compiler will simply cause the first EVALuation to occur (via EXPAND) and compile the result. The single element of a compiled MACRO is an RSUBR or RSUBR-ENTRY.

17.2.2. Example

Suppose you want to change the following simple FUNCTION to a MACRO:

```
<DEFINE DOUBLE (X) #DECL ((X) FIX) <+ .X .X>>
```

You may be tempted to write:

```
<DEFMAC DOUBLE (X) #DECL ((X) FIX) <FORM + .X .X>>
```

This MACRO works, but only when the argument does not use temporary bindings. Consider

```
<DEFINE TRIPLE (Y) <+ .Y <DOUBLE .Y>>>
```

If this FUNCTION is applied, the top-level binding of Y is used, not the binding just created by the application. Compilation of this FUNCTION would probably fail, because the compiler probably would have no top-level binding for Y. Well, how about

```
<DEFMAC DOUBLE ('X) <FORM + .X .X>>   ;"The DECL has to go."
```

Now this is more like the original FUNCTION, because no longer is the argument evaluated and the result evaluated again. And TRIPLE works. But now consider

```
<DEFINE INC-AND-DOUBLE (Y) <DOUBLE <SET Y <+ 1 .Y>>>>
```

You might hope that

```
<INC-AND-DOUBLE 1> -> <DOUBLE <SET Y <+ 1 1>>>
                   -> <DOUBLE 2>
                   -> <+ 2 2>
                   -> 4
```

But, when DOUBLE is applied to that FORM, the argument is QUOTEd, so:

```
<INC-AND-DOUBLE 1> -> <DOUBLE <SET Y <+ 1 .Y>>>
                   -> <FORM + <SET Y <+ 1 .Y>> <SET Y <1 .Y>>>
                   -> <+ 2 3>
                   -> 5
```

So, since the evaluation of DOUBLE's argument has a side effect, you should ensure that the evaluation is done exactly once, say by FORM:

```
<DEFMAC DOUBLE ('ANY)
        <FORM PROG ((X .ANY)) #DECL ((X) FIX) '<+ .X .X>>>
```

As a bonus, the DECL can once more be used.

This example is intended to show that writing good MACROs is a little trickier than writing good FUNCTIONs. But the effort may be worthwhile if the compiled program must be speedy.

Chapter 18. Machine Words and Bits

The MDL facility for dealing with uninterpreted machine words and bits involves two data TYPEs: WORD and BITS. A WORD is simply an uninterpreted machine word, while a BITS is a "pointer" to a set of bits within a WORD. Operating on WORDs is usually done only when compiled programs are used (chapter 19).

18.1. WORDs

A WORD in MDL is a PDP-10 machine word of 36 bits. A WORD always PRINTs in "# format", and its contents are always printed in octal (hence preceded and followed by *). Examples:

```
#WORD 0                   ;"all 0s"$
#WORD *000000000000*

#WORD *2000*              ;"one bit 1"$
#WORD *000000002000*

#WORD *525252525252*      ;"every other bit 1"$
#WORD *525252525252*
```

WORD is its own PRIMTYPE; it is also the PRIMTYPE of FIX, FLOAT, CHARACTER, and any other TYPE which can fit its data into one machine word.

A WORD cannot be an argument to +, -, or indeed any SUBRs except for CHTYPE, GETBITS, PUTBITS and several bitwise logical functions, all to be described below. Thus any arithmetic bit manipulation must be done by CHTYPEing a WORD to FIX, doing the arithmetic, and then CHTYPEing back to WORD. However, bit manipulation can be done without CHTYPEing the thing to be played with to a WORD, so long as it is of PRIMTYPE WORD; the result of the manipulation will be of the same TYPE as the original object or can be CHTYPEd to it.

18.2. BITS

An object of TYPE BITS is of PRIMTYPE WORD, and PRINTs just like a WORD. The internal form of a BITS is precisely that of a PDP-10 "byte pointer", which is, in fact, just what a BITS is.

For purposes of explaining what a BITS is, assume that the bits in a WORD are numbered from <u>right</u> to <u>left</u>, with the rightmost bit numbered 0 and the leftmost numbered 35, as in

       35 34 33 ... 2 1 0

(This is not the "standard" ordering; the "standard" one goes from left to right.)

A BITS is most conveniently created via the SUBR BITS:

       ‹BITS *width:fix right-edge:fix*›

returns a BITS which "points to" a set of bits *width* wide, with rightmost bit *right-edge*. Both arguments must be of TYPE FIX, and the second is optional with default 0.

Examples: the indicated application of BITS returns an object of TYPE BITS which points to the indicated set of bits in a WORD:

       ‹BITS 7›   35 ... 7 <u>6 ... 0</u>

       ‹BITS 4 18›   35 ... 22 <u>21 20 19 18</u> 17 ... 0

       ‹BITS 36›   <u>35 ... 0</u>

18.3. GETBITS

       ‹GETBITS *from:primtype-word bits*›

where *from* is an object of PRIMTYPE WORD, returns a new object whose TYPE is WORD. This object is constructed in the following way: the set of bits in *from* pointed to by *bits* is copied into the new object, right-adjusted, that is, lined up against the right edge (bit number 0) of the new object. All those bits of the new object which are not copied are set to zero. In other words, GETBITS takes bits from an arbitrary place in *from* and puts them at the right of a new object. The *from* argument to GETBITS is not affected.

Examples:

```
<GETBITS #WORD *777777777777* <BITS 3>>$
#WORD *000000000007*
<GETBITS *012345670123* <BITS 6 18>>$
#WORD *000000000045*
```

18.4. PUTBITS

<PUTBITS *to:primtype-word bits from:primtype-word*>

where *to* and *from* are of PRIMTYPE WORD, returns a copy of *to*, modified as follows: the set of bits in *to* which are pointed to by *bits* are replaced by the appropriate number of rightmost bits copied from *from* (optional, default 0). In other words: PUTBITS takes bits from the right of *from* and stuffs them into an arbitrary position in *to*. None of the arguments to PUTBITS is affected.

Examples:

```
<PUTBITS #WORD *777777777777* <BITS 6 3>>$
#WORD *777777777007*
<PUTBITS #WORD *666777000111* <BITS 5 15> #WORD *123*>$
#WORD *666776300111*
<PUTBITS #WORD *765432107654* <BITS 18>>$
#WORD *765432000000*
```

18.5. Bitwise Boolean Operations

Each of the SUBRs ANDB, ORB, XORB, and EQVB takes arguments of PRIMTYPE WORD and returns a WORD which is the bitwise Boolean "and", inclusive "or", exclusive "or", or "equivalence" (inverse of exclusive "or"), respectively, of its arguments. Each takes any number of arguments. If no argument is given, a WORD with all bits off (ORB and XORB) or on (ANDB and EQVB) is returned. If only one argument is given, it is returned unchanged but CHTYPEd to a WORD. If more than two arguments are given, the operator is applied to the first two, then applied to that result and the third, etc. Be careful not to confuse AND and OR with ANDB and ORB.

Chapter 19. Compiled Programs

19.1. RSUBR (the TYPE)

RSUBRs ("relocatable subroutines") are machine-language programs written to run in the MDL environment. They are usually produced by the MDL assembler (often from output produced by the compiler) although this is not necessary. All RSUBRs have two components: the "reference vector" and the "code vector". In some cases the code vector is in pure storage. There is also a set of "fixups" associated with every RSUBR, although it may not be available in the running MDL.

19.2. The Reference Vector

An RSUBR is basically a VECTOR that has been CHTYPEd to TYPE RSUBR via the SUBR RSUBR (see below). This ex-VECTOR is the reference vector. The first three elements of the reference vector have predefined meanings:

> The first element is of TYPE CODE or PCODE and is the impure or pure code vector respectively.
> The second element is an ATOM and specifies the name of the RSUBR.
> The third element is of TYPE DECL and declares the type/structure of the RSUBR's arguments and result.

The rest of the elements of the reference vector are objects in garbage-collected storage that the RSUBR needs to reference and any impure slots that the RSUBR needs to use.

When the RSUBR is running, one of the PDP-10 accumulators (with symbolic name R) is always pointing to the reference vector, to permit rapid access to the various elements.

19.3. RSUBR Linking

RSUBRs can call any APPLICABLE object, all in a uniform manner. In general, a call to an F/SUBR is linked up at assembly/compile time so that the calling instruction (UUO) points directly at the code in the interpreter for the F/SUBR. However, the locations of most other APPLICABLEs are not known at assembly/compile time. Therefore, the calling UUO is set up to point at a slot in the reference vector (by indexing off accumulator R). This slot initially contains the ATOM whose G/LVAL is the called object. The calling mechanism (UUO handler) causes control to be transferred to the called object and, depending on the state of the RSUBR-link flag, the ATOM will be replaced by its G/LVAL. (If the call is of the "quick" variety, the called RSUBR or RSUBR-ENTRY will be CHTYPEd to a QUICK-RSUBR or QUICK-ENTRY, respectively, before replacement.) Regardless of the RSUBR-link flag's state, calls to FUNCTIONs are never permanently linked. A call to a non-Subroutine generates an extra FRAME, whose FUNCT is the dummy ATOM CALLER.

RSUBRs are linked together for faster execution, but linking may not be desirable if the RSUBRs are being debugged, and various versions are being reloaded. A linked call will forever after go to the same code, regardless of the current G/LVAL of the called ATOM. Thus, while testing RSUBRs, you may want to disable linking, by calling the RSUBR-LINK SUBR with a FALSE argument. Calling it with a non-FALSE argument enables linking thereafter. It returns the previous state of the link flag, either T or #FALSE ( ). Calling it with no argument returns the current state.

19.4. Pure and Impure Code

The first element of an RSUBR is the code vector, of TYPE CODE or PCODE. TYPE CODE is of PRIMTYPE UVECTOR, and the UTYPE should be of PRIMTYPE WORD. The code vector is simply a block of words that are the instructions which comprise the RSUBR. Since the code vector is stored just like a standard UVECTOR, it will be moved around by the garbage collector. Therefore, all RSUBR code is required to be location-insensitive. The compiler guarantees the location-insensitivity of its output. The assembler helps to make the code location-insensitive by defining all labels as offsets relative to the beginning of the code vector and causing instructions that refer to labels to index automatically off the PDP-10 accumulator symbolically named M. M, like R, is set up by the UUO handler, but it points to the code vector instead of the reference vector. The code vector of an RSUBR can be frozen (using the FREEZE SUBR) to prevent it from moving during debugging by DDT in the superior job or fork.

If the first element of an RSUBR is of TYPE PCODE ("pure code"), the code vector of the RSUBR is pure and sharable. TYPE PCODE is of PRIMTYPE WORD. The left half of the word specifies an offset into an internal table of pure RSUBRs, and the right half specifies an offset into the block of code where this RSUBR starts. The PCODE prints out as:

    %<PCODE *name:string offset:fix*>

where *name* names the entry in the user's pure-RSUBR table, and *offset* is the offset. (Obviously, PCODE is also the name of a SUBR, which generates a pure code vector.) Pure RSUBRs may also move around, but only by being included in MDL's page map at different places. Once again M can be used exactly as before to do location-independent address referencing. Individual pure code vectors can be "unmapped" (marked as being not in primary storage but in their original pure-code disk files) if the space in storage allocated for pure code is exhausted. An unmapped RSUBR is mapped in again whenever needed. All pure RSUBRs are unmapped before a SAVE file is written, so that the code is not duplicated on disk. A purified RSUBR must use RGLOC ("relative GLOC") instead of GLOC. RGLOC produces objects of TYPE LOCR instead of LOCD.

19.5. TYPE-C and TYPE-W

In order to handle user NEWTYPEs reasonably, the internal TYPE codes for them have to be able to be different from one MDL run to another. Therefore, references to the TYPE codes must be in the reference vector rather than the code vector. To help handle this problem, two TYPEs exist, TYPE-C ("type code") and TYPE-W ("type word"), both of PRIMTYPE WORD. They print as:

        %<TYPE-C *type primtype:atom*>
        X<TYPE-W *type primtype:atom*>

The SUBR TYPE-C produces an internal TYPE code for the *type*, and TYPE-W produces a prototype "TYPE word" (appendix 1) for an object of that TYPE. The *primtype* argument is optional, included only as a check against the call to NEWTYPE. TYPE-W can also take a third argument, of PRIMTYPE WORD, whose right half is included in the generated "TYPE word". If *type* is not a valid TYPE, a NEWTYPE is automatically done.

To be complete, a similar SUBR and TYPE should be mentioned here.

        <PRIMTYPE-C *type*>

produces an internal "storage allocation code" (appendix 1) for the *type*. The value is of TYPE PRIMTYPE-C, PRIMTYPE WORD. In almost all cases the SUBR TYPEPRIM gives just as much information, except in the case of TEMPLATEs: all TYPEs of TEMPLATEs have the same TYPEPRIM, but they all have different PRIMTYPE-Cs.

19.6. RSUBR (the SUBR)

        <RSUBR [*code name decl ref ref* ...]>

CHTYPEs its argument to an RSUBR, after checking it for legality. RSUBR is rarely called other than

in the MDL Assembler (ref 2). It can be used if changes must be made to an RSUBR that are prohibited by MDL's built-in safety mechanisms. For example, if the GVAL of *name* is an RSUBR:

```
<SET FIXIT <CHTYPE ,name VECTOR>>$
[...]

...(changes to .FIXIT)...

<SETG name <RSUBR .FIXIT>>$
#RSUBR [...]
```


19.7. RSUBR-ENTRY

RSUBRs can have multiple entry points. An RSUBR-ENTRY can be applied to arguments exactly like an RSUBR.

<RSUBR-ENTRY [*rsubr-or-atom name:atom decl*] *offset:fix*>

returns the VECTOR argument CHTYPEd to an RSUBR-ENTRY into the *rsubr* at the specified *offset*. If the RSUBR-ENTRY is to have a DECL (RSUBR style), it should come as shown.

<ENTRY-LOC *rsubr-entry*>

("entry location") returns the *offset* into the RSUBR of this entry.

19.8. RSUBRs in Files

There are three kinds of files that can contain RSUBRs, identified by second names BINARY, NBIN and FBIN. There is nothing magic about these names, but they are used by convention.

A BINARY file is a completely ASCII file containing complete impure RSUBRs in character representation. Even a code vector appears as #CODE followed by a UVECTOR of FIXes, LOSEs or WORDs. BINARY files are generally slow to load, because of all the parsing that must be done.

An NBIN file contains a mixture of ASCII characters and binary code. The start of a binary portion is signalled to READ by the character control-C, so naive readers of an NBIN file under ITS may incorrectly assume that it ends before any binary code appears. An NBIN file cannot be edited with a text editor. An RSUBR is written in NBIN format by being PRINTed on a "PRINTB" CHANNEL. The RSUBRs in NBIN files are not purified either.

An FBIN file is actually part of a triad of files. The FBIN file(s) itself is the impure part of a collection of purified RSUBRs. It is simply ASCII and can be edited at will. (Exception: under ITS, the first object in the file should not be removed or changed in any way, lest a "grim reaper" program for FBIN files think that the other files in the triad are obsolete and delete them.) The pure code itself resides (under ITS) in a special large file that contains all currently-used pure code, or (under TENEX) in a file in a special disk directory with first name the same as the *name* argument to PCODE for the RSUBR. The pure-code file is page-mapped directly into MDL storage in read-only mode. It can be unmapped when the pure storage must be reclaimed, and it can be mapped at a different storage address when pure storage must be compacted. There is also a fixup file (see below) or portion of a file associated with the FBIN to round out the triad.

An initial MDL can have pure RSUBRs in it that were "loaded" during the initialization process. The files are not page-mapped in until they are actually needed. The "loading" has other side effects, such as the creation of OBLISTs (chapter 15). Exactly what is pre-loaded is outside the scope of this document.

19.9. Fixups

The purpose of fixups is to fix up references in the RSUBR to parts of the interpreter that change from one version of MDL to the next. The reason the fixups contain a version number is so that they can be completely ignored when an RSUBR is loaded into the same version of MDL from which it was last written out.

There are three forms of fixups, corresponding to the three kinds of RSUBR files. ASCII RSUBRs, found in BINARY files, have ASCII fixups. The fixups are contained in a LIST that has the following format:

```
(MDL-version:fix
        name:atom value:fix (use:fix use:fix ...)
        name:atom value:fix (use:fix use:fix ...)
        ...)
```

The fixups in NBIN files and the fixup files associated with FBIN files are in a fast internal format that looks like a UVECTOR of PRIMTYPE WORDs.

Fixups are usually discarded after they are used during the loading process. However, if, while reading a BINARY or NBIN file the ATOM KEEP-FIXUPS!- has a non-FALSE LVAL, the fixups will be kept, via an association between the RSUBR and the ATOM RSUBR. It should be noted that, besides fixing up the code, the fixups themselves are fixed up when KEEP-FIXUPS is bound and true. Also, the assembler and compiler make the same association when they first create an RSUBR, so that it can be written out with its fixups.

In the case of pure RSUBRs (FBIN files), things are a little different. If a pure-code file exists for this version of MDL, it is used immediately, and the fixups are completely ignored. If a pure-code file for this version doesn't exist, the fixup file is used to create a new copy of the file from an old one, and also a new version of the fixup file is created to go with the new pure-code file. This all goes on automatically behind the user's back.

Chapter 20. Multiprocessing

This chapter purports to explain the multiprocessing primitives of MDL. It does make some attempt to explain multiprocessing as such, but only as required to specify the primitives. If you are unfamiliar with the basic concepts, confusion shall probably reign.

MDL's built-in multiprocessing primitives do not include a "time-sharing system". Only one "process" is ever running at a time, and control is passed back and forth between "processes" on a coroutine-like basis. The primitives are sufficient, however, to allow the writing of a "time-sharing system" in MDL, with the additional use of the MDL interrupt primitives. This has, in fact, been done.

20.1. PROCESS (the TYPE)

A PROCESS is an object which contains the "current state" of a computation. This includes the LVALs of ATOMs ("bindings"), "depth" of functional application, and "position" within the application of each applied function. Some of the things which are not part of any specific PROCESS are the GVALs of ATOMs, associations (ASOCs), and the contents of OBLISTs. GVALs (with OBLISTs) are a chief means of communication and sharing between PROCESSes (all PROCESSes can access the SUBR which is the GVAL of +, for instance). Note that an LVAL in one PROCESS cannot easily be directly referenced from another PROCESS.

A PROCESS PRINTs as #PROCESS p, where p is a FIX which uniquely identifies the PROCESS; p is the "process number" typed out by LISTEN. A PROCESS cannot be read in by READ.

The term "run a PROCESS" will be used below to mean "perform some computation, using the PROCESS to record the intermediate states of that computation".

N.B.: A PROCESS is a rather large object; creating one will often cause a garbage collection.

20.2. STATE of a PROCESS

<STATE *process*>

returns an ATOM (in the ROOT OBLIST) which indicates the "state" of the PROCESS *process*. The ATOMs which STATE can return, and their meanings, are as follows:

RUNABLE (sic) — *process* has never ever been run.

RUNNING — *process* is currently running, that is, it did the application of STATE.

RESUMABLE — *process* has been run, is not currently running, and can run again.

DEAD — *process* has been run, but it can not run again; it has "terminated".

In addition, an interrupt (chapter 21) can be enabled to detect the time at which a PROCESS becomes "blocked" (waiting for console input) or "unblocked" (console input arrived).

20.3. PROCESS (the SUBR)

<PROCESS *starter:applicable*>

creates and returns a new PROCESS but does not run it; the STATE of the returned PROCESS is RUNABLE (sic).

starter is something applicable to one argument, which must be evaluated. *starter* is used both in starting and "terminating" a PROCESS. In particular, if the *starter* of a PROCESS ever returns a value, that PROCESS becomes DEAD.

20.4. RESUME

The SUBR RESUME is used to cause a computation to start or to continue running in another PROCESS. An application of RESUME looks like this:

<RESUME *retval:any process*>

where *retval* is the "returned value" (see below) of the PROCESS that does the RESUME, and *process* is the PROCESS to be started or continued.

The *process* argument to RESUME is optional, and the default is the last PROCESS, if any, to RESUME

the PROCESS in which this RESUME is applied. If and when the current PROCESS is later RESUMEd by another PROCESS, that RESUME's *retval* is returned as the value of this RESUME.

20.5. Switching PROCESSes

20.5.1. Starting Up a New PROCESS

Let us say that we are running in some PROCESS, and that this original PROCESS is the GVAL of P0. Somewhere, we have evaluated

        <SETG P1 <PROCESS ,STARTER>>

where ,STARTER is some appropriate function. Now, in ,P0, we evaluate

        <RESUME .A ,P1>

and the following happens:

>   (1) In ,P0 the arguments of the RESUME are evaluated: that is, we get that LVAL of A which is current in ,P0 and the GVAL of P1.

>   (2) The STATE of ,P0 is changed to RESUMABLE and ,P0 is "frozen" right where it is, in the middle of the RESUME.

>   (3) The STATE of ,P1 is changed to RUNNING, and ,STARTER is applied to ,P0's LVAL of A in ,P1. ,P1 now continues on its way, evaluating the body of ,STARTER.

The .A in the RESUME could have been anything, of course. The important point is that, whatever it is, it is evaluated in ,P0.

What happens next depends, of course, on what ,STARTER does.

20.5.2. Top-level Return

Let us initially assume that ,STARTER does nothing relating to PROCESSes, but instead simply returns a value -- say *starval*. What happens when ,STARTER returns is this:

>   (1) The STATE of ,P1 is changed to DEAD. ,P1 can never again be RESUMEd.

>   (2) The last PROCESS to RESUME ,P1 is found, namely ,P0, and its STATE is changed to RUNNING.

(3) *starval* is returned in ,PO as the value of the original RESUME, and ,PO continues where it left off.

All in all, this simple case looks just like an elaborate version of applying ,STARTER to .A in ,PO.

20.5.3. Symmetric RESUMEing

Now suppose that while still in ,P1 the following is evaluated, either in ,STARTER or in something called by ,STARTER:

        <RESUME .BAR ,PO>

This is what happens:

(1) The arguments of the RESUME are evaluated <u>in ,P1</u>.

(2) The STATE of ,P1 is changed to RESUMABLE, and ,P1 is "frozen" right in the middle of the RESUME.

(3) The STATE of ,PO is changed to RUNNING, and ,P1's LVAL of BAR is returned as the value of <u>,PO's</u> original RESUME. ,PO then continues right where it left off.

This is <u>the</u> interesting case, because ,PO can now do <u>another</u> RESUME of ,P1; this will "turn off" ,PO, pass a value to ,P1 and "turn on" ,P1. ,P1 can now again RESUME ,PO, which can RESUME ,P1 back again, etc. <u>ad nauseam</u>, with everything done in a perfectly symmetric manner. This can obviously also be done with three or more PROCESSes in the same manner.

Note how this differs from normal functional application: you cannot "return" from a function without destroying the state that function is in. The whole point of PROCESSes is that you can "return" (RESUME), remembering your state, and later continue where you left off.

20.6. Example

```
;"Initially, we are in LISTEN in some PROCESS."
<DEFINE SUM3 (A)
        #DECL ((A) <OR FIX FLOAT>)
        <REPEAT ((S .A))
                #DECL ((S) <OR FIX FLOAT>)
                <SET S <+ .S <RESUME "GOT 1">>>
                <SET S <+ .S <RESUME "GOT 2">>>
                <SET S <RESUME .S>>>>$
SUM3
```

```
;"SUM3, used as the startup function of another PROCESS,
gets RESUMEd with numbers.  It returns the sum of the last
three numbers it was given every third RESUME."
<SETG SUMUP <PROCESS ,SUM3>>$
#PROCESS 2
;"Now we start SUMUP and give SUM3 its three numbers."
<RESUME 5 ,SUMUP>$
"GOT 1"
<RESUME 1 ,SUMUP>$
"GOT 2"
<RESUME 2 ,SUMUP>$
8
```

Just as a note, by taking advantage of MDL's order of evaluation, SUM3 could have been written as:

```
<DEFINE SUM3 (A)
        <REPEAT ((S .A))
            #DECL ((A S) <OR FIX FLOAT>)
            <SET S <RESUME <+ .S <RESUME "GOT 1"> <RESUME "GOT 2">>>>>>
```

20.7. Other Multiprocessing Features

20.7.1. BREAK-SEQ

<BREAK-SEQ *any process*>

("break evaluation sequence") returns *process*, which must be RESUMABLE, after having modified it so that when it is next RESUMEd, it will first evaluate *any* and then do an absolutely normal RESUME; the value returned by *any* is thrown away, and the value given by the RESUME is used normally.

If a PROCESS is BREAK-SEQed more than once between RESUMEs, all of the *anys* BREAK-SEQed onto it will be remembered and evaluated when the RESUME is finally done. The *anys* will be evaluated in "last-in first-out" order. The FRAME generated by EVALing more than one *any* will have as its FUNCT the dummy ATOM BREAKER.

20.7.2. MAIN

When you initially start up MDL, the PROCESS in which you are running is slightly "special" in these two ways:

(1) Any attempt to cause it to become DEAD will be met with an ERROR.

(2) <MAIN> always returns that PROCESS.

The PROCESS number of <MAIN> is always 1. The initial GVAL of THIS-PROCESS is what MAIN always returns, #PROCESS 1.

20.7.3. ME

        <ME>

returns the PROCESS in which it is evaluated. The LVAL of THIS-PROCESS in a RUNABLE (new) PROCESS is what ME always returns.

20.7.4. RESUMER

        <RESUMER *process*>

returns the PROCESS which last RESUMEd *process*. If no PROCESS has ever RESUMEd *process*, it returns #FALSE ( ). *process* is optional, with default <ME>. Note that <MAIN> does not ever have any resumer. Example:

```
<PROG ((R <RESUMER>))            ;"not effective in <MAIN>"
    #DECL ((R) <OR PROCESS FALSE>)
    <AND .R
        <==? <STATE .R> RESUMABLE>
        <RESUME T .R>>>
```

20.7.5. SUICIDE

        <SUICIDE *retval process*>

acts just like RESUME, but clobbers the PROCESS (which cannot be <MAIN>) in which it is evaluated to the STATE DEAD.

20.7.6. 1STEP

        <1STEP *process*>

returns *process*, after putting it into "single-step mode".

A PROCESS in single-step mode, whenever RESUMEd, runs only until an application of EVAL in it

begins or finishes. At that point in time, the PROCESS that did the 1STEP is RESUMEd, with a *retval* which is a TUPLE. If an application of EVAL just began, the TUPLE contains the ATOM EVLIN and the arguments to EVAL. If an application of EVAL just finished, the TUPLE contains the ATOM EVLOUT and the result of the evaluation.

process will remain in single-step mode until FREE-RUN (below) is applied to it. Until then, it will stop before and after each EVAL in it. Exception: if it is RESUMEd from an EVLIN break with a *retval* of TYPE DISMISS (PRIMTYPE ATOM), it will leave single-step mode only until the current call to EVAL is about to return. Thus lower-level EVALs are skipped over without leaving the mode. The usefulness of this mode in debugging is obvious.

20.7.7. FREE-RUN

<FREE-RUN *process*>

takes its argument out of single-step mode. Only the PROCESS that put *process* into single-step mode can take it out of the mode; if another PROCESS tries, FREE-RUN returns a FALSE.

20.8. Sneakiness with PROCESSes

FRAMEs, ENVIRONMENTs, TAGs, and ACTIVATIONs are specific to the PROCESS which created them, and each "knows its own father". Any SUBR which takes these objects as arguments can take one which was generated by any PROCESS, no matter where the SUBR is really applied. This provides a rather sneaky means of crossing between PROCESSes. The various cases are as follows:

GO, RETURN, AGAIN, and ERRET, given arguments which lie in another PROCESS, each effectively "restarts" the PROCESS of its argument and acts as if it were evaluated over there. If the PROCESS in which it was executed is later RESUMEd, it returns a value just like RESUME!

SET, UNASSIGN, BOUND?, ASSIGNED?, LVAL, VALUE, and LLOC, given optional ENVIRONMENT arguments which lie in another PROCESS, will gleefully change, or return, the local values of ATOMs in the other PROCESS. The optional argument can equally well be a PROCESS, FRAME, or ACTIVATION in another PROCESS; in those cases, each uses the ENVIRONMENT which is current in the place specified.

FRAME, ARGS, and FUNCT will be glad to return the FRAMEs, argument TUPLEs, and applied Subroutine names of another PROCESS. If one is given a PROCESS (including <ME>) as an argument instead of a FRAME, it returns all or the appropriate part of the topmost FRAME on that PROCESS's control stack.

If EVAL is applied in PROCESS P1 with an ENVIRONMENT argument from a PROCESS P2, it will do the evaluation in P1 but with P2's ENVIRONMENT (!). That is, the other PROCESS's LVALs, etc. will be used,

but (1) any <u>new</u> FRAMEs needed in the course of the evaluation will be created in P1; and (2) <u>P1</u> will be RUNNING -- not P2. Note the following: if the EVAL in P1 eventually causes a RESUME of P2, P2 could functionally return to below the point where the ENVIRONMENT used in P1 is defined; a RESUME of P1 at this point would cause an ERROR due to an invalid ENVIRONMENT. (Once again, LEGAL? can be used to forestall this.)

20.9. Final Notes

(1) A RESUMABLE PROCESS can be used in place of an ENVIRONMENT in any application. The "current" ENVIRONMENT of the PROCESS is effectively used.

(2) FRAMEs and ENVIRONMENTs can be CHTYPEd arbitrarily to one another, or an ACTIVATION can be CHTYPEd to either of them, and the result "works".

(3) Bugs in multi-process programs usually exhibit a degree of subtlety and nastiness otherwise unknown to the human mind. If when attempting to work with multiple processes you begin to feel that you are rapidly going insane, you are in good company.

Chapter 21. Interrupts

The MDL interrupt-handling facilities essentially provide the ability to say the following: whenever "this event" occurs, stop whatever is being done at the time and perform "this action"; when "this action" is finished, continue with whatever was originally being done. "This action" is the application of a specified APPLICABLE object to specified arguments. "This event" can be things like the typing of a character at a console, a time interval ending, a PROCESS becoming blocked, or a user-defined and -generated "event". The sets of events and actions can be changed in extremely flexible ways, which accounts for both the variety of SUBRs and arguments, and the rich interweaving of the topics in this chapter. If you think you detect a similarity between interrupt handling and multiprocessing, you're right.

21.1. Definitions of Terms

An interrupt is not an object in MDL, but rather a class of events, for example, "ticks" of a clock, garbage collections, the typing of a character at a console, etc.

An interrupt is said to have occurred when one of the events in its class takes place.

An interrupt is said to have been enabled when MDL has been told to react to its occurrence in some way -- that is, perform one or more actions when it occurs. A disabled interrupt causes no actions to be performed.

An interrupt is said to have been offed when even the interpreter does not react to its occurrence. An interrupt that is on (or "has been oned") is noticed by the interpreter, which then goes to see whether the interrupt is enabled.

An action consists of applying a handler to arguments supplied by the interrupt system. The number and meaning of the arguments depend on the specific interrupt.

Each interrupt has a name which is either a STRING (for example, "GC", "CHAR", "WRITE") or an ATOM with that PNAME in a special OBLIST, named INTERRUPTS!- . (This OBLIST is returned by <INTERRUPTS>.) Certain names must be further qualified by a CHANNEL or a LOCATIVE to tell which interrupt by that name is meant.

Each interrupt that is on has a priority, a FIX greater than 0 which specifies its "importance". The processing of a higher-priority (larger-numbered) interrupt will supercede the processing of a lower-priority interrupt.

21.2. IHEADER and HANDLER (the TYPEs)

One IHEADER ("interrupt header") and a chain of HANDLERs are associated with each interrupt that is on. Both TYPEs are of PRIMTYPE VECTOR, but they do not print as such, since they are circular. Instead they PRINT as

#type most-interesting-component

The contents of IHEADERs and HANDLERs can be changed by PUT, and the new values will then determine the behavior of MDL.

21.2.1. IHEADERs

There is one IHEADER associated with each interrupt that is on. EVENT and ON (see below) will automatically create one if none exists, and OFF (see below) will destroy one when it has no reason to continue existing. The elements of an IHEADER are as follows:

(1) name of interrupt (STRING, or CHANNEL if the name is "CHAR", or LOCATIVE if the name is "READ" or "WRITE")
(2) nonzero if and only if disabled
(3) first HANDLER, if any, else a zero-length HANDLER
(4) priority

Obtaining an IHEADER is a little complicated.

For "READ" interrupts, <GET *locative* READ!-INTERRUPTS!-> returns the IHEADER.

For "WRITE" interrupts, <GET *locative* WRITE!-INTERRUPTS!-> returns the IHEADER.

For "CHAR" interrupts, <GET *channel* INTERRUPT!-> returns the IHEADER.

Otherwise, the IHEADER is PUT on the name ATOM with the indicator INTERRUPT!- . Thus, for example, <GET CLOCK!-INTERRUPTS INTERRUPT!-> returns the IHEADER for the clock interrupt (if it exists).

21.2.2. HANDLERs

A new HANDLER is generated each time HANDLER or ON (see below) is applied. A HANDLER specifies a particular action for a particular interrupt. The elements of a HANDLER are as follows:

(1) next HANDLER if any, else a zero-length HANDLER

(2) previous HANDLER or the IHEADER (Thus the HANDLERs of a given interrupt form a "doubly-linked list" chaining between each other and back to the IHEADER.)

(3) handler to be applied (anything APPLICABLE that evaluates its arguments -- the application is done not by APPLY but by RUNINT, which can take a PROCESS argument: see next line)

(4) PROCESS in which the handler will be applied, or #PROCESS 0, meaning whatever PROCESS was running when the interrupt occurred. In the former case, RUNINT is applied to the handler and its arguments in the currently running PROCESS, which causes an APPLY in the interrupt PROCESS, which must be RESUMABLE. The running PROCESS becomes RESUMABLE, and the interrupt PROCESS becomes RUNNING, but no other PROCESS variables (for example RESUMER) are changed.

21.3. EVENT

<EVENT *name priority which*>

This SUBR sets up an enabled IHEADER without adding any HANDLERs. It returns the IHEADER. The *name* may be an ATOM in the INTERRUPTS OBLIST or a STRING. (If it is a STRING, EVENT does a LOOKUP or INSERT in <INTERRUPTS>.)

which must be given only for certain *names*. It must be a CHANNEL if and only if *name* is "CHAR" or CHAR!-INTERRUPTS!-. In this case it is the input CHANNEL from the (pseudo-)console or Network socket whose received characters will cause the interrupt to occur, or the output CHANNEL to the pseudo-console or Network socket whose desired characters will cause the interrupt to occur. (See section 21.10.1 below. Pseudo-consoles are not available under TENEX.) The argument must be a LOCATIVE if and only if *name* is "READ" (or the ATOM) or "WRITE" (or the ATOM). In this case it specifies an object to be "monitored" (section 21.10.7).

EVENT can also be called with one argument, an IHEADER. If this IHEADER was previously disabled or OFFed, EVENT will reinstate and enable it.

21.4. HANDLER (the SUBR)

<HANDLER *iheader applicable process*>

This SUBR makes a HANDLER, attaches it to the front of *iheader*'s HANDLER chain (first to be performed), and returns it as a value. *applicable* may be either an APPLICABLE handler or a previously OFFed HANDLER. (In the latter case the HANDLER is put into the chain, and a new HANDLER is not created.) *process* is optional, with default #PROCESS 0.

21.5. ON

<ON *name applicable priority:fix process which*>

ON is a combination of EVENT and HANDLER: it creates or finds the IHEADER, reinstates and enables it, and adds a HANDLER to the front of the chain (first to be performed). It returns the HANDLER.

The *process* argument is optional. It can be supplied as, and defaults to, 0, meaning #PROCESS 0. *which* is only sometimes required, as for EVENT.

Note: *priority* is associated with an interrupt, not with a particular action. If ON specifies a previously-on interrupt, the new *priority* is effected for all existing actions on that interrupt. For this reason the SUBR HANDLER is often preferable. (ON can leave the priority unchanged by GETing it out of the IHEADER.)

21.6. OFF

<OFF *ispec which*>

turns off an interrupt or an action and returns *ispec*. Exactly what happens depends on the *ispec* argument as follows:

If *ispec* is a name (STRING, ATOM, or IHEADER), OFF completely turns off the interrupt; all actions associated with that interrupt are thrown away. (An error occurs if the interrupt is not ON or has no HANDLERs.) The IHEADER is thrown away if nobody points to it. Even the interpreter does not respond to occurrences of the interrupt.

If *ispec* is a HANDLER, OFF removes it from its chain. There is no effect on any other HANDLERs associated with that interrupt.

which must be specified as a CHANNEL if and only if *ispec* is "CHAR" or CHAR!-INTERRUPTS!- . Caution: don't <OFF "CHAR" ,INCHAN> without ONing it again or MDL will become deaf.

which must be specified as a locative if and only if *ispec* is "READ" or "WRITE" (or one of the corresponding ATOMs).

21.7. DISABLE

<DISABLE *iheader*>

returns its argument and causes interrupts associated with *iheader* to be noticed by the interpreter but not passed on to the HANDLERs in the chain. This is inefficient, unless the HANDLER chain must be preserved.

21.8. ENABLE

<ENABLE *iheader*>

returns its argument and causes interrupts associated with *iheader* to be passed on to the HANDLERs.

21.9. Priorities and Interrupt Levels

At any given time there is a defined interrupt level. This is a FIX which determines which interrupts can really "interrupt" -- that is, cause the current processing to cease while their wants are satisfied.

Normal, non-interrupt programs operate at an interrupt level of 0. Processing done for interrupts is done at an interrupt level equal to the interrupt's priority.

21.9.1. Interrupt Processing

Interrupts "actually" occur only at well-defined points in time: during a call to a Subroutine, or at critical places within Subroutines (for example, during each iteration of MAPF on a LIST, which may be circular), or while a PROCESS is "BLOCKED" (see below). No interrupts can occur during garbage collection.

What actually happens when an enabled interrupt occurs is that the priority of the interrupt is compared with the current interrupt level, and the following is done:

If the priority is greater than the current interrupt level, the current processing is "frozen in its tracks" and processing of the action(s) specified for that interrupt begins.

If the priority is less than or equal to the current interrupt level, the interrupt occurrence is <u>queued</u> -- that is, the fact that it occurred is saved away for processing when the interrupt level becomes low enough.

When the processing of an interrupt's actions is completed, MDL usually (1) "acts as if" the previously-existing interrupt level is restored, and processing continues on what was left off (perhaps for no time duration); and (2) "acts as if" any queued interrupt occurrences actually occurred right then, in their original order of occurrence.

The value returned by the handler of an interrupt's HANDLER is ignored, unless it is of TYPE DISMISS (PRIMTYPE ATOM), in which case none of the remaining HANDLERs in the chain will be invoked.

The processing of an interrupt's actions can terminate prematurely if a handler calls the SUBR DISMISS (see below).

21.9.2. INT-LEVEL

The SUBR INT-LEVEL is used to examine and change the current interrupt level directly.

        <INT-LEVEL>

simply returns the current interrupt level.

        <INT-LEVEL *fix*>

changes the interrupt level to its argument and returns the <u>previously</u>-existing interrupt level.

If INT-LEVEL lowers the interrupt level, it does not "really" return until all queued occurrences of interrupts of priority higher than the target priority have been processed.

Setting the INT-LEVEL extremely high (for example, <INT-LEVEL <CHTYPE <MIN> FIX>>) effectively disables all interrupts (but occurrences of enabled interrupts will still be queued).

If LISTEN or ERROR is called when the INT-LEVEL is not zero, then the typeout will be

        LISTENING-AT-LEVEL *l* PROCESS *p* INT-LEVEL *i*

21.9.3. DISMISS

DISMISS permits a handler to return an arbitrary value for an arbitrary ACTIVATION at an arbitrary interrupt level. The call is as follows:

        <DISMISS *value:any activation int-level:fix*>

where only the *value* is required. If *activation* is omitted, return is to the place interrupted from. If *int-level* is omitted, the INT-LEVEL prior to the current interrupt is restored.

21.10. Specific Interrupts

Descriptions of the characteristics of particular "built-in" MDL interrupts follow. Each is named by its STRING name. Expect this list to be incomplete yesterday.

21.10.1. "CHAR"

"CHAR" is currently the most complex built-in interrupt. It occurs every time either

    (1) a character arrives from the (pseudo-)console or Network socket, or

    (2) a character is required by the pseudo-console or Network socket, or

    (3) a line-feed character is output or (under ITS) the operating system generates a screen-full interrupt

for the CHANNEL in the IHEADER. There is an IHEADER for each CHANNEL for which the "CHAR" interrupt is on.

Under ITS, even though the interrupt occurs in case (1) for each console character received, the operating system holds off receipt until an "activation" character is typed. The activation characters are ^@ through ^G, ^K, ^L, ^N through ^^, and DEL (that is, ASCII codes 0-7, 13, 14, 16-36, and 177 octal). Under TENEX, this interrupt occurs immediately. The operating system can be told which characters typed on a console should cause this interrupt to occur, by calling the SUBR ACTIVATE-CHARS with a STRING argument containing those characters. If called with no argument, ACTIVATE-CHARS returns a STRING containing the characters that currently interrupt. Initially, only ^G, ^S, and ^O interrupt. Pseudo-consoles are not available under TENEX.

An initial MDL already has "CHAR" enabled on ,INCHAN with priority 8 and a handler named QUITTER to run in process 0; this is how ^G and ^S are processed. In addition, every time a new CHANNEL is opened in "READ" mode to a console, a similar IHEADER and HANDLER are ONed for that new CHANNEL automatically. These automatically-generated interrupt handlers use the standard IHEADER and HANDLER machinery, and can be DISABLEd or OFFed at will. However, the IHEADER for ,INCHAN must not be OFFed: MDL knows that $ is typed only by an interrupt (!).

The handler supplied for an input "CHAR" interrupt on a real console must take two arguments: (1) the CHARACTER which was typed, and (2) the CHANNEL on which it was typed. The handler supplied for an output "CHAR" interrupt on a real console must take one or two arguments (using

"OPTIONAL" or "TUPLE"): if two arguments are supplied by the interrupt system, they are the line number (FIX) and the CHANNEL, respectively, and the interrupt is for a line-feed; if only one argument is supplied (only under ITS), it is the CHANNEL, and the interrupt is for a full console screen.

Example: the following causes the given message to be printed out whenever a ^Y is typed on .INCHAN:

```
<HANDLER <GET .INCHAN INTERRUPT>
       #FUNCTION ((CHAR CHAN)
        #DECL ((VALUE) ANY (CHAR) CHARACTER (CHAN) CHANNEL)
        <AND <==? .CHAR !\^Y>
              <PRINC " some of my best friends are ^Ys.">>)>$
#HANDLER #FUNCTION((CHAR CHAN) ...)
UVWX^Y some of my best friends are ^Ys.ZAB$
UVWX^YZAB
```

Note that occurrences of "CHAR" do not wait for the $ to be typed.

Under ITS, a "CHAR" interrupt can also be associated with a CHANNEL open to a pseudo-console ("STY" device and friends) for either input or output. An input CHANNEL will cause an interrupt when a character is available, and an output CHANNEL will cause an interrupt when the program at the other end needs a character (and the operating-system buffer is empty). These interrupts are enabled in exactly the same way as real-console interrupts, except that the handler gets applied to only one argument, the CHANNEL.

A "CHAR" interrupt can also be associated with a CHANNEL open to a Network socket ("NET" device) for either input or output. The handler gets applied to a NETSTATE array (which see) and the CHANNEL.

21.10.2. "GC"

"GC" occurs just after every garbage collection. Enabling this interrupt is the only way to tell from a program that a garbage collection has occurred. A handler for "GC" takes three arguments. The first is the time for garbage collection. This is a FLOAT indicating the number of seconds it took. The second argument is a FIX indicating the cause of the garbage collection, as follows (chapter 22):

0. Program called GC.
1. Movable storage was exhausted.
2. Control stack overflowed.
3. Top-level LVALs overflowed.
4. Global vector overflowed.
5. TYPE vector overflowed.
6. Immovable garbage-collected storage was exhausted.

7. Internal stack overflowed.

8. Both control and internal stacks overflowed (rare).

9. Pure storage was exhausted.

10. Second, exhaustive garbage collection occurred.

The third argument is an ATOM indicating what initiated the garbage collection: BLOAT, GROW, LIST, VECTOR, SET, SETG, FREEZE, GC, NEWTYPE, PURE-PAGE-LOADER (pure storage was exhausted), or INTERRUPT-HANDLER (stack overflow, unfortunately).

21.10.3. "DIVERT-AGC"

"DIVERT-AGC" ("Automatic Garbage Collection") occurs just before a deferrable garbage collection because of exhausted movable garbage-collected storage. Enabling this interrupt is the only way to tell from a MDL program that a garbage collection is about to occur. A handler takes two arguments: a FIX telling the number of words needed and an ATOM telling what initiated the garbage collection (above). If it wishes, a handler can avoid a garbage collection by calling BLOAT. If the pending request for garbage-collected storage cannot then be satisfied, a garbage collection occurs anyway. AGC-FLAG is SET to T while the handler is running, so that new storage requests do not try to cause a garbage collection.

21.10.4. "CLOCK"

"CLOCK", when enabled, occurs every half second (the ITS slow clock tick). It is not available under TENEX. It wants handlers which take no arguments. Example:

```
<ON "CLOCK" <FUNCTION () <PRINC "TICK ">> 1>
```

21.10.5. "BLOCKED"

"BLOCKED" occurs whenever any PROCESS (not only the PROCESS which may be in a HANDLER) starts waiting for console input; that is, it indicates that somewhere, somebody did a READ, READCHR, NEXTCHR, TYI, etc. to a console. The handler for a "BLOCKED" interrupt should take one argument, namely the PROCESS which started waiting (which will also be the PROCESS in which the handler runs, if none is in the HANDLER).

Example: the following will cause MDL to acquire a * prompt character.

```
<ON "BLOCKED" #FUNCTION ((IGNORE) <PRINC !\*>) 5>
```

21.10.6. "UNBLOCKED"

"UNBLOCKED" occurs whenever an $ (ESC) is typed on a console where a program was hanging, waiting for input, or when a TYI (which see) is satisfied. The handler it wants takes one argument: the CHANNEL on which the $ or character was typed.

21.10.7. "READ" and "WRITE"

"READ" and "WRITE" are associated with read or write access to MDL objects. These interrupts are often called "monitors", and enabling the interrupt is often called "monitoring" the associated object. "Read access" to an ATOM's local value includes applying BOUND? or ASSIGNED? to the ATOM; similarly for a global value and GASSIGNED?.

Monitors are set up with EVENT or ON, using an extra argument, which is a locative to the object being monitored, just as a CHANNEL is given for "CHAR". A handler for "READ" takes two arguments: the locative and the FRAME of the application that does the read. A handler for "WRITE" takes three arguments: the locative, the new value, and the FRAME. For example:

```
<SET A '(1 2 3)>
<SET B <AT .A 2>>
<ON "WRITE" <FUNCTION (OBJ VAL FRM)
        #DECL ((VALUE) FALSE (FRM) FRAME)
        <TERPRI>
        <PRINC "Attempt to smash ">
        <PRIN1 .OBJ>
        <PRINC " with ">
        <PRIN1 .VAL>
        <PRINC " from ">
        <PRIN1 .FRM>
        <TERPRI>>
        4 0 .B>
```

21.10.8. "SYSDOWN"

"SYSDOWN" occurs when a system-going-down or system-revived signal is received from ITS. It is not available under TENEX. If no HANDLERs exist, a warning is printed on the console. A handler takes one argument: a FIX giving the number of 30ths of a second until the end of the world (-1 for a reprieve).

186

21.10.9. "ERROR"

In an effort to simplify ERROR handling by programs, MDL has a facility allowing errors to be handled like interrupts. SETGing ERROR to a user function is a distasteful method, not safe if any bugs are around. An "ERROR" interrupt wants a handler that takes any number of arguments. When an error occurs, each handler will be applied to the FRAME of the ERROR call and the TUPLE of ERROR arguments. If a given handler "takes care of the error", it can ERRET with a value from the ERROR FRAME, after having done an <INT-LEVEL 0>. If no handler takes care of the error, it falls into the normal ERROR.

If an ERROR occurs at an INT-LEVEL greater than or equal to that of the "ERROR" interrupt, real ERROR will be called, because attempts to defer "ERROR" interrupts are illegal.

21.10.10. "IPC"

"IPC" occurs when a message is received on the ITS IPC device (chapter 23). It is not available under TENEX.

21.10.11. "INFERIOR"

"INFERIOR" occurs when an inferior ITS job interrupts the MDL job. It is not available under TENEX. A handler takes one argument: a FIX between 0 and 7 inclusive, telling which inferior is interrupting.

21.10.12. "RUNT" and "REALT"

These are not available under TENEX.

"RUNT", if enabled, occurs once, N seconds of MDL running time (CPU time) after calling <RUNTIMER N:fix-or-float>, which returns its argument. A handler takes no arguments.

"REALT", if enabled, occurs every N seconds of real-world time after calling <REALTIMER N:fix-or-float>, which returns its argument. A handler takes no arguments.

21.10.13. "Dangerous" Interrupts

Ideally, these interrupts should never occur. If one does, and only if no handlers are enabled for it, the argument(s) that would have gone to a handler go to ERROR instead. If any of them are OFFed (always, under TENEX), the interrupt goes to the superior job or fork instead.

"MPV" ("memory-protection violation") occurs if MDL tries to refer to a storage address not in its address space. "PURE" occurs if MDL tries to alter read-only storage. "ILOPR" occurs if MDL executes an illegal instruction ("operator"). "PARITY" occurs if the CPU detects a parity error in MDL's address space (it is fatal if not handled). All of these require a handler that takes one argument: the address (TYPE WORD) at which the offending instruction resides.

"IOC" occurs if MDL tries to deal illegally with an I/O channel. A handler must take two arguments: a three-element FALSE like OPEN might return, and the CHANNEL getting the error.

21.11. User-Defined Interrupts (INTERRUPT)

If the interrupt name given to EVENT or ON is not one of the standard predefined interrupts of MDL, they will gleefully create an IHEADER anyway, making the assumption that you are enabling a "user-defined interrupt".

User-defined interrupts are made to occur by applying the SUBR INTERRUPT, as in

        <INTERRUPT *name arg1 ... argN*>

where *name* is a STRING, ATOM or IHEADER, and *arg1* through *argN* are the arguments wanted by the handlers for the interrupt.

If the interrupt specified by INTERRUPT is enabled, INTERRUPT returns T; otherwise it returns #FALSE ( ). All the usual priority and queueing rules hold, so that even if INTERRUPT returns T, it is possible that nothing "really happened" (yet).

INTERRUPT can also be used to cause "artificial" occurrences of standard predefined MDL interrupts.

21.12. Waiting for Interrupts

21.12.1. HANG

        <HANG *pred*>

hangs interruptibly, without consuming any time, potentially forever. HANG is nice for a program that cannot do anything until an interrupt occurs. If the optional *pred* is given, it is evaluated every time an interrupt occurs and is dismissed back into the HANG; if the result of evaluation is not FALSE, HANG unhangs and returns it as a value. If *pred* is not given, there had better be a named ACTIVATION somewhere to which a handler can return.

21.12.2. SLEEP

<SLEEP *time:fix-or-float pred*>

hangs for *time* seconds interruptibly, where *time* is non-negative, and then returns T. *pred* is the same as for HANG.

Chapter 22. Storage Management

The reason this chapter comes so late in this document is that, except for special cases, MDL programs have their storage needs handled automatically. There is usually no need even to consider storage management, except as it affects efficiency (chapter 24). This chapter gives some explanation of why this is so, and covers those special means by which a program can assume control of storage management.

The MDL address space is divided into five parts, which are usually called

- (1) movable garbage-collected space,
- (2) immovable space (both garbage-collected and not),
- (3) user pure/page space,
- (4) pure-RSUBR mapping space, and
- (5) internal storage.

Internal storage occupies both the highest and lowest addresses in the address space, and its size never changes as MDL executes. The other spaces can vary in size according to the needs of the executing program. Generally the interpreter allocates a contiguous set of addresses for each space, and each space gradually fills up as new objects are created and as disk files are mapped in. The action taken when a space becomes full varies, as discussed below.

22.1. Movable Garbage-collected Storage

Most storage used explicitly by MDL programs is obtained from a pool of free storage managed by a "garbage collector". Storage is obtained from this pool by the SUBRs which construct objects. When such a SUBR finds that the pool of available storage is exhausted, it automatically calls the garbage collector.

The garbage collector examines the storage pool and separates all the objects there into two classes: those which can possibly be referenced by a program, and those which cannot. The former are then compacted (moved) into one section of the pool, and the remainder of the pool is made available for new constructed objects. A garbage collection can be normal or exhaustive, the difference being whether certain kinds of storage that require complicated treatment (for example, associations) are

reclaimed. An exhaustive garbage collection occurs every 32nd time, or possibly when the garbage collector is explicitly called by a program (see below), or when a normal garbage collection cannot satisfy the storage request.

If the request for more storage still cannot be satisfied from reclaimed storage, the garbage collector will attempt to obtain more total storage from the operating system under which MDL runs. (Also, if there is a gross superfluity of storage space, the garbage collector will politely return some storage to the operating system.) Only when the total system resources are exhausted will you finally lose.

Thus, if you just "forget about" a datum, that is, lose all possible means of referencing it, its storage area is automatically reclaimed. "Datum" in this context includes that stack-structured storage space used in PROCESSes for functional application.

22.1.1. Stacks and Other Internal Vectors

Control stacks are used in MDL to control the changes in environment caused by calling and binding. Each active PROCESS has its own control stack. On this stack are stored LVALs for ATOMs; PRIMTYPE TUPLEs, which are otherwise like VECTORs; PRIMTYPE FRAMEs, which are generated by calling Subroutines; and ACTIVATIONs, which are generated by calling FUNCTIONs with named ACTIVATIONs, PROG, and REPEAT. TAG and LLOC can make TAGs and LOCDs (respectively) that refer to a specific place on a specific control stack. (LEGAL? returns T if and only if the portion of the control stack in which its argument is found or to which its argument refers is still active, or if its argument doesn't care about the control stack. The garbage collector may change a non-LEGAL? object to TYPE ILLEGAL before reclaiming it.) As the word "stack" implies, things can be put on it and removed from it at only one end, called the top. It has a maximum size (or depth), and attempting to put too many things on it will cause overflow. A stack is stored like a vector, and it must be GROWn if and when it overflows.

A control stack is actually two stacks in one. One section is used for "top-level" LVALs -- those SET while the ATOM is not bound by any active Function's argument LIST or Subroutine's SPECIAL binding -- and the other section is used for everything else. Either section can overflow, of course. The top-level-LVAL section is below the other one, so that a top-level LVAL will be found only if the ATOM is not currently bound elsewhere, namely in the other section.

MDL also has an internal stack, used for calling and temporary storage within the interpreter and compiled programs. It too is stored like a vector and can overflow. There are other internal vectors that can overflow: the "global vector" holds pairs ("slots") of ATOMs and corresponding GVALs ("globally bound" means that the ATOM in question is in this vector, whether or not it currently has a global value), and the "TYPE vector" holds TYPE names (predefined and NEWTYPEs) and how they are to be treated.

22.2. Immovable Storage

22.2.1. Garbage-collected: FREEZE

In very special circumstances, such as debugging RSUBRs, you may need to prevent an object from being moved by the garbage collector. FREEZE takes one argument, of PRIMTYPE VECTOR, UVECTOR, STRING or TUPLE. It copies its argument into non-moving garbage-collected space. FREEZE returns the copy CHTYPEd to its PRIMTYPE, except in the case of a TUPLE, which is changed to a VECTOR.

22.2.2. Non-garbage-collected: STORAGE (the PRIMTYPE)

An object of PRIMTYPE STORAGE is really a frozen UVECTOR whose UTYPE is of PRIMTYPE WORD, but it is always pointed to by something internal to MDL and thus is never garbage-collectible. The use of FREEZE is always preferable, except when for historical reasons a STORAGE is necessary, for example when displaying PICTUREs: the principle use of STORAGE is in communicating with the Evans and Sutherland Display, which autonomously accesses primary storage and must run concurrently with garbage collection.

22.3. Other Storage

User pure/page space serves two purposes. First, when a user program PURIFYs (see below) MDL objects, they are copied into this space. Second, so-called hand-crafted RSUBRs (assembled but not compiled) can call on the interpreter to map pages of data files into this space for arbitrary purposes.

Pure-RSUBR mapping space is used by the interpreter to dynamically map pages of pure compiled programs into and out of the MDL address space. Pure code can refer to impure storage through the "transfer vector", another internal vector. This space is the most vulnerable to being compressed in size by the long-term growth of other spaces.

Internal storage has both pure and impure parts. The interpreter program itself is pure and sharable, while impure storage is used for internal pointers, counters, and flags, for example, pointers to the boundaries of other spaces. In the pure part of this space are most of the ATOMs in an initial MDL, along with their OBLIST buckets (LISTs) and GVAL slots (a pure extension of the global vector), where possible. A SET or SETG of a pure ATOM automatically impurifies the ATOM and as much of its OBLIST bucket as needs to be impure.

22.4. Garbage Collection: Details

When either of the garbage-collected spaces (movable or immovable) becomes full, MDL goes through the following procedure:

(1) A "DIVERT-AGC" interrupt occurs if the garbage collection can be deferred temporarily by shifting boundaries between storage spaces slightly. The interrupt handler may postpone a garbage collection by moving boundaries itself with a call to BLOAT (below).

(2) The garbage collector begins execution. It creates an inferior job (under ITS, named AGC) or fork (under TENEX) whose address space is used to hold the new copies of non-garbage objects. MDL accesses the inferior's address space through two pages ("frontier" and "window") in its internal space that are shared with the inferior.

(3) The garbage collector marks and moves all objects that can possibly be referenced hereafter. It begins with the <MAIN> PROCESS and the currently running PROCESS <ME>, considered as vectors containing the control stacks, object pointers in live registers, etc. Every object in these "process vectors" is marked "accessible", and every element of these objects (bindings, etc.), and so on recursively.

(4) If the garbage collection is "exhaustive", then both the chain of associations and top-level local/global bindings are marked last, which takes more time but is more likely to uncover garbage therein. In a normal garbage collection these constructs are not treated specially.

(5) Finally, the inferior job/fork's address space is mapped into MDL's own, replacing old garbagey storage with the new compact storage, and the inferior is destroyed.

22.5. GC

> <GC min:fix exh?:false-or-any>

causes the garbage collector to run and returns the total number of words of storage reclaimed. Both of its arguments are optional; if they are not supplied, a call to GC simply causes a normal garbage collection.

If *min* is explicitly supplied as an argument, a garbage-collection parameter is changed permanently before the garbage collector runs. *min* is the smallest number of words of "free" (unclaimed, available for use) movable garbage-collected storage the garbage collector will be satisfied with having after it is done. Initially it is 8192 words. If the total amount of reclaimed storage is less than *min*, it will ask the operating system for enough storage (in 1024-word blocks) to make it up. N.B.: the system may be incivil enough not to grant the request; in that case, the garbage collector will be content with what it has, unless that is not enough to satisfy a pending request for storage.

Then it will inform you that it is losing. A large *min* will result in fewer total garbage collections, but they will take longer since the total quantity of storage to be scanned will generally be larger. Smaller *mins* result in shorter, more frequent garbage collections.

exh? tells whether or not the garbage collection should be exhaustive. It is optional, with default a FALSE.

22.6. BLOAT

BLOAT is used to cause a temporary expansion of the available storage space with or without changing the garbage-collection parameters. BLOAT is particularly useful for avoiding unnecessary garbage collections when loading a large file. It will cause (at most) one garbage collection, at the end of which the available storage will be at least the amount specified in BLOAT's application. (Unless, of course, the operating system is cranky and will not provide the storage. Then you will get an error. <ERRET 1> from this error will cause the BLOAT to return 1, which usually just causes you to lose at a later time -- unless the operating system feels nicer when the storage is absolutely necessary.)

A call to BLOAT looks like this:

```
<BLOAT fre stk lcl glb typ sto pstk
       min plcl pglb ptyp imp pur dpstk dstk>
```

where all arguments on the first line are FIX, optional (default 0), and indicate the following:

fre: number of words of free movable storage desired (for LISTs, VECTORs, ATOMs, etc.)

stk: number of words of free control-stack space desired (for functional applications and binding of ATOMs)

lcl: number of new top-level LVALs for which to leave space (SETs of ATOMs which are not currently bound)

glb: number of new GVALs for which to leave space (in the global vector)

typ: number of new TYPE definitions for which to leave space (in the TYPE vector)

sto: number of words of immovable garbage-collected storage desired

pstk: number of words of free internal-stack space desired (for READing large STRINGs, and calling routines within the interpreter and compiled programs)

194

Arguments on the second line are also FIX and optional, but they set garbage-collection parameters permanently, as follows:

min: as for GC

plc: number of slots for LVALs added when the space for top-level LVALs is expanded (initially 64)

pglb: number of slots for GVALs added when the global vector is grown (initially 64)

ptyp: number of slots for TYPEs added when the TYPE vector is grown (initially 32)

imp: number of words of immovable garbage-collected storage added when it is expanded (initially 1024)

pur: number of words reserved for pure compiled programs, if possible (initially 0)

dpstk: most desirable size for the internal stack, to prevent repeated shrinking and GROWing (initially 512)

dstk: most desirable size for the control stack (initially 4096)

BLOAT returns the actual number of words of free movable garbage-collected storage available when it is done.

22.7. BLOAT-STAT

BLOAT-STAT can be used with BLOAT to "tune" the garbage collector to particular program requirements.

        <BLOAT-STAT *length-27:uvector*>

fills the *uvector* with information about the state of storage of MDL. The argument should be a UVECTOR of length 27 and UTYPE FIX. If BLOAT-STAT does not get an argument, it will provide its own UVECTOR. The information returned is as follows: the first 8 elements indicate the number of garbage collections that are attributable to certain causes, and the other 19 give information about certain areas of storage. In detail:

1. number of garbage collections caused by exhaustion of movable garbage-collected storage
2. ditto by overflow of control stack(s)
3. ditto by overflow of top-level-LVAL section of control stack(s)
4. ditto by overflow of global vector
5. ditto by overflow of TYPE vector

6. ditto by exhaustion of immovable garbage-collected storage
7. ditto by overflow of internal stack
8. ditto by overflow of both stacks at the same time (rare)

9. number of words of movable storage
10. number of words of movable storage used since last BLOAT-STAT
11. maximum number of words of movable storage ever existing
12. number of words of movable storage used since MDL began running
13. maximum size of control stack
14. number of words on control stack in use
15. maximum size of control stack(s) ever reached
16. number of slots for top-level LVALs
17. number of top-level LVALs existing
18. number of slots for GVALs in global vector
19. number of GVALs existing
20. number of slots for TYPEs in TYPE vector
21. number of TYPEs existing
22. number of words of immovable garbage-collected storage
23. number of words of immovable storage unused
24. size of largest unused contiguous immovable-storage block
25. number of words on internal stack
26. number of words on internal stack in use
27. maximum size of internal stack ever reached

22.8. GC-MON

<GC-MON *pred*>

("garbage-collector monitor") determines whether or not the interpreter will hereafter print information on the console when a garbage collection starts and finishes, according to whether or not its argument is true. It returns the previous state. Calling it with no argument returns the current state. The initial state is false.

When typing is enabled, the garbage collector prints, when it starts:

GIN *reason subr-that-caused:atom*

and, when it finishes:

GOUT *seconds-needed*

22.9. Related Subroutines

Two SUBRs, described next, use only part of the garbage-collector algorithm, in order to find all pointers to an object. GC-DUMP and GC-READ, as their names imply, also use part in order to translate between MDL objects and binary representations thereof.

22.9.1. SUBSTITUTE

                <SUBSTITUTE *new:any old:any*>

returns *old*, after causing a miniature garbage collection to occur, during which <u>all</u> references to *old* are changed so as to refer to *new*. Neither argument can be of PRIMTYPE STRING or BYTES or LOCD or live on the control stack, unless both are of the same PRIMTYPE. One TYPE name cannot be substituted for another. One of the few legitimate uses for it is to substitute the "right" ATOM for the "wrong" one, after OBLISTs have been in the wrong state. This is more or less the way ATOMs are impurified. It is also useful for unlinking RSUBRs. SUBSTITUTE returns *old* as a favor: unless you hang onto *old* at that point, it will be garbage-collected.

22.9.2. PURIFY

                <PURIFY *any-1 ... any-n*>

returns its last argument, after causing a miniature garbage collection that results in all the arguments becoming pure and sharable, and ignored afterward by the garbage collector. No argument can live on the control stack or be of PRIMTYPE PROCESS or LOCD or ASOC. Sharing between jobs or forks actually occurs after a SAVE, if and when the SAVE file is RESTOREd.

Chapter 23. MDL as an ITS Job

This chapter treats MDL considered as executing in an ITS job, and interactions between MDL and other ITS jobs. See also section 21.10.11. Unless otherwise indicated, none of the features in this chapter is available under TENEX.

23.1. TIME

This SUBR is available under TENEX. TIME takes any number of arguments, which are evaluated but ignored, and returns a FLOAT giving the number of seconds of CPU time MDL has used so far. It is often used in debugging to examine the values of its arguments, by having MDL's superior job or fork (say, DDT) plant a breakpoint in the code for TIME.

23.2. UNAME

<UNAME> returns a STRING of up to six characters which is the "user name" (login identification) of MDL's job. The characters belong to the "sixbit" or "printing" subset of ASCII, namely those between <ASCII *40*> and <ASCII *137*> inclusive.

23.3. JNAME

<JNAME> returns a "sixbit" STRING of up to six characters which is the "job name" of MDL's job.

23.4. LOGOUT

        <LOGOUT>

attempts to log out the job in which it is executed. It will succeed only if the MDL is the top-level

job, that is, it is running disowned or as a daemon. If it succeeds, it of course never returns. If it does not, it returns #FALSE ( ).

23.5. VALRET

        <VALRET *string*>

("value return") seldom returns. It passes control back up the ITS job tree to the superior of MDL, passing *string* as a message to that superior. If it does return, the value is #FALSE ( ).

23.6. QUIT

        <QUIT>

causes your MDL to die in an orderly manner. Under ITS, it is equivalent to <VALRET ":KILL ">. Under TENEX, <QUIT> is equivalent to a control-C signal, and control passes to the superior fork.

23.7. DEMSIG

        <DEMSIG *daemon:string*>

signals to ITS that the daemon named by its argument should run now. It returns T if the daemon exists, #FALSE ( ) otherwise.

23.8. Inter-job Communication

The IPC ("interprocess communication") device is treated as an I/O device by ITS but not explicitly so by MDL; that is, it is never OPENed. It allows MDL to communicate to other ITS jobs by means of sending and receiving messages. A job identifies itself as sender or recipient of a message with an ordered pair of "sixbit" STRINGs, which are often but not always <UNAME> and <JNAME>. A message has a "body" and a "type".

23.8.1. SEND and SEND-WAIT

        <SEND *othern1 othern2 body type myname1 myname2*>

```
<SEND-WAIT othern1 othern2 body type myname1 myname2>
```

both send an IPC message to any job that is listening for it as *othern1 othern2*. *body* must be either a STRING, or a UVECTOR of objects of PRIMTYPE WORD. *type* is an optional FIX, with default 0, which is part of the information the other guy receives. The last two arguments are from whom the message is to be sent. These are optional and default to <UNAME> and <JNAME> respectively. SEND returns a FALSE if no one is listening, while SEND-WAIT hangs until someone wants it. Both return T if someone accepts the message.

23.8.2. The "IPC" Interrupt

When your MDL job receives an IPC message, "IPC" occurs (chapter 21). A handler is called with either four or six arguments gleaned from the received message. *body, type, othern1*, and *othern2* are always supplied. *myname1* and *myname2* are supplied only if they are not this job's <UNAME> and <JNAME>.

There is a default HANDLER for the "IPC" interrupt, with a handler named IPC-HANDLER and 0 in the process slot. The handler prints out on the console the *body*, whom it is from, the *type* if not 0, and whom it is to if not <UNAME> <JNAME>. If the *type* is 1 and the *body* is a STRING, then, after the message information is printed out, the STRING is PARSEd and EVALuated.

23.8.3. IPC-OFF

<IPC-OFF> stops all listening on the IPC device.

23.8.4. IPC-ON

```
<IPC-ON myname1 myname2>
```

causes listening on the IPC device as *myname1 myname2*. If no arguments are provided, listening is on <UNAME> <JNAME>. When a message arrives, "IPC" occurs.

MDL is initially listening as <UNAME> <JNAME> with the default HANDLER set up on the "IPC" interrupt with a priority of 1.

200

Chapter 24. Efficiency and Tastefulness

24.1. Efficiency

Actually, you make MDL programs efficient by thinking hard about what they really make the interpreter do, and making them do less. Some guidelines, in order of decreasing expense:

(1) Free storage is expensive.
(2) Calling functions is expensive.
(3) PROG and REPEAT are expensive.

Explanation:

(1) Unnecessary use of free storage (creating needless LISTs, VECTORs, UVECTORs, etc.) will cause the garbage collector to run more often. This is expensive!! A fairly large MDL (for example, 60 000 36-bit words) can take ten seconds of PDP-10 CPU time to garbage-collect. Be especially wary of constructions like (0). Every time that is evaluated, it creates a new one-element LIST; it is too easy to write such things when they aren't really necessary. Unless you are doing PUTs or PUTRESTs on it, use '(0) instead.

(2) Sad, but true. Also generally ignored. If you call a function only once, or if it is short (less than one line), you are much better off in speed if you substitute its body in by hand. On the other hand, you may be much worse off in modularity. There are techniques for combining several FUNCTIONs into one RSUBR (with RSUBR-ENTRYs), either during or after compilation, and for changing FUNCTIONs into MACROs.

(3) PROG is almost never necessary, given (a) "AUX" in FUNCTIONs; (b) the fact that FUNCTIONs can contain any number of FORMs; (c) the fact that COND clauses can contain any number of FORMs; and (d) the fact that new variables can be generated and initialized by REPEAT.

By the way, REPEAT is faster than GO in a PROG. The <GO x> FORM has to be separately interpreted, right? In fact, if you organize things properly you very seldom need a GO; using GO is generally considered "bad style", but in some cases it's needed. Very few.

In many cases, a REPEAT can be replaced with a MAPF or MAPR, or an ILIST, IVECTOR, etc. of the form

```
<ILIST .N '<SET X <+ .X 1>>
```

which generates an N-element LIST of successive numbers starting at X+1.

Whether a program is interpreted or compiled, the first two considerations mentioned above hold: garbage collection and function calling remain expensive. Garbage collection is, clearly, exactly the same. Function calling is relatively more expensive. However, the compiler careth not whether you use REPEAT, GO, PROG, ILIST, MAPF, or whatnot: it all gets compiled into practically the same thing. However, the REPEAT or PROG will be slower if it has an ACTIVATION that is SPECIAL or used other than by RETURN or AGAIN.

24.1.1. Example

There follows an example of a FUNCTION that does many things wrong. It is accompanied by commentary, and two better versions of the same thing. (This function actually occurred in practice. Needless to say, names are withheld to protect the guilty.)

Blunt comment: this is terrible. Its purpose is to output the characters needed by a graphics console to draw lines connecting a set of points. The points are specified by two input lists: X values and Y values. The output channel is the third argument. The actual characters for each line are returned in a LIST by the function TRANS.

```
<DEFINE PLOTVDSK (X Y CHN "AUX" L LIST)
    <COND (<NOT <==? <SET L <LENGTH .X>><LENGTH .Y> >>
            <ERROR "LENGTHS NOT EQUAL">)>
    <SET LIST (29)>
    <REPEAT ((N 1))
        <SET LIST (!.LIST !<TRANS <.N .X> <.N .Y>>)>
        <COND (<G? <SET N <+ .N 1>> .L><RETURN .N>)> >
    <REPEAT ((N 1) (L1 <LENGTH .LIST>))
        <PRINC <ASCII <.N .LIST>> .CHN>
        <COND (<G? <SET N <+ .N 1>> .L1>
                <RETURN "DONE">)>      >>
```

Comments:

(1) LIST is only temporarily necessary. It is just created and then thrown away.

(2) Worse, the construct (!.LIST !<TRANS ...>) copies the previous elements of LIST every time it is executed!!!

(3) Indexing down the elements of LIST as in <.N .LIST> takes a long time, if the LIST is long. <3 ...> or <4 ...> is not worth worrying about, but <10 ...> is, and <100 ...> takes quite a while. Even if the indexing were not phased out, the compiler would be happier with <NTH .LIST .N>.

(4) The variable CHN is unnecessary if OUTCHAN is bound to the argument CHANNEL.

(5) It is tasteful to call ERROR in the same way that F/SUBRs do. This includes using an ATOM from the ERRORS OBLIST (if one is appropriate) to tell what is wrong, and it includes identifying yourself.

So, do it this way:

```
<DEFINE PLOTVDSK (X Y OUTCHAN)
#DECL ((OUTCHAN) <SPECIAL CHANNEL>)
<COND (<NOT <==? <LENGTH .X> <LENGTH .Y>>
        <ERROR VECTOR-LENGTHS-DIFFER!-ERRORS PLOTVDSK>)>
<PRINC <ASCII 29>>
<REPEAT ()
        <COND (<EMPTY? .X> <RETURN "DONE">)>
        <REPEAT ((OL <TRANS <1 .X> <1 .Y>>))
                <PRINC <ASCII <1 .OL>>>
                <COND (<EMPTY? <SET OL <REST .OL>>>
                        <RETURN>)>>
        <SET X <REST .X>>
        <SET Y <REST .Y>>>>
```

Of course, if you know how long is the LIST that TRANS returns, you can avoid using the inner REPEAT loop and have explicit PRINCs for each element. This can be done even better by using MAPF, as in the next version, which does exactly the same thing as the previous one, but uses MAPF to do the RESTing and the end conditional:

```
<DEFINE PLOTVDSK (X Y OUTCHAN)
#DECL ((OUTCHAN) <SPECIAL CHANNEL>)
<COND (<NOT <==? <LENGTH .X> <LENGTH .Y>>
        <ERROR VECTOR-LENGTHS-DIFFER!-ERRORS PLOTVDSK>)>
<PRINC <ASCII 29>>
<MAPF <>
    #FUNCTION ((XE YE)
                <MAPF <> #FUNCTION ((T) <PRINC <ASCII .T>>) <TRANS .XE .YE>>)
    .X
    .Y>
"DONE">
```

24.2. Creating a LIST in Forward Order

If you must create the elements of a LIST in sequence from first to last, you can avoid copying earlier ones when adding a later one to the end. One way is to use MAPF or MAPR or STACKFORM with a first argument of ,LIST: the elements are put on the control stack rather than in free storage, until the final call to LIST. If you know how many elements there will be, you can put them on the control stack yourself, in a TUPLE built for that purpose. Another way is used when REPEAT is necessary:

```
<REPEAT ((FIRST (T)) (LAST .FIRST) ...)
        #DECL ((VALUE FIRST LAST) LIST ...)
        ...
        <SET LAST <REST <PUTREST .LAST (.NEW)>>>
        ...
        <RETURN <REST .FIRST>>
        ...>
```

Here, .LAST always points to the current last element of the LIST. Because of the order of evaluation, the <SET LAST ...> could also be written <PUTREST .LAST <SET LAST (.NEW)>>.

24.3. Read-only Free Variables

If a Function uses the value of a free variable (<GVAL *unmanifest:atom*> or <LVAL *special:atom*>) without changing it, the compiled version may be more efficient if the value is assigned to a dummy UNSPECIAL ATOM in the Function's "AUX" list. This is true because an UNSPECIAL ATOM gets compiled into a slot on the control stack, which is accessible very quickly. The tradeoff is probably worthwhile if a *special* is referenced more than once, or if an *unmanifest* is referenced more than twice. Example:

```
<DEFINE MAP-LOOKUP (THINGS "AUX" (DB ,DATA-BASE))
        #DECL ((VALUE) VECTOR (THINGS DB) <UNSPECIAL <PRIMTYPE LIST>>)
        <MAPF ,VECTOR <FUNCTION (T) <MEMQ .T .DB>> .THINGS>>
```

24.4. Global and Local Values

In the interpreter the sequence ,X .X ,X .X is slower than ,X ,X .X .X because of interference between the GVAL and LVAL mechanisms (appendix 1). Thus it is not good to use both the GVAL and LVAL of the same ATOM frequently, unless references to the LVAL will be compiled away (made into control stack references).

204

24.5. Making Offsets for Arrays

It is often the case that you want to attach some meaning to each element of an array and access it independently of other elements. Firstly, it is a good idea to use names (ATOMs) rather than integers (FIXes) for offsets into the array, to make future changes easier. Secondly, it is a good idea to use the GVALs of the name ATOMs to remember the actual FIXes, so that the ATOMs can be MANIFEST for the compiler's benefit. Thirdly, to establish the GVALs, both the interpreter and the compiler will be happier with <SETG *name fix*> rather than <DEFINE *name* (X) <*fix* .X>> for read-only elements.

24.6. Tables

There are several ways in MDL to store a table, that is, a collection of (names and) values that will be searched. Unsurprisingly, choosing the best way is often dictated by the size of the table and/or the nature of the (names and) values.

For a small table, the names and values can be put in (separate) structures -- the choice of LIST or array being determined by volatility and limitability -- which are searched using MEMQ or MEMBER. This method is very space-efficient. If the table gets larger, and if the elements are completely orderable, a (uniform) vector can be used, kept sorted, and searched with a binary search.

For a large table, where reasonably efficient searches are required, a hashing scheme is probably best. Two methods are available in MDL: associations and OBLISTs.

In the first method, PUTPROP and GETPROP are used, which are very fast. The number of hashing buckets is fixed. Duplicates are eliminated by ==? testing. If it is necessary to use =? testing, or to find all the entries in the table, you can duplicate the table in a LIST or array, to be used only for those purposes.

In the second method, INSERT and LOOKUP on a specially-built OBLIST are used. (If the names are not STRINGs, they can be converted to STRINGs using UNPARSE, which takes a little time.) The number of hashing buckets can be chosen for best efficiency. Duplicates are eliminated by =? testing. MAPF/R can be used to find all the entries in the table.

24.7. Nesting

The beauty of deeply-nested control structures in a single FUNCTION is definitely in the eye of the beholder. (PPRINT, a preloaded RSUBR, finds them trying. However, the compiler often produces better code from them.) If you don't like excessive nesting, then you will agree that

```
<SET X ...>
<COND (<0? .X> ...) ...>
```

looks better than

```
<COND (<0? <SET X ...>> ...) ...>
```

and that

```
<REPEAT ...
        <COND ...
              (... <RETURN ...>)>
        ...
        ...>
```

looks better than

```
<REPEAT ...
        <COND ...
              (... <RETURN ...>)
              (ELSE ...)>
        ...>
```
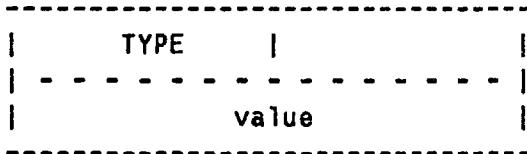
You can see the nature of the choices. Nesting is still and all better than GO.

Appendix 1. A Look Inside

This appendix tells about the mapping between MDL objects and PDP-10 storage -- in other words, the way things look "on the inside". None of this information is essential to knowing how to program in MDL, but it does give some reasons for capabilities and restrictions that otherwise you have to memorize. The notation and terminology get a little awkward in this discussion, because we are in a twilight zone between the worlds of MDL objects and of bit patterns. In general the words and phrases appearing in diagrams refer to bit patterns not MDL objects. A lower-case word (like "tuple") refers to the storage occupied by an object of the corresponding PRIMTYPE (like TUPLE).

First some terminology needs discussion. The sine qua non of any MDL object is a pair of 36-bit computer words. In general, lists consist of pairs chained together by pointers (addresses), and vectors consist of contiguous blocks of pairs. ==? essentially tests two pairs to see whether they contain the same bit patterns.

The first (lower-addressed) word of a pair is called the TYPE word, because it contains a numeric TYPE code that represents the object's TYPE. The second (higher-addressed) word of a pair is called the value word, because it contains (part of or the beginning of) the "data part" of the object. The TYPE word (and sometimes the value word) is considered to be made of a left half and a right half. We will picture a pair like this:

```
 ----------------------------------
|      TYPE     |                  |
| - - - - - - - - - - - - - - - -  |
|            value                 |
 ----------------------------------
```

where a vertical bar in the middle of a word means the word's halves are used independently. You can see that the TYPE code is confined to the left half of the TYPE word. (Half-)words are sometimes subdivided into fields appropriate for the context; fields are also pictured as separated by vertical bars. The right half of the TYPE word is used for different purposes depending on the TYPE of the object and actual location of the value.

Actually the 18-bit TYPE field is further decoded. The high-order (leftmost) bit is the mark bit, used exclusively by the garbage collector when it runs. The next two bits are monitor bits, used to cause "READ" and "WRITE" interrupts on read and write access to the pair. The next bit is used to differentiate between list elements and vector dope words. The next bit is unused but could be used in the future for an "execute" monitor. The remaining 13 bits specify the actual TYPE code. What CHTYPE does is to copy the pair and put a new TYPE code in the new pair.

Each data TYPE (predefined and NEWTYPEs) must belong to one of about 25 "storage allocation classes" (roughly corresponding to MDL PRIMTYPEs). These classes are characterized primarily by the manner in which the garbage collector treats them. Some of these classes will now be described.

"One Word"

This class includes all data that are not pointers to some kind of structure. All external (program-available) TYPEs in this class are of PRIMTYPE WORD. Example:

```
-----------------------------------
|      FIX       |       0        |
| - - - - - - - - - - - - - - - - |
|               105               |
-----------------------------------
```

"Two Word"

The members of this class are all 18-bit pointers to list elements. All external TYPEs in this class are of PRIMTYPE LIST. Example:

```
-----------------------------------
|      LIST      |       0        |
| - - - - - - - - - - - - - - - - |
|       0        |    pointer     |
-----------------------------------
```

where pointer is a pointer to the first list element. If there are no elements, pointer is zero; thus empty objects of PRIMTYPE LIST are ==? if their TYPEs are the same.

"Two N Word"

Members of this class are all "counting pointers" to blocks of two-word pairs. The right half of a counting pointer is an address, and the left half is the negative of the number of 36-bit words in the block. (This format is tailored to the PDP-10 AOBJN instruction.) The number of pairs in the block (LENGTH) is half that number, since each pair is two words. All external TYPEs in this class are of PRIMTYPE VECTOR. Example:

```
-----------------------------------
|     VECTOR     |       0        |
| - - - - - - - - - - - - - - - - |
|   -2*length    |    pointer     |
-----------------------------------
```

where length is the LENGTH of the VECTOR and pointer is the location of the start (the element selected by an NTH argument of 1) of the VECTOR.

"N Word"

This class is the same as the previous one, except that the block contains objects all of the same TYPE without individual TYPE words. The TYPE code for all the elements is in vector dope words, which are at addresses just larger than the block itself. All external TYPEs in this class are of PRIMTYPE UVECTOR. Example:

```
--------------------------------
|    UVECTOR    |      0       |
| - - - - - - - - - - - - - - - |
|    -length    |   pointer    |
--------------------------------
```

where length is the LENGTH of the UVECTOR and pointer points to the beginning of the UVECTOR.

"Byte String" and "Character String"

These two classes are almost identical. Byte strings are byte pointers to strings of arbitrary-size bytes. PRIMTYPE BYTES is the only member of this class. Character strings are byte pointers to strings of ASCII characters. PRIMTYPE STRING is the only member of this class. Both of these classes consist of a length and a PDP-10 byte pointer. In the case of character strings, the byte-size field in the byte pointer is always seven bits per byte (hence five bytes per word). Example:

```
--------------------------------
|   STRING    |    length      |
| - - - - - - - - - - - - - - - |
|         byte-pointer          |
--------------------------------
```

where length is the LENGTH of the STRING (in bytes) and byte-pointer points to a byte just before the beginning of the string (an ILDB instruction is needed to get the first byte). A newly-created STRING always has *010700* in the left half of byte-pointer. Unless the string was created by SPNAME, byte-pointer points to a uvector, where the elements (characters) of the STRING are stored, packed together five to a word.

"Frame"

This class gives the user program a handle on its control and variable-access structures. All external TYPEs in this class are of PRIMTYPE FRAME. Three numbers are needed to designate a frame: a unique 18-bit identifying number, a pointer to the frame's storage on a control stack, and a pointer to the PROCESS associated with the frame. Example:

```
---------------------------------
|    FRAME      |process-pointer|
| - - - - - - - - - - - - - - - |
|   unique-id   | frame-pointer |
---------------------------------
```

where process-pointer points to the dope words of a PROCESS vector, and unique-id is used for validating (testing LEGAL?) the frame-pointer, which points to a frame for some Subroutine call on the control stack.

"Tuple"

A tuple pointer is a counting pointer to a vector on the control stack. It may be a pointer to the arguments to a Subroutine or a pointer generated by the "TUPLE" declaration in a FUNCTION. Like objects in the previous class, these objects contain a unique identifying number used for validation. PRIMTYPE TUPLE is the only member of this class. Example:

```
---------------------------------
|    TUPLE      |   unique-id   |
| - - - - - - - - - - - - - - - |
|    -2*length  |    pointer    |
---------------------------------
```
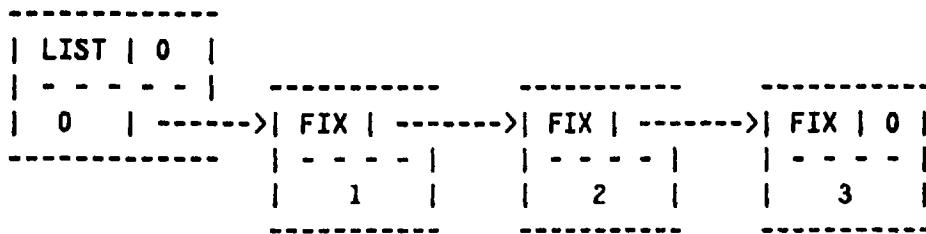

Other Storage Classes

The rest of the storage classes include strictly internal TYPEs and pointers to special kinds of lists and vectors like locatives, ATOMs and ASOCs. A pair for any LOCATIVE except a LOCD looks like a pair for the corresponding structure, except of course that the TYPE is different. A LOCD pair looks like a tuple pair and needs a word and a half for its value; the unique-id refers to a binding on the control stack or to the "global stack" if zero. Thus LOCDs are in a sense "stack objects" and are more restricted than other locatives.

Basic Data Structures
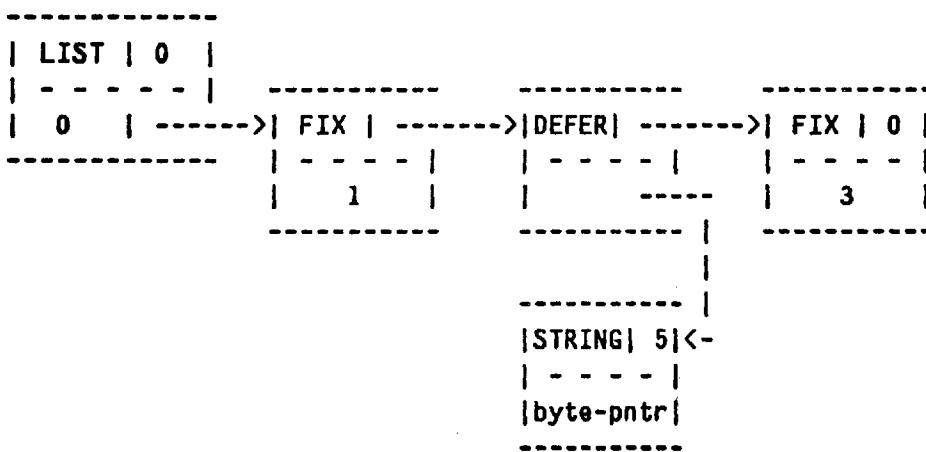
Lists

List elements are pairs linked together by the right halves of their first words. The list is terminated by a zero in the right half of the last pair. For example the LIST (1 2 3) would look like this:

```
--------------
| LIST | 0  |
| - - - - - |      ------------      ------------      ------------
| 0    | ------>| FIX | -------->| FIX | -------->| FIX | 0 |
--------------   | - - - - |      | - - - - |      | - - - - |
                 |   1   |        |   2   |        |   3   |
                 ------------      ------------      ------------
```

The use of pointers to tie together elements explains why new elements can be added easily to a list, how sharing and circularity work, etc. The links go in only one direction through the list, which is why a list cannot be BACKed or TOPped: there's no way to find the RESTed elements.

Since some MDL values require a word and a half for the value in the pair, they do not fit directly into list elements. This problem is solved by having "deferred pointers". Instead of putting the datum directly into the list element, a pointer to another pair is used as the value with the special internal TYPE DEFER, and the real datum is put in the deferred pair. For example the LIST (1 "hello" 3) would look like this:

```
--------------
| LIST | 0  |
| - - - - - |      ------------      ------------      ------------
| 0    | ------>| FIX | -------->|DEFER| -------->| FIX | 0 |
--------------   | - - - - |      | - - - - |      | - - - - |
                 |   1   |        |    -----       |   3   |
                 ------------      ------------ |    ------------
                                               |
                                   ----------- |
                                   |STRING| 5|<-
                                   | - - - - |
                                   |byte-pntr|
                                   -----------
```

Vectors

A vector is a block of contiguous words. More than one pair can point to the block, possibly at different places in the block; this is how sharing occurs among vectors. Pointers that are different arise from REST or GROW/BACK operations. The block is followed by two "dope words", at addresses just larger than the largest address in the block. Dope words have the following format:

```
/                               /
|                               |
|                               |
---------------------------------
|    type      |    grow        |
| - - - - - - - - - - - - - - - |
|    length    |     gc         |
---------------------------------
```

The various fields have the following meanings:

type -- The fourth bit from the left (the "vector bit", 40000 octal) is always one, to distinguish these vector dope words from a TYPE/value pair.

If the high-order bit is zero, then the vector is a UVECTOR, and the remaining bits specify the uniform TYPE of the elements. CHUTYPE just puts a new TYPE code in this field. Each element is limited to a one-word value: clearly PRIMTYPE STRINGs and BYTESes and stack objects can't go in uniform vectors.

If the high-order bit is one and the TYPE bits are zero, then this is a regular VECTOR.
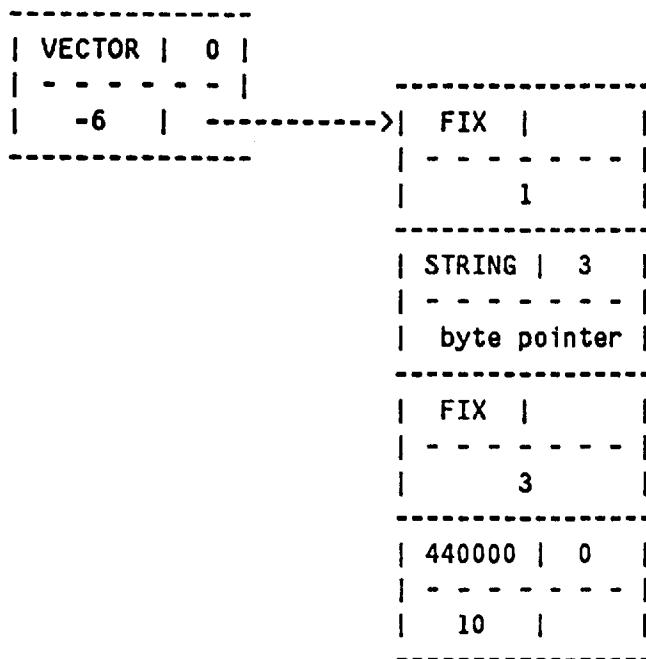
If the high-order bit is one and the TYPE bits are not all zero, then this is either an ATOM, a PROCESS, an ASOC, or a TEMPLATE. The special internal format of these objects will be described a little later in this appendix.

length -- The high-order bit is the mark bit, used by the garbage collector. The rest of this field specifies the number of words in the block, including the dope words. This differs from the length given in pairs pointing to this vector, since such pairs may be the result of REST operations.
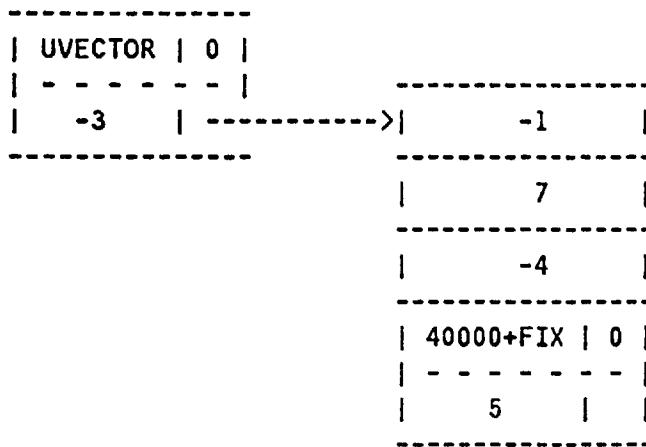
grow -- This is actually two nine-bit fields, specifying either growth or shrinkage at both the high and low ends of the vector. The fields are usually set only when a stack must be grown or shrunk.

gc -- This is used by the garbage collector to specify where this vector is moving during compaction.

Examples (numbers in octal): the VECTOR [1 "bye" 3] looks like:

```
----------------
| VECTOR |  0 |
| - - - - - - |                ----------------
|   -6   | ------------>| FIX  |       |
----------------               | - - - - - - - |
                               |      1        |
                               ----------------
                               | STRING |  3  |
                               | - - - - - - - |
                               | byte pointer  |
                               ----------------
                               | FIX  |        |
                               | - - - - - - - |
                               |      3        |
                               ----------------
                               | 440000 |  0  |
                               | - - - - - - - |
                               |   10   |      |
                               ----------------
```

The UVECTOR ![-1 7 -4!] looks like:

```
----------------
| UVECTOR |  0 |
| - - - - - - |                 ----------------
|   -3   | ------------>|       -1      |
----------------                ----------------
                                |       7       |
                                ----------------
                                |      -4       |
                                ----------------
                                | 40000+FIX | 0 |
                                | - - - - - - - |
                                |      5    |   |
                                ----------------
```

Atoms

Internally, atoms are special vector-like objects. An atom contains a value cell (the first two words of the block, filled in whenever the global or local value of the ATOM is referenced and is not already there), an OBLIST pointer, and a print name (PNAME), in the following format:

```
-----------------------------------
|    type    |    bindid    |
-----------------------------------
|       pointer-to-value      |
-----------------------------------
|       pointer-to-OBLIST      |
-----------------------------------
|         print-name          |
/                             /
/                             /
|(ASCII with NUL padding on end)|
-----------------------------------
|    ATOM    |  valid-type  |
| - - - - - - - - - - - - - - |
|   length   |     gc       |
-----------------------------------
```

If the type field corresponds to TYPE UNBOUND, then the ATOM is locally and globally unbound. (This is different from a pair, where the same TYPE UNBOUND is used to mean unassigned.) If it corresponds to TYPE LOCI (an internal TYPE), then the value cell points either to the global stack, if bindid is zero, or to a local control stack, if bindid is non-zero. The bindid field is used to verify whether the local value pointed to by the value cell is valid in the current environment. The pointer-to-OBLIST is either a counting pointer to an oblist (uvector), a positive offset into the "transfer vector" (for pure ATOMs), or zero, meaning that this ATOM is not on an OBLIST. The valid-type field tells whether or not the ATOM represents a TYPE and if so the code for that TYPE; grow values are never needed for atoms.

Associations

Associations are also special vector-like objects. The first six words of the block contain TYPE/value pairs for the ITEM, INDICATOR and AVALUE of the ASOC. The next word contains forward and backward pointers in the chain for that bucket of the association hash table. The last word contains forward and backward pointers in the chain of all the associations.

```
-----------------------------------
|              ITEM               |
| - - - - - - - - - - - - - - -   |
|              pair               |
-----------------------------------
|            INDICATOR            |
| - - - - - - - - - - - - - - -   |
|              pair               |
-----------------------------------
|             AVALUE              |
| - - - - - - - - - - - - - - -   |
|              pair               |
-----------------------------------
|      bucket-chain pointers      |
-----------------------------------
|   association-chain pointers    |
-----------------------------------
|     ASOC      |        0        |
| - - - - - - - - - - - - - - -   |
|    12 octal   |       gc        |
-----------------------------------
```

Processes

A PROCESS vector looks exactly like a vector of TYPE/value pairs. It is different only in that the garbage collector treats it differently from a normal vector, and it contains extremely volatile information when the PROCESS is RUNNING.

Templates

In a template, the number in the type field (left half of first dope word) identifies to which "storage allocation class" this TEMPLATE belongs, and it is used to find PDP-10 instructions in internal tables (frozen uvectors) for performing LENGTH, NTH, and PUT operations on any object of this TYPE. The programs to build these tables are not part of the interpreter, but the interpreter does know how to use them properly. The compiler can put these instructions directly in compiled programs if a TEMPLATE is never RESTed; otherwise it must let the interpreter discover the appropriate instruction. The value word of a template pair contains, not a counting pointer, but the number of elements that have been RESTed off in the left half and a pointer to the first dope word in the right half.

The Control Stack

Accumulators with symbolic names AB, TB, and TP are all pointers into the RUNNING PROCESS's control stack. AB ("argument base") is a pointer to the arguments to the Subroutine now being run. It is set up by the Subroutine-call mediator, and its old value is always restored after a mediated Subroutine call returns. TB ("temporaries base") points to the frame for the running Subroutine and also serves as a stack base pointer. The TB pointer is really all that is necessary to return from a Subroutine -- given a value to return, for example by ERRET -- since the frame specifies the entire state of the calling routine. TP ("temporaries pointer") is the actual stack pointer and always points to the current top of the control stack.

While we're on the subject of accumulators, we might as well be complete. Each accumulator contains the value word of a pair, the corresponding TYPE words residing in the RUNNING PROCESS vector. When a PROCESS is not RUNNING (or when the garbage collector is running), the accumulator contents are stored in the vector, so that the objects they point to look like elements of the PROCESS and thus are not garbage-collectible.

Accumulators A, B, C, D, E and O are used almost entirely as scratch accumulators, and they are not saved or restored across Subroutine calls. Of course the interrupt machinery always saves these and all other accumulators. A and B are used to return a pair as the value of a Subroutine call. Other than that special feature, they are just like the other scratch accumulators.

M and R are used in running RSUBRs. M is always set up to point to the start of the RSUBR's code, which is actually just a uniform vector of instructions. All jumps and other references to the code use M as an index register. This makes the code location-insensitive, which is necessary because the code uvector will move around. R is set up to point to the vector of objects needed by the RSUBR. This accumulator is necessary because objects in garbage-collected space can move around, but the pointers to them in the reference vector are always at the same place relative to its beginning.

FRM is the internal frame pointer, used in compiled code to keep track of pending Subroutine calls when the control stack is heavily used. P is the internal-stack pointer, used primarily for internal calls in the interpreter.

One of the nicest features of the MDL environment is the uniformity of the calling and returning sequence. All Subroutines -- both built-in F/SUBRs and compiled RSUBR(-ENTRY)s -- are called in exactly the same way and return the same way. Arguments are always passed on the control stack and results always end up in the same accumulators. For efficiency reasons, a lot of internal calls within the interpreter circumvent the calling sequence. However, all calls made by the interpreter when running user programs go through the standard calling sequence.

A Subroutine call is initiated by one of three UUOs (PDP-10 instructions executed by software rather than hardware). MCALL ("MDL call") is used when the number of arguments is known at assemble or compile time, and this number is less than 16. QCALL ("quick call") may be used if, in addition, an RSUBR(-ENTRY) is being called that can be called "quickly" by virtue of its having

special information in its reference vector. ACALL ("accumulator call") is used otherwise. The general method of calling a Subroutine is to PUSH (a PDP-10 instruction) pairs representing the arguments onto the control stack via TP and then either (1) MCALL or QCALL or (2) put the number of arguments into an accumulator and ACALL. Upon return the object returned by the Subroutine will be in accumulators A and B, and the arguments will have been POPped off the control stack.

The call mediator stores the contents of P and TP and the address of the calling instruction in the current frame (pointed to by TB). It also stores MDL's "binding pointer" to the topmost binding in the control stack. (The bindings are linked together through the control stack so that searching through them is more efficient than looking at every object on the stack.) This frame now specifies the entire state of the caller when the call occurred. The mediator then builds a new frame on the control stack and stores a pointer back to the caller's frame (the current contents of TB), a pointer to the Subroutine being called, and the new contents of AB, which is a counting pointer to the arguments and is computed from the information in the MCALL or QCALL instruction or the ACALL accumulator. TB is then set up to point to the new frame, and its left half is incremented by one, making a new unique-id. The mediator then transfers control to the Subroutine.

A control stack frame has seven words as shown:

```
-----------------------------------
|    ENTRY      |  called-addr  |
-----------------------------------
|   unique-id   |   prev frame  |
-----------------------------------
|        argument pointer       |
-----------------------------------
|      saved binding pointer    |
-----------------------------------
|            saved P            |
-----------------------------------
|            saved TP           |
-----------------------------------
|      saved calling address    |
-----------------------------------
```
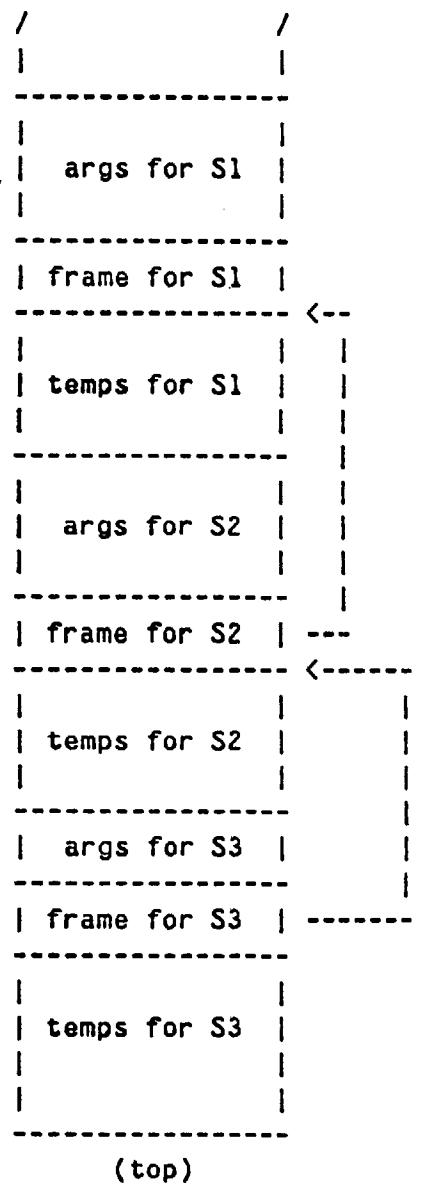
The first three words are set up during the call to the Subroutine. The rest are filled in when this routine calls another Subroutine. The left half of TB is incremented every time a Subroutine call occurs and is used as the unique-id for the frame, stored in frame and tuple pairs as mentioned before. Obviously this id is not strictly unique, since each 256K calls it wraps around to zero. The right half of TB is always left pointing one word past the saved-calling-address word in the frame. TP is also left pointing at that word, since that is the top of the control stack at Subroutine entry. The arguments to the called Subroutine are below the frame on the control stack (at lower storage addresses), and the temporaries for the called Subroutine are above the frame (at higher storage addresses). These arguments and temporaries are just pairs stored on the control stack while needed; they are all that remain of UNSPECIAL values in compiled programs.

The following figure shows what the control stack might look like after several Subroutine calls.

```
        /               /
        |               |
        -----------------
        |               |
      , |   args for S1 |
        |               |
        -----------------
        | frame for S1  |
        ----------------- <--
        |               |  |
        | temps for S1  |  |
        |               |  |
        -----------------  |
        |               |  |
        |  args for S2  |  |
        |               |  |
        -----------------  |
        | frame for S2  | ---
        ----------------- <------
        |               |       |
        | temps for S2  |       |
        |               |       |
        -----------------       |
        |  args for S3  |       |
        -----------------       |
        | frame for S3  | -------
        -----------------
        |               |
        | temps for S3  |
        |               |
        |               |
        -----------------
             (top)
```

The above figure shows the frames all linked together through the control stack (the "execution path"), so that it is easy to return to the caller of a given Subroutine (ERRET or RETRY).

Subroutine exit is accomplished simply by the call mediator, which loads the right half of TB from the previous frame pointer, restores the "binding pointer", P, and TP, and transfers control back to the instruction following the saved calling address.

Appendix 1

Variable Bindings

All local ATOM values are kept on the control stack of the PROCESS to which they are local. As described before, the atom contains a word that points to the value on the control stack. The pointer is actually to a six-word "binding block" on the control stack. Binding blocks have the following format:

```
-----------------------------------
| BIND or UBIND |        prev      |
-----------------------------------
|          pointer to ATOM         |
-----------------------------------
|              value               |
| - - - - - - - - - - - - - - - -  |
|              pair                |
-----------------------------------
|     decl      |    unique-id     |
-----------------------------------
|        previous-binding          |
-----------------------------------
```

where:

  BIND means this is a binding for a SPECIAL ATOM (the only kind used by compiled programs), and UBIND means this is a binding for an UNSPECIAL ATOM -- for SPECIAL checking by the interpreter;

  prev points to the closest previous binding block for any ATOM (the "access path" -- UNWIND objects are also linked in this chain);

  decl points to a DECL associated with this value, for SET(LOC) to check;

  unique-id is used for validation of this block; and

  previous-binding points to the closest previous binding for this ATOM (used in unbinding).

Bindings are generated by an internal subroutine called SPECBIND (name comes from SPECIAL). The caller to SPECBIND PUSHes consecutive six-word blocks onto the control stack via TP before calling SPECBIND. The first word of each block contains the TYPE code for ATOM in its left half and all ones in its right half. SPECBIND uses this bit pattern to identify the binding blocks. SPECBIND's caller also fills in the next three words and leaves the last two words empty. SPECBIND fills in the rest and leaves the "binding pointer" pointing at the topmost binding on the control stack. SPECBIND also stores a pointer to the current binding in the value cell of the atom.

Unbinding is accomplished during Subroutine return. When the previous frame is being restored, the call mediator checks to see if the saved "binding pointer" and the current one are different; if they are, SPECSTORE is called. SPECSTORE runs through the binding blocks, restoring old value pointers in atoms until the "binding pointer" is equal to the one saved in the frame.

Obviously variable binding is more complicated than this, because ATOMs can have both local and global values and even different local values in different PROCESSes. The solution to all of these additional problems lies in the bindid field of the atom. Each PROCESS vector also contains a current bindid. Whenever an ATOM's local value is desired, the RUNNING PROCESS's bindid is checked against that of the atom: if they are the same, the atom points to the current value; if not, the current PROCESS's control stack must be searched to find a binding block for this ATOM. This binding scheme might be called "shallow binding". The searching is facilitated by having all binding blocks linked together. Accessing global variables is accomplished in a similar way, using a VECTOR that is referred to as the "global stack". The global stack has only an ATOM and a value slot for each variable, since global values never get rebound.

EVAL with respect to a different environment causes some additional problems. Whenever this kind of EVAL is done, a brand new bindid is generated, forcing all current local value cells of atoms to appear invalid. Local values must now be obtained by searching the control stack, which is inefficient compared to just pulling them out of the atoms. (The greatest inefficiency occurs when an ATOM's LVAL is never accessed twice in a row in the same environment.) A special block is built on the control stack and linked into the binding-block chain. This block is called a "skip block" or "environment splice", and it diverts the access path to the new environment, causing searches to become relative to this new environment.

Appendix 1

Appendix 2. Predefined Subroutines

The following is a very brief description of all the primitives currently available in MDL. These descriptions are in no way to be considered a definition of the effects or values produced by the primitives. They just try to be as complete and as accurate as is possible in a single-statement description. However, because of the complexity of most primitives, many important defaults and restrictions have been omitted.

A description is given in this format:

NAME
rsubr-decl
English description

The NAME is just the ATOM that is used to refer to the primitive. The *rsubr-decl* is what would go **inside** an "RSUBR DECL" (chapter 14) for the primitive. One special convention is used: an element of a DECL Pattern enclosed in hyphens (for example -STRING-) means that **that element only** is optional. This convention is handy for F/SUBRs that take flexible argument Patterns. Even though all primitives return a value, some English descriptions only mention the side effects produced by a primitive, because these primitives are most often used for this effect rather than the value.

*
"VALUE" <OR FIX FLOAT>
"TUPLE" <TUPLE [REST <OR FIX FLOAT>]>
Arithmetic: multiplication

\+
"VALUE" <OR FIX FLOAT>
"TUPLE" <TUPLE [REST <OR FIX FLOAT>]>
Arithmetic: addition

\-
"VALUE" <OR FIX FLOAT>
"TUPLE" <TUPLE [REST <OR FIX FLOAT>]>
Arithmetic: subtraction

/
"VALUE" <OR FIX FLOAT>
"TUPLE" <TUPLE [REST <OR FIX FLOAT>]>
Arithmetic: division

0?
"VALUE" <OR FALSE 'T>
<OR FIX FLOAT>
Predicate: equality to number zero

1?
"VALUE" <OR FALSE 'T>
<OR FIX FLOAT>
Predicate: equality to number one

1STEP
"VALUE" PROCESS
PROCESS
Cause a process to enter single-step mode

==?
"VALUE" <OR 'T FALSE>
ANY ANY
Predicate: "exact" equality

=?
"VALUE" <OR 'T FALSE>
ANY ANY
Predicate: "structural" equality

ABS
"VALUE" <OR FIX FLOAT>
<OR FIX FLOAT>
Arithmetic: absolute value

ACCESS
"VALUE" CHANNEL
CHANNEL FIX
Randomly access disk file etc.

ACTIVATE-CHARS
"VALUE" STRING
"OPTIONAL" STRING
Set or inspect interrupt characters for console typing (TENEX only)

AGAIN
"VALUE" ANY
"OPTIONAL" ACTIVATION
Restart a given activation block

ALLTYPES
"VALUE" <VECTOR [REST ATOM]>
Return a vector of all type names

AND
"VALUE" <OR FALSE ANY>
"ARGS" LIST
Logical: "and" of truthvalues, evaluated one at a time by the Subroutine

AND?
"VALUE" <OR FALSE ANY>
"TUPLE" TUPLE
Logical: "and" of truthvalues, evaluated at call time

ANDB
"VALUE" WORD
"TUPLE" <TUPLE [REST <PRIMTYPE WORD>]>
Bitwise "and" of words

APPLICABLE?
"VALUE" <OR 'T FALSE>
ANY
Predicate: is object applicable

APPLY
"VALUE" ANY
APPLICABLE "TUPLE" TUPLE
Apply first argument to the rest of them

APPLYTYPE
"VALUE" <OR ATOM APPLICABLE FALSE>
ATOM "OPTIONAL" <OR ATOM APPLICABLE>
Specify or return how a data type is applied

ARGS
"VALUE" TUPLE
<OR FRAME ACTIVATION ENVIRONMENT PROCESS>
Return arguments of a given previous Subroutine call

ASCII
"VALUE" <OR CHARACTER FIX>
<OR FIX CHARACTER>
Return character with given ASCII code or vice versa

ASSIGNED?
"VALUE" <OR 'T FALSE>
ATOM "OPTIONAL" <OR ENVIRONMENT ACTIVATION FRAME PROCESS>
Predicate: is an atom locally assigned

ASSOCIATIONS
"VALUE" <OR ASOC FALSE>
Return the first entry in the association chain

AT
"VALUE" LOCATIVE
STRUCTURED "OPTIONAL" FIX
Return a locative to the Nth element of a structure

ATAN
"VALUE" FLOAT
<OR FIX FLOAT>
Arithmetic: arc tangent

ATOM
"VALUE" ATOM
STRING
Create an atom with a given name

AVALUE
"VALUE" ANY
ASOC
Return the "value" field of an association

BACK
"VALUE" <OR VECTOR UVECTOR STRING TEMPLATE>
<OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR> <PRIMTYPE STRING> <PRIMTYPE TEMPLATE>>
    "OPTIONAL" FIX
Replace some elements removed from a non-list structure by RESTing and change to primitive data
    type

BIND
"VALUE" ANY
"ARGS" <LIST -ATOM- LIST -DECL- ANY>
Execute sequential expressions without providing default activation

BITS
"VALUE" BITS
FIX "OPTIONAL" FIX
Create bit mask for PUTBITS and GETBITS

Appendix 2

BLOAT
"VALUE" FIX
"OPTIONAL" FIX FIX FIX FIX FIX FIX FIX FIX FIX FIX FIX FIX FIX FIX FIX
Allocate extra storage temporarily

BLOAT-STAT
"VALUE" <UVECTOR [27 FIX]>
"OPTIONAL" <UVECTOR [27 ANY]>
Give garbage-collector and storage statistics

BLOCK
"VALUE" <OR OBLIST <LIST [REST <OR OBLIST 'DEFAULT>]>>
<OR OBLIST <LIST [REST <OR OBLIST 'DEFAULT>]>>
Create a current path of oblists for READing

BOUND?
"VALUE" <OR 'T FALSE>
ATOM "OPTIONAL" <OR ENVIRONMENT ACTIVATION FRAME PROCESS>
Predicate: is an atom locally bound

BREAK-SEQ
"VALUE" PROCESS
ANY PROCESS
Modify execution sequence of another process

BUFOUT
"VALUE" CHANNEL
"OPTIONAL" CHANNEL
Write out all internal MDL buffers for an I/O channel

BYTES
"VALUE" BYTES
FIX "TUPLE" <TUPLE [REST <PRIMTYPE WORD>]>
Create a byte-string of explicit arguments

BYTE-SIZE
"VALUE" FIX
BYTES
Return size of bytes in a byte-string

CHANLIST
"VALUE" <LIST [REST CHANNEL]>
Return a list of currently open I/O channels

CHANNEL
"VALUE" CHANNEL
"OPTIONAL" STRING "TUPLE" TUPLE
Create an I/O channel

CHTYPE
"VALUE" ANY
ANY ATOM
Make a new object with a given data type from an old one

CHUTYPE
"VALUE" <PRIMTYPE UVECTOR>
<PRIMTYPE UVECTOR> ATOM
Change the data type of the elements of a uniform vector

CLOSE
"VALUE" CHANNEL
CHANNEL
Close an I/O channel

CLOSURE
"VALUE" CLOSURE
FUNCTION "TUPLE" <TUPLE [REST ATOM]>
Bind the free variables of a FUNCTION to current values

COND
"VALUE" ANY
"ARGS" <LIST LIST [REST LIST]>
Conditional evaluation of expressions

CONS
"VALUE" LIST
ANY LIST
Add an element to the front of a list

COS
"VALUE" FLOAT
<OR FIX FLOAT>
Arithmetic: cosine

CRLF
"VALUE" 'T
"OPTIONAL" CHANNEL
Print a carriage-return and line-feed on an I/O channel

DECL-CHECK
"VALUE" <OR FALSE ANY>
"OPTIONAL" <OR FALSE ANY>
Enable or disable type-declaration checking

DECL?
"VALUE" <OR 'T FALSE>
ANY <OR ATOM <FORM 'QUOTE FORM>>
Predicate: does given object match given type declaration

DEFINE
"VALUE" ATOM
ATOM "ARGS" <LIST -ATOM- LIST -DECL- ANY>
Set the global value of an atom to a FUNCTION

DEFMAC
"VALUE" ATOM
ATOM "ARGS" <LIST -ATOM- LIST -DECL- ANY>
Set the global value of an atom to a MACRO

DEMSIG
"VALUE" <OR 'T FALSE>
STRING
Signal an ITS daemon

DISABLE
"VALUE" IHEADER
IHEADER
Disable an interrupt

DISMISS
"VALUE" ANY
ANY "OPTIONAL" ACTIVATION FIX
Dismiss an interrupt occurrence

DISPLAY
"VALUE" CHANNEL
CHANNEL "OPTIONAL" PICTURE
Display pictures on the E&S (ITS only)

ECHOPAIR
"VALUE" CHANNEL
CHANNEL CHANNEL
Coordinate I/O channels for echoing characters on rubout

EMPTY?
"VALUE" <OR 'T FALSE>
STRUCTURED
Predicate: does a structure have zero elements

ENABLE
"VALUE" IHEADER
IHEADER
Enable an interrupt

ENDBLOCK
"VALUE" <OR OBLIST <LIST [REST <OR OBLIST 'DEFAULT>]>>
Restore path of oblists extant before last call to BLOCK

ENTRY-LOC
"VALUE" FIX
RSUBR-ENTRY
Return the offset of an RSUBR-ENTRY

EQVB
"VALUE" WORD
"TUPLE" <TUPLE [REST <PRIMTYPE WORD>]>
Bitwise "equivalence" of words

ERASE
"VALUE" CHANNEL
CHANNEL "OPTIONAL" PICTURE
Erase a picture from the E&S (ITS only)

ERRET
"VALUE" ANY
"OPTIONAL" ANY FRAME
Continue evaluation from the last ERROR or LISTEN or from a given frame

ERROR
"VALUE" ANY
"TUPLE" TUPLE
Stop and inform user of an error

ERRORS
"VALUE" OBLIST
Return the oblist where error messages are located

EVAL
"VALUE" ANY
ANY "OPTIONAL" <OR ACTIVATION FRAME ENVIRONMENT PROCESS>
Evaluate an expression in a given environment

EVALTYPE
"VALUE" <OR ATOM APPLICABLE FALSE>
ATOM "OPTIONAL" <OR ATOM APPLICABLE>
Specify or return how a data type is evaluated

EVENT
"VALUE" IHEADER
<OR STRING ATOM IHEADER> FIX "OPTIONAL" <OR CHANNEL LOCATIVE>
Set up an interrupt

EXP
"VALUE" FLOAT
<OR FIX FLOAT>
Arithmetic: exponentiation to the base "e"

EXPAND
"VALUE" ANY
ANY
Evaluate argument (only once if a MACRO is involved) in top-level environment

FILE-LENGTH
"VALUE" FIX
CHANNEL
Return the system-provided file length of an input channel

FILECOPY
"VALUE" FIX
"OPTIONAL" CHANNEL CHANNEL
Copy characters from one channel to another until an EOF on the input channel

FIX
"VALUE" FIX
<OR FLOAT FIX>
Arithmetic: return integer value of a number

FLATSIZE
"VALUE" <OR FALSE FIX>
ANY FIX "OPTIONAL" FIX
Return number of characters needed to PRIN1 an object

FLOAD
"VALUE" '"DONE"
"OPTIONAL" -STRING- -STRING- -STRING- -STRING- <OR OBLIST <LIST [REST <OR OBLIST
    'DEFAULT>]>>
Read and evaluate all objects in a file

FLOAT
"VALUE" FLOAT
<OR FIX FLOAT>
Arithmetic: return floating value of a number

FORM
"VALUE" FORM
"TUPLE" TUPLE
Create a form of explicit arguments

FRAME
"VALUE" FRAME
"OPTIONAL" <OR PROCESS FRAME ACTIVATION ENVIRONMENT>
Return a previous Subroutine call

FREE
"VALUE" <PRIMTYPE STORAGE>
<PRIMTYPE STORAGE>
De-allocate non-garbage-collected storage (archaic) (ITS only)

FREE-RUN
"VALUE" <OR PROCESS FALSE>
PROCESS
Cause a process to leave single-step mode

FREEZE
"VALUE" <OR VECTOR UVECTOR STRING>
<OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR> <PRIMTYPE STRING> <PRIMTYPE TUPLE>>
Make copy of argument in non-moving garbage-collected space

FUNCT
"VALUE" ATOM
<OR PROCESS FRAME ACTIVATION ENVIRONMENT>
Return Subroutine name of a given previous Subroutine call

FUNCTION
"VALUE" FUNCTION
"ARGS" <LIST -ATOM- LIST -DECL- ANY>
Create a FUNCTION

```
G=?
"VALUE" <OR 'T FALSE>
<OR FIX FLOAT> <OR FIX FLOAT>
```
Predicate: is first argument greater than or equal to second

```
G?
"VALUE" <OR 'T FALSE>
<OR FIX FLOAT> <OR FIX FLOAT>
```
Predicate: is first argument greater than second

```
GASSIGNED?
"VALUE" <OR 'T FALSE>
ATOM
```
Predicate: is an atom globally assigned

```
GBOUND?
"VALUE" <OR 'T FALSE>
ATOM
```
Predicate: is an atom globally bound

```
GC
"VALUE" FIX
"OPTIONAL" FIX <OR FALSE ANY>
```
Cause a garbage collection and change a garbage-collection parameter

```
GC-DUMP
"VALUE" <OR ANY <UVECTOR <PRIMTYPE WORD>>>
ANY <OR CHANNEL FALSE>
```
Dump an object so that it can be reproduced exactly

```
GC-MON
"VALUE" <OR FALSE ANY>
"OPTIONAL" <OR FALSE ANY>
```
Turn garbage-collection monitoring off or on

```
GC-READ
"VALUE" ANY
CHANNEL "OPTIONAL" ANY
```
Input an object that was previously GC-DUMPed

```
GDECL
"VALUE" ANY
"ARGS" <LIST [REST <LIST [REST ATOM]> <OR ATOM FORM>]>
```
Declare the type/structure of the global value of an atom

GET
"VALUE" ANY
ANY ANY "OPTIONAL" ANY
Return a given property associated with an item

GET-DECL
"VALUE" <OR ATOM FORM FALSE>
LOCD
Get the type declaration for an atom's value

GETBITS
"VALUE" WORD
<OR <PRIMTYPE WORD> <PRIMTYPE STORAGE>> BITS
Return a bit field of a word

GETL
"VALUE" <OR LOCATIVE ANY>
ANY ANY "OPTIONAL" ANY
Return a locative to an association or to an element

GETPL
"VALUE" <OR LOCAS ANY>
ANY ANY "OPTIONAL" ANY
Return a locative to an association

GETPROP
"VALUE" ANY
ANY ANY "OPTIONAL" ANY
a more general version of GET

GLOC
"VALUE" LOCD
ATOM "OPTIONAL" <OR FALSE ANY>
Return a locative to the global-value cell of an atom

GO
"VALUE" ANY
<OR ATOM TAG>
Go to a tag and continue evaluation from there

GROW
"VALUE" <OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR>>
<OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR>> FIX FIX
Extend the bounds of a vector or uniform vector

GUNASSIGN
"VALUE" ATOM
ATOM
Cause an atom to be globally unassigned

GVAL
"VALUE" ANY
ATOM
Return the global value of an atom

HANDLER
"VALUE" HANDLER
IHEADER <OR HANDLER APPLICABLE> "OPTIONAL" PROCESS
Create an interrupt handler

HANG
"VALUE" ANY
"OPTIONAL" ANY
Do nothing, interruptibly, forever

IBYTES
"VALUE" BYTES
FIX FIX "OPTIONAL" ANY
Create a byte-string of implicit arguments

IFORM
"VALUE" FORM
FIX "OPTIONAL" ANY
Create a form of implicit arguments

ILIST
"VALUE" LIST
FIX "OPTIONAL" ANY
Create a list of implicit arguments

IMAGE
"VALUE" FIX
FIX "OPTIONAL" CHANNEL
Send an image-mode character to a channel

IN
"VALUE" ANY
LOCATIVE
Return the object pointed to by a locative

```
INDICATOR
"VALUE" ANY
ASOC
```
Return the "indicator" field of an association

```
INSERT
"VALUE" ATOM
<OR ATOM STRING> OBLIST
```
Add an atom to an oblist

```
INT-LEVEL
"VALUE" FIX
"OPTIONAL" FIX
```
Return and/or set current interrupt level

```
INTERRUPT
"VALUE" <OR 'T FALSE>
<OR IHEADER STRING ATOM> "TUPLE" TUPLE
```
Cause an interrupt to occur

```
INTERRUPTS
"VALUE" OBLIST
```
Return the oblist on which interrupt names are kept

```
IPC-HANDLER
"VALUE" 'T
<OR STRING UVECTOR STORAGE> FIX STRING STRING "OPTIONAL" STRING STRING
```
Default handler for "IPC" (ITS only)

```
IPC-OFF
"VALUE" 'T
```
Stop all listening on the IPC device (ITS only)

```
IPC-ON
"VALUE" 'T
"OPTIONAL" STRING STRING
```
Listen on the IPC device (ITS only)

```
ISTORAGE
"VALUE" STORAGE
FIX "OPTIONAL" ANY
```
Create a non-garbage-collected storage of implicit arguments (archaic)

ISTRING
"VALUE" STRING
FIX "OPTIONAL" ANY
Create a string of implicit arguments

ITEM
"VALUE" ANY
ASOC
Return the "item" field of an association

ITUPLE
"VALUE" TUPLE
FIX "OPTIONAL" ANY
Create a tuple (vector on the control stack) of implicit arguments

IUVECTOR
"VALUE" UVECTOR
FIX "OPTIONAL" ANY
Create a uniform vector of implicit arguments

IVECTOR
"VALUE" VECTOR
FIX "OPTIONAL" ANY
Create a vector of implicit arguments

JNAME
"VALUE" STRING
Return job name of MDL

L=?
"VALUE" <OR 'T FALSE>
<OR FIX FLOAT> <OR FIX FLOAT>
Predicate: is first argument less than or equal to second

L?
"VALUE" <OR 'T FALSE>
<OR FIX FLOAT> <OR FIX FLOAT>
Predicate: is first argument less than second

LEGAL?
"VALUE" <OR 'T FALSE>
ANY
Predicate: is argument (frame etc.) legal

LENGTH
"VALUE" FIX
STRUCTURED
Return the number of elements in a structure

LENGTH?
"VALUE" <OR FIX FALSE>
STRUCTURED FIX
Predicate: is length of structure less than or equal to an integer

LINK
"VALUE" ANY
ANY STRING "OPTIONAL" OBLIST
Create a symbolic link to any object for READing

LIST
"VALUE" LIST
"TUPLE" TUPLE
Create a list of explicit arguments

LISTEN
"VALUE" ANY
"TUPLE" TUPLE
Stop and inform user that you are waiting

LLOC
"VALUE" LOCD
ATOM "OPTIONAL" <OR ENVIRONMENT ACTIVATION FRAME PROCESS>
Return a locative to the local-value cell of an atom

LOAD
"VALUE" '"DONE"
CHANNEL "OPTIONAL" <OR OBLIST <LIST [REST <OR OBLIST 'DEFAULT>]>>
Read and evaluate all objects from an I/O channel

LOCATIVE?
"VALUE" <OR FALSE 'T>
ANY
Predicate: is object a locative

LOG
"VALUE" FLOAT
<OR FIX FLOAT>
Arithmetic: natural logarithm

Appendix 2

LOGOUT
"VALUE" FALSE
Log out of ITS (useful for disowned and daemon jobs)

LOOKUP
"VALUE" <OR ATOM FALSE>
STRING OBLIST
Return an atom found on a given oblist

LPARSE
"VALUE" LIST
"OPTIONAL" STRING FIX <OR OBLIST <LIST [REST <OR OBLIST 'DEFAULT>]>> VECTOR CHARACTER
Return a list of the objects parsed from a string (sections 7.6.6.3, 15.7.2, 17.1.3)

LVAL
"VALUE" ANY
ATOM "OPTIONAL" <OR ENVIRONMENT ACTIVATION FRAME PROCESS>
Return the local value of an atom

MAIN
"VALUE" PROCESS
Return process 1 (the main process)

MANIFEST
"VALUE" 'T
"TUPLE" <TUPLE [REST ATOM]>
Declare the global values of atoms to be constant

MANIFEST?
"VALUE" <OR 'T FALSE>
ATOM
Predicate: is the global value of an atom constant

MAPF
"VALUE" ANY
<OR FALSE APPLICABLE> APPLICABLE "TUPLE" <TUPLE [REST STRUCTURED]>
Map function onto elements of structures

MAPLEAVE
"VALUE" ANY
"OPTIONAL" ANY
Leave the most recent MAPF/R with a value

MAPR
"VALUE" ANY
<OR FALSE APPLICABLE> APPLICABLE "TUPLE" <TUPLE [REST STRUCTURED]>
Map function onto "rests" of structures

MAPRET
"TUPLE" TUPLE
Return a variable number of objects to the current MAPF/R

MAPSTOP
"TUPLE" TUPLE
MAPRET, then stop looping of MAPF/R and cause application

MAX
"VALUE" <OR FIX FLOAT>
"TUPLE" <TUPLE [REST <OR FIX FLOAT>]>
Arithmetic: maximum argument

ME
"VALUE" PROCESS
Return the current process

MEMBER
"VALUE" <OR FALSE STRUCTURED>
ANY STRUCTURED
Predicate: is object =? to some element of a structure

MEMQ
"VALUE" <OR FALSE STRUCTURED>
ANY STRUCTURED
Predicate: is object ==? to some element of a structure

MIN
"VALUE" <OR FIX FLOAT>
"TUPLE" <TUPLE [REST <OR FIX FLOAT>]>
Arithmetic: minimum argument

MOBLIST
"VALUE" OBLIST
ATOM "OPTIONAL" FIX
Create or get an oblist

MOD
"VALUE" FIX
FIX FIX
Arithmetic: numerical modulus or remainder

MONAD?
"VALUE" <OR 'T FALSE>
ANY
Predicate: is object unstructured or empty structure

N==?
"VALUE" <OR 'T FALSE>
ANY ANY
Predicate: NOT "exact" equality

N=?
"VALUE" <OR 'T FALSE>
ANY ANY
Predicate: NOT "structural" equality

NETACC
"VALUE" <OR FALSE CHANNEL>
CHANNEL
Accept a network connection

NETS
"VALUE" CHANNEL
CHANNEL
Force system network buffer to be sent

NETSTATE
"VALUE" <UVECTOR FIX FIX FIX>
CHANNEL
Return a uniform vector of state info for a network channel

NEWTYPE
"VALUE" ATOM
ATOM ATOM "OPTIONAL" <OR ATOM <FORM 'QUOTE FORM>>
Define a new data type

NEXT
"VALUE" <OR FALSE ASOC>
ASOC
Return the next entry in the association chain

```
NEXTCHR
"VALUE" <OR CHARACTER FIX>
"OPTIONAL" CHANNEL ANY
```
Return the next input character from an I/O channel

```
NOT
"VALUE" <OR 'T FALSE>
<OR FALSE ANY>
```
Logical: "not" of a truthvalue

```
NTH
"VALUE" ANY
STRUCTURED "OPTIONAL" FIX
```
Return the Nth element of a structure

```
OBLIST?
"VALUE" <OR OBLIST FALSE>
ATOM
```
Predicate: return atom's oblist or false if none

```
OFF
"VALUE" <OR FALSE HANDLER IHEADER>
<OR HANDLER IHEADER STRING ATOM> "OPTIONAL" <OR CHANNEL LOCATIVE>
```
Remove an interrupt handler or destroy an interrupt

```
ON
"VALUE" HANDLER
<OR STRING ATOM> APPLICABLE FIX "OPTIONAL" <OR FIX PROCESS> <OR CHANNEL LOCATIVE>
```
Turn on an interrupt and create an interrupt handler

```
OPEN
"VALUE" <OR CHANNEL <FALSE STRING STRING FIX>>
"OPTIONAL" STRING "TUPLE" TUPLE
```
Create and open an I/O channel

```
OPEN-NR
"VALUE" <OR CHANNEL <FALSE STRING STRING FIX>>
"OPTIONAL" STRING "TUPLE" TUPLE
```
Create and open an I/O channel without changing file's reference date

```
OR
"VALUE" <OR FALSE ANY>
"ARGS" LIST
```
Logical: inclusive "or" of truthvalues, evaluated one at a time by the Subroutine

Appendix 2

OR?
"VALUE" <OR FALSE ANY>
"TUPLE" TUPLE
Logical: inclusive "or" of truthvalues, evaluated at call time

ORB
"VALUE" WORD
"TUPLE" <TUPLE [REST <PRIMTYPE WORD>]>
Bitwise inclusive "or" of words

OVERFLOW
"VALUE" <OR 'T '#FALSE ()>
"OPTIONAL" <OR ANY FALSE>
Arithmetic: enable or disable overflow error

PARSE
"VALUE" ANY
"OPTIONAL" STRING FIX <OR OBLIST <LIST [REST <OR OBLIST 'DEFAULT>]>> VECTOR CHARACTER
Parse a string into an object (sections 7.6.6.2, 15.7.2, 17.1.3)

PCODE
"VALUE" PCODE
STRING FIX
Create pointer to pure RSUBR code

PNAME
"VALUE" STRING
ATOM
Return the print name of an atom as a distinct copy

PRIMTYPE
"VALUE" ATOM
ANY
Return the primitive data type of an object

PRIMTYPE-C
"VALUE" PRIMTYPE-C
ATOM
Get a "storage allocation code" for a TYPE

PRIN1
"VALUE" ANY
ANY "OPTIONAL" CHANNEL
Print an object on an I/O channel without format

Appendix 2

PRINC
"VALUE" ANY
ANY "OPTIONAL" CHANNEL
Print an object on an I/O channel without format or indicators

PRINT
"VALUE" ANY
ANY "OPTIONAL" CHANNEL
Print an object on an I/O channel

PRINTB
"VALUE" <OR UVECTOR STORAGE>
<<OR UVECTOR STORAGE> [REST <PRIMTYPE WORD>]> CHANNEL
Write binary information on an I/O channel

PRINTSTRING
"VALUE" FIX
STRING "OPTIONAL" CHANNEL FIX
Write contents of string on an I/O channel

PRINTTYPE
"VALUE" <OR ATOM APPLICABLE FALSE>
ATOM "OPTIONAL" <OR ATOM APPLICABLE>
Specify or return how a data type is printed

PROCESS
"VALUE" PROCESS
APPLICABLE
Create a new process with startup function

PROG
"VALUE" ANY
"ARGS" <LIST -ATOM- LIST -DECL- ANY>
Execute sequential expressions

PURIFY
"VALUE" ANY
"TUPLE" TUPLE
Purify objects everywhere

PUT
"VALUE" ANY
ANY ANY "OPTIONAL" ANY
Associate a property with an item

PUT-DECL
"VALUE" LOCD
LOCD <OR ATOM <FORM 'QUOTE FORM>>
Put the type declaration for an atom's value

PUTBITS
"VALUE" <PRIMTYPE WORD>
<PRIMTYPE WORD> BITS "OPTIONAL" <PRIMTYPE WORD>
Set a masked field in a word

PUTPROP
"VALUE" ANY
ANY ANY "OPTIONAL" ANY
a more general version of PUT

PUTREST
"VALUE" <PRIMTYPE LIST>
<PRIMTYPE LIST> <PRIMTYPE LIST>
Replace the rest of a list

QUIT
"VALUE" '#FALSE ()
Exit from MDL gracefully

QUITTER
"VALUE" CHARACTER
CHARACTER CHANNEL
interrupt handler for ^G and ^S quit features

QUOTE
"VALUE" ANY
"ARGS" LIST
Return first argument unevaluated

RANDOM
"VALUE" FIX
"OPTIONAL" FIX FIX
Arithmetic: generate a uniform pseudo-random integer

READ
"VALUE" ANY
"OPTIONAL" CHANNEL ANY <OR OBLIST <LIST [REST <OR OBLIST 'DEFAULT>]>> VECTOR
Read one object from an I/O channel (sections 11.1.1.1, 11.3, 15.7.1, 17.1.3)

READB
"VALUE" FIX
<<OR UVECTOR STORAGE> [REST <PRIMTYPE WORD>]> CHANNEL "OPTIONAL" ANY
Read binary information from an I/O channel

READCHR
"VALUE" <OR CHARACTER FIX>
"OPTIONAL" CHANNEL ANY
Read one character from an I/O channel

READSTRING
"VALUE" FIX
STRING "OPTIONAL" CHANNEL <OR FIX STRING> ANY
Read into a string from an I/O channel

REALTIMER
"VALUE" <OR FIX FLOAT>
<OR FIX FLOAT>
Set interval for real-time interrupts

REMOVE
"VALUE" <OR ATOM FALSE>
<OR ATOM STRING> "OPTIONAL" OBLIST
Remove an atom from an oblist

RENAME
"VALUE" <OR 'T <FALSE STRING FIX>>
"TUPLE" <TUPLE <OR STRING CHANNEL>>
Rename or delete a disk file

REP
"VALUE" ANY
default for Read-Eval-Print loop

REPEAT
"VALUE" ANY
"ARGS" <LIST -ATOM- LIST -DECL- ANY>
Execute sequential expressions repeatedly

RESET
"VALUE" <OR CHANNEL <FALSE STRING STRING FIX>>
CHANNEL
Reopen an I/O channel at its beginning

REST
"VALUE" STRUCTURED
STRUCTURED "OPTIONAL" FIX
Remove the first N elements from a structure and change to primitive data type

RESTORE
"VALUE" '"RESTORED"
"OPTIONAL" STRING STRING STRING STRING
Restore MDL's state from a file

RESUME
"VALUE" ANY
ANY "OPTIONAL" PROCESS
Transfer execution to another process

RESUMER
"VALUE" <OR PROCESS FALSE>
"OPTIONAL" PROCESS
Return the process that most recently resumed this one

RETRY
"OPTIONAL" FRAME
Retry a previous Subroutine call from the error level

RETURN
"VALUE" ANY
"OPTIONAL" ANY ACTIVATION
Leave an activation block with a value

RGLOC
"VALUE" LOCR
ATOM "OPTIONAL" <OR FALSE ANY>
Return a locative to the global-value cell of an atom for pure-program use

ROOT
"VALUE" OBLIST
Return the oblist containing primitives

RSUBR
"VALUE" RSUBR
<VECTOR <OR CODE PCODE> ATOM DECL [REST ANY]>
Create a relocatable subroutine

Appendix 2

RSUBR-ENTRY
"VALUE" RSUBR-ENTRY
<VECTOR <OR ATOM RSUBR> ATOM DECL> FIX
Add entry point to RSUBR

RSUBR-LINK
"VALUE" <OR FALSE ANY>
"OPTIONAL" <OR FALSE ANY>
Enable or disable the automatic linking feature

RUNINT
"VALUE" ANY
APPLICABLE "TUPLE" TUPLE
Apply interrupt handler (for internal use only)

RUNTIMER
"VALUE" <OR FIX FLOAT>
<OR FIX FLOAT>
Set interval for run-time interrupt

SAVE
"VALUE" '"SAVED"
"OPTIONAL" -STRING- -STRING- -STRING- -STRING- <OR FALSE ANY>
Write the entire state of MDL to a file

SEND
"VALUE" <OR 'T FALSE>
STRING STRING <OR STRING STORAGE <UVECTOR [REST <PRIMTYPE WORD>]>> "OPTIONAL" FIX
    STRING STRING
Send an IPC message (ITS only)

SEND-WAIT
"VALUE" 'T
STRING STRING <OR STRING STORAGE <UVECTOR [REST <PRIMTYPE WORD>]>> "OPTIONAL" FIX
    STRING STRING
Send an IPC message and wait for it to be received (ITS only)

SET
"VALUE" ANY
ATOM ANY "OPTIONAL" <OR ENVIRONMENT ACTIVATION FRAME PROCESS>
Change the local value of an atom

SETG
"VALUE" ANY
ATOM ANY
Change the global value of an atom

SETLOC
"VALUE" ANY
LOCATIVE ANY
Change the contents pointed to by a locative

SIN
"VALUE" FLOAT
<OR FIX FLOAT>
Arithmetic: sine

SLEEP
"VALUE" ANY
<OR FIX FLOAT> "OPTIONAL" ANY
Do nothing, interruptibly, the number of seconds indicated

SNAME
"VALUE" STRING
"OPTIONAL" STRING
Set or return the default directory name for new I/O channels

SORT
"VALUE" <OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR> <PRIMTYPE TUPLE>>
<OR FALSE APPLICABLE> <OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR> <PRIMTYPE TUPLE>>
    "OPTIONAL" FIX FIX "TUPLE" <TUPLE [REST <OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR>
    <PRIMTYPE TUPLE>> FIX]>
Sort elements of a structure

SPECIAL-CHECK
"VALUE" <OR 'T FALSE>
"OPTIONAL" <OR ANY FALSE>
Turn interpreter special-checking on or off

SPECIAL-MODE
"VALUE" <OR 'SPECIAL 'UNSPECIAL>
"OPTIONAL" <OR 'SPECIAL 'UNSPECIAL>
Set default speciality

SPNAME
"VALUE" STRING
ATOM
Return the print name of an atom by sharing it

SQRT
"VALUE" FLOAT
<OR FIX FLOAT>
Arithmetic: square root

SQUOTA
"VALUE" <OR FIX FALSE>
<PRIMTYPE WORD>
Get the address of an internal interpreter symbol (for internal use only)

STACKFORM
"VALUE" ANY
"ARGS" <LIST APPLICABLE ANY ANY>
Apply a function to stacked arguments

STATE
"VALUE" ATOM
PROCESS
Return a process's current state

STORE
"VALUE" STORAGE
<<OR <PRIMTYPE UVECTOR> <PRIMTYPE STORAGE>> [REST <PRIMTYPE WORD>]>
Copy into new non-garbage-collected storage (archaic) (ITS only)

STRCOMP
"VALUE" FIX
<OR ATOM STRING> <OR ATOM STRING>
Compare two character strings or two print names

STRING
"VALUE" STRING
"TUPLE" <TUPLE [REST <OR STRING CHARACTER>]>
Create a string of explicit arguments

STRUCTURED?
"VALUE" <OR FALSE 'T>
ANY
Predicate: is argument structured

SUBSTITUTE
"VALUE" ANY
ANY ANY
Substitute one object for another everywhere

SUBSTRUC
"VALUE" <OR LIST VECTOR UVECTOR STRING BYTES>
<OR <PRIMTYPE LIST> <PRIMTYPE VECTOR> <PRIMTYPE TUPLE> <PRIMTYPE UVECTOR> <PRIMTYPE
    STRING> <PRIMTYPE BYTES>> "OPTIONAL" FIX FIX <OR LIST VECTOR UVECTOR STRING BYTES>
Copy part of a structure into another

SUICIDE
"VALUE" ANY
ANY "OPTIONAL" PROCESS
Cause the current process to die and resume another

TAG
"VALUE" TAG
ATOM
Create a tag in an activation block

TERPRI
"VALUE" '#FALSE ()
"OPTIONAL" CHANNEL
Print a carriage-return and line-feed on an I/O channel

TIME
"VALUE" FLOAT
"TUPLE" TUPLE
Return the elapsed execution time in seconds

TOP
"VALUE" <OR VECTOR UVECTOR STRING TEMPLATE>
<OR <PRIMTYPE VECTOR> <PRIMTYPE UVECTOR> <PRIMTYPE STRING> <PRIMTYPE TEMPLATE>>
Replace all elements removed from a non-list structure by RESTing and change to primitive data
    type

TTYECHO
"VALUE" CHANNEL
CHANNEL <OR FALSE ANY>
Turn echoing of characters typed on a console on or off

TUPLE
"VALUE" TUPLE
"TUPLE" TUPLE
Create a tuple (vector on the control stack) of explicit arguments

TYI
"VALUE" CHARACTER
"OPTIONAL" CHANNEL
Input a character from a console immediately

TYPE
"VALUE" ATOM
ANY
Return the data type of an object

TYPE-C
"VALUE" TYPE-C
ATOM "OPTIONAL" ATOM
Get a data-type code for pure-program use

TYPE-W
"VALUE" TYPE-W
ATOM "OPTIONAL" ATOM <PRIMTYPE WORD>
Get a data-type word for pure-program use

TYPE?
"VALUE" <OR FALSE ATOM>
ANY "TUPLE" <TUPLE ATOM [REST ATOM]>
Predicate: is object one of specified data types

TYPEPRIM
"VALUE" ATOM
ATOM
Return a data type's primitive type

UNAME
"VALUE" STRING
Return user name of MDL

UNASSIGN
"VALUE" ATOM
ATOM "OPTIONAL" <OR ENVIRONMENT ACTIVATION FRAME PROCESS>
Cause an atom to be locally unassigned

UNMANIFEST
"VALUE" 'T
"TUPLE" <TUPLE [REST ATOM]>
Declare the global values of atoms not to be constants

UNPARSE
"VALUE" STRING
ANY "OPTIONAL" FIX
Create a string representation of an object

UNWIND
"VALUE" ANY
"ARGS" <LIST ANY ANY>
Specify cleaning-up during non-local return

UTYPE
"VALUE" ATOM
<PRIMTYPE UVECTOR>
Return the data type of all elements of a uniform vector

UVECTOR
"VALUE" UVECTOR
"TUPLE" TUPLE
Create a uniform vector of explicit arguments

VALID-TYPE?
"VALUE" <OR TYPE-C FALSE>
ATOM
Predicate: is an atom the name of a type

VALRET
"VALUE" FALSE
STRING
Pass a string to superior job using .VALUE UUO

VALUE
"VALUE" ANY
ATOM "OPTIONAL" <OR ENVIRONMENT ACTIVATION FRAME PROCESS>
Return the local or else the global value of an atom

VECTOR
"VALUE" VECTOR
"TUPLE" TUPLE
Create a vector of explicit arguments

XORB
"VALUE" WORD
"TUPLE" <TUPLE [REST <PRIMTYPE WORD>]>
Bitwise exclusive "or" of words

Appendix 3. Predefined Types

On these two pages is a table showing each of MDL's predefined TYPEs, its primitive type if different, and various flags: S for STRUCTURED, E for EVALTYPE not QUOTE, and A for APPLICABLE.

X means that an object of that TYPE cannot be CHTYPEd to and hence cannot be READ in (if attempted, CAN'T-CHTYPE-INTO ERROR is usual).

B means that an object of that TYPE cannot be READ in (if attempted, STORAGE-TYPES-DIFFER ERROR is usual), that instead it is built by the interpreter or CHTYPEd to by a program, and that its PRINTed representation makes it look as though its TYPEPRIM were different.

% means that an object of that TYPE is PRINTed using % notation and can be READ in only that way.

TYPE	TYPEPRIM	S	E	A	comments
ACTIVATION	FRAME			X	
ASOC				B	sic: only one S
ATOM					
BITS	WORD				
BYTES		S			
CHANNEL	VECTOR	S		X	
CHARACTER	WORD				
CLOSURE	LIST	S		A	
CODE	UVECTOR	S			
DECL	LIST	S			
DISMISS	ATOM				can be returned by interrupt handler
ENVIRONMENT	FRAME			B	
FALSE	LIST	S			
FIX	WORD			A	
FLOAT	WORD				
FORM	LIST	S	E		
FRAME				B	
FSUBR	WORD			A	X
FUNCTION	LIST	S		A	
HANDLER	VECTOR	S		X	
IHEADER	VECTOR	S		X	"interrupt header"
ILLEGAL	WORD			X	Garbage collector may put this on non-LEGAL? object.
INTERNAL	WORD			X	should not be seen by programs
LINK	ATOM			X	for console shorthand
LIST		S	E		
LOCA				B	locative to TUPLE

LOCAS				B		locative to ASOC
LOCB				B		locative to BYTES
LOCD				%		locative to G/LVAL
LOCL				B		locative to LIST
LOCR				%		locative to GVAL in pure program
LOCS				B		locative to STRING
LOCT				B		locative to TEMPLATE
LOCU				B		locative to UVECTOR
LOCV				B		locative to VECTOR
LOSE	WORD					a place holder
MACRO	LIST	S	A			
OBLIST	UVECTOR	S		X		
PCODE	WORD			%		"pure code"
PICTURE	STORAGE	S				
PRIMTYPE-C	WORD			%		"primtype code"
PROCESS				B		
QUICK-ENTRY	VECTOR	S	A	%		an RSUBR-ENTRY that has been QCALLed and RSUBR-LINKed
QUICK-RSUBR	VECTOR	S	A	%/B		an RSUBR that has been QCALLed and RSUBR-LINKed
READA	FRAME			X		in eof slot during recursive READ via READ-TABLE
RSUBR	VECTOR	S	A	%/B		if code vector is pure/impure, respectively
RSUBR-ENTRY	VECTOR	S	A	%		
SEGMENT	LIST	S	E			
SPLICE	LIST	S				for returning many things via READ-TABLE
STORAGE		S				If possible, use FREEZE SUBR instead.
STRING		S				
SUBR	WORD		A	X		
TAG	VECTOR	S		X		for non-local GOs
TEMPLATE		S		B		The interpreter itself can't build one. See ref 1.
TIME	WORD					used internally to identify FRAMEs
TUPLE		S		B		vector on the control stack
TYPE-C	WORD			%		"type code"
TYPE-W	WORD			%		"type word"
UNBOUND	WORD			X		value of unassigned but bound ATOM, as seen by locatives
UVECTOR		S	E			"uniform vector"
VECTOR		S	E			
WORD						

Appendix 4. Error Messages

This is a list of all ATOMs initially in the ERRORS OBLIST, in the left-hand column, and appropriate examples or elucidations, where necessary, in the right-hand column.

ACCESS-FAILURE	ACCESS, RESTORE (TENEX only)
ALREADY-DEFINED-ERRET-NON-FALSE-TO-REDEFINE	
APPLY-OR-STACKFORM-OF-FSUBR	First argument to APPLY, STACKFORM, MAPF/R doesn't EVAL all its arguments.
ARG-WRONG-TYPE	
ARGUMENT-OUT-OF-RANGE	<ASCII 999>$ Second argument to NTH or REST too big or small.
ATOM-ALREADY-THERE	<INSERT "T" <ROOT>>$ <LINK 'T "T" <ROOT>>$
ATOM-NOT-TYPE-NAME-OR-SPECIAL-SYMBOL	DECL problem
ATOM-ON-DIFFERENT-OBLIST	INSERT, LINK, REMOVE
ATTEMPT-TO-BREAK-OWN-SEQUENCE	<BREAK-SEQ T <ME>>$
ATTEMPT-TO-CHANGE-MANIFEST-VARIABLE	
ATTEMPT-TO-CLOSE-TTY-CHANNEL	<CLOSE ,INCHAN>$
ATTEMPT-TO-DEFER-UNDEFERABLE-INTERRUPT	"Undeferable" interrupt (e.g. "ERROR") while INT-LEVEL is too high to handle it
ATTEMPT-TO-GROW-VECTOR-TOO-MUCH	GROW argument greater than <* 16 1024>
ATTEMPT-TO-MUNG-ATOMS-PNAME	<PUT <SPNAME T> 1 !\T>$
ATTEMPT-TO-MUNG-PURE-STRUCTURE	attempt to write into pure page
ATTEMPT-TO-SUICIDE-TO-SELF	<SUICIDE <ME>>$
BAD-ARGUMENT-LIST	<GDECL ("HI") STRING>$
BAD-ASCII-CHARACTER	A character with wrong byte size or ASCII code more than 177 octal has been read (how?).
BAD-BYTES-DECL	
BAD-CHANNEL	
BAD-CLAUSE	Argument to COND is non-LIST or empty LIST.
BAD-DECLARATION-LIST	DECL in bad form
BAD-DEFAULT-OBLIST-SPECIFICATION	bad use of DEFAULT in LIST of OBLISTs
BAD-E&S-TABLE	bad character definitions for PICTUREs (ITS only)
BAD-ENTRY-BLOCK	RSUBR-ENTRY does not point to good RSUBR.
BAD-ENVIRONMENT	
BAD-FIXUPS	
BAD-FUNARG	CLOSURE in bad form

BAD-GC-READ-FILE	
BAD-INPUT-BUFFER	(for a CHANNEL)
BAD-LINK	<GUNASSIGN <CHTYPE *link* ATOM>>
BAD-MACRO-TABLE	.READ-TABLE or .PARSE-TABLE is not a vector.
BAD-OBLIST-OR-LIST-THEREOF	Alleged oblist path is not of TYPE OBLIST or LIST.
BAD-PARSE-STRING	non-STRING argument to PARSE
BAD-PNAME	attempt to output ATOM with missing or zero-length PNAME
BAD-PRIMTYPEC	
BAD-TEMPLATE-DATA	
BAD-TYPE-CODE	
BAD-TYPE-NAME	ATOM purports to be a TYPE but isn't.
BAD-TYPE-SPECIFICATION	DECL problem
BAD-USE-OF-BYTE-STRING	#3$
BAD-USE-OF-MACRO	
BAD-USE-OF-SQUIGGLY-BRACKETS	{}$
BAD-VECTOR	Bad argument to RSUBR-ENTRY
BYTE-SIZE-BAD	"NET" CHANNEL
CAN'T-PARSE	<PARSE "">$ <PARSE ")">$
CANT-CHTYPE-INTO	<CHTYPE 1 SUBR>$
CANT-FIND-TEMPLATE	attempt to GC-READ a structure containing a TEMPLATE whose TYPE does not exist
CANT-OPEN-OUTPUT-FILE	SAVE
CANT-RETRY-ENTRY-GONE	attempt to RETRY a call to an RSUBR-ENTRY whose RSUBR cannot be found
CANT-SUBSTITUTE-WITH-STRING-OR-TUPLE-AND-OTHER	<SUBSTITUTE "T" T>$
CHANNEL-CLOSED	<READ <CLOSE *channel*>>$
CONTROL-G?	^G
COUNT-GREATER-THAN-STRING-SIZE	<PRINTSTRING "" ,OUTCHAN 1>$
DANGEROUS-INTERRUPT-NOT-HANDLED	(See section 21.10.13.) (ITS only)
DATA-CAN'T-GO-IN-STORAGE	FREEZE ISTORAGE
DATA-CANT-BE-STORED	STORE (ITS only)
DATA-CANT-GO-IN-UNIFORM-VECTOR	!["STRING"]$![<FRAME>]$
DECL-ELEMENT-NOT-FORM-OR-ATOM	
DECL-VIOLATION	
DEVICE-OR-SNAME-DIFFERS	RENAME
DISPLAY-FULL	DISPLAY (ITS only)
ELEMENT-TYPE-NOT-ATOM-FORM-OR-VECTOR	DECL problem
EMPTY-FORM-IN-DECL	
EMPTY-OR/PRIMTYPE-FORM	<OR> or <PRIMTYPE> in DECL
EMPTY-STRING	<READSTRING "">$
END-OF-FILE	
ERRET-TYPE-NAME-DESIRED	

Appendix 4

ERROR-IN-COMPILED-CODE
FILE-NOT-FOUND   RESTORE
FILE-SYSTEM-ERROR
FIRST-ARG-WRONG-TYPE
FIRST-ELEMENT-OF-VECTOR-NOT-CODE   RSUBR in bad form.
FIRST-VECTOR-ELEMENT-NOT-REST-OR-A-FIX   #DECL ((X) <LIST [FOO]>)
FRAME-NO-LONGER-EXISTS   (unused)
HANDLER-ALREADY-IN-USE
HAS-EMPTY-BODY   <#FUNCTION ((X)) 1>$
ILLEGAL
ILLEGAL-ARGUMENT-BLOCK   attempt to PRINT a TUPLE that no longer
                                                  exists

ILLEGAL-FRAME
ILLEGAL-LOCATIVE
ILLEGAL-SEGMENT   Third and later arguments to MAPF/R
                                                  not STRUCTURED.
ILLEGAL-TENEX-FILE-NAME   (TENEX only)
INT-DEVICE-WRONG-TYPE-EVALUATION-RESULT   function for "INT" input CHANNEL
                                                  returned non-CHARACTER.
INTERNAL-BACK-OR-TOP-OF-A-LIST   in compiled code
INTERNAL-INTERRUPT   (unused)
ITS-CHANNELS-EXHAUSTED   Interpreter couldn't open an ITS channel.
MEANINGLESS-PARAMETER-DECLARATION   bad object in argument LIST of Function
MESSAGE-TOO-BIG   IPC (ITS only)
MUDDLE-VERSIONS-DIFFER   RESTORE
NEGATIVE-ARGUMENT
NIL-LIST-OF-OBLISTS   <SET OBLIST '()> T$
NO-ITS-CHANNELS-FREE   IPC-ON (ITS only)
NO-MORE-PAGES   for pure-code mapping
NO-PROCESS-TO-RESUME   <OR <RESUMER> <RESUME>>$
NO-STORAGE   No free storage available for GROW.
NO-SUCH-DEVICE?   (TENEX only)
NON-6-BIT-CHARACTER-IN-FILE-NAME
NON-APPLICABLE-REP   <VALUE REP> not APPLICABLE
NON-APPLICABLE-TYPE
NON-ATOMIC-ARGUMENT
NON-ATOMIC-OBLIST-NAME   T!-3$
NON-DSK-DEVICE   (unused)
NON-EVALUATEABLE-TYPE   (unused)
NON-EXISTENT-TAG   (unused)
NON-STRUCTURED-ARG-TO-INTERNAL-PUT-REST-NTH-TOP-OR-BACK in compiled code
NON-TYPE-FOR-PRIMTYPE-ARG   <PRIMTYPE not-type> in DECL
NOT-A-TTY-TYPE-CHANNEL
NOT-HANDLED   First argument to OFF not ONed.

NOT-IN-ARG-LIST	TUPLE or ITUPLE called outside argument LIST.
NOT-IN-MAP-FUNCTION	MAPRET, MAPLEAVE, MAPSTOP not within MAPF/R
NOT-IN-PROG	<RETURN>$ <AGAIN>$
NTH-BY-A-NEGATIVE-NUMBER	in compiled code
NTH-REST-PUT-OUT-OF-RANGE	in compiled code
NULL-STRING	zero-length STRING
NUMBER-OUT-OF-RANGE	2E38$
ON-AN-OBLIST-ALREADY	<INSERT T <ROOT>>$
OUT-OF-BOUNDS	<1 '()>$ BLOAT argument too large
OVERFLOW	</ 1 0>$ <* 1E30 1E30>$
PDL-OVERFLOW-BUFFER-EXHAUSTED	Stack overflow while trying to expand stack: use RETRY.
PICTURE-NOT-FOUND	ERASE (ITS only)
PROCESS-NOT-RESUMABLE	use of another PROCESS's FRAME, etc.
PROCESS-NOT-RUNABLE-OR-RESUMABLE	
PURE-LOAD-FAILURE	Pure-code file disappeared.
READER-SYNTAX-ERROR-ERRET-ANYTHING-TO-GO-ON	
RSUBR-ENTRY-UNLINKED	RSUBR-ENTRY whose RSUBR cannot be found
RSUBR-IN-BAD-FORMAT	
RSUBR-LACKS-FIXUPS	KEEP-FIXUPS should have been true when RSUBR was input.
SECOND-ARG-WRONG-TYPE	
STORAGE-TYPES-DIFFER	<CHTYPE 1 LIST>$ <CHUTYPE '![1] LIST>$
STRUCTURE-CONTAINS-UNDUMPABLE-TYPE	<GC-DUMP <ME> <>>$
SUBSTITUTE-TYPE-FOR-TYPE	<SUBSTITUTE SUBR FSUBR>$
TEMPLATE-TYPE-NAME-NOT-OF-TYPE-TEMPLATE	attempt to GC-READ a structure containing a TEMPLATE whose TYPE is defined but is not a TEMPLATE
TEMPLATE-TYPE-VIOLATION	
THIRD-ARG-WRONG-TYPE	
TOO-FEW-ARGUMENTS-SUPPLIED	
TOO-MANY-ARGS-TO-SPECIAL-UNSPECIAL-DECL	<SPECIAL any ...>
TOO-MANY-ARGUMENTS-SUPPLIED	
TOP-LEVEL-FRAME	<ERRET> <FRAME <FRAME <FRAME>>>$
TYPE-ALREADY-EXISTS	NEWTYPE
TYPE-MISMATCH	attempt to make a value violate its DECL
TYPE-UNDEFINED	
TYPES-DIFFER-IN-STORAGE-OBJECT	ISTORAGE
TYPES-DIFFER-IN-UNIFORM-VECTOR	![T <>]$
UNASSIGNED-VARIABLE	
UNATTACHED-PATH-NAME-SEPARATOR	!-$

UNBOUND-VARIABLE
UNMATCHED   ENDBLOCK with no matching BLOCK
UVECTOR-PUT-TYPE-VIOLATION   PUT, SETLOC, SUBSTRUC in compiled
                                         code
VECTOR-LESS-THAN-2-ELEMENTS   #DECL ((X) <LIST [REST]>)
WRONG-DIRECTION-CHANNEL   <OPEN "MYFILE">$ (Mode missing or
                                         misspelt.)
WRONG-NUMBER-OF-ARGUMENTS

Appendix 5. Initial Settings

The various switches and useful variables in MDL are initially set up with the following values:

```
<ACTIVATE-CHARS <STRING <ASCII 7> <ASCII 19> <ASCII 15>>>
                                             ;"TENEX only"
<DECL-CHECK T>
<UNASSIGN <GUNASSIGN DEV>>
<GC-MON <>>
<SET INCHAN <SETG INCHAN <OPEN "READ" "TTY:">>>
<UNASSIGN KEEP-FIXUPS>
<UNASSIGN <GUNASSIGN NM1>>
<UNASSIGN <GUNASSIGN NM2>>
<SET OBLIST <SETG OBLIST (<MOBLIST INITIAL 151> <ROOT>)>>
<SET OUTCHAN <SETG OUTCHAN <OPEN "PRINT" "TTY:">>>
<OVERFLOW T>
<UNASSIGN REDEFINE>
<RSUBR-LINK T>
<SETG <UNASSIGN SNM> "working-directory">
<SPECIAL-CHECK <>>
<SPECIAL-MODE UNSPECIAL>
<SET THIS-PROCESS <SETG THIS-PROCESS <MAIN>>>
<ON "CHAR" ,QUITTER 8 0 ,INCHAN>
<ON "IPC" ,IPC-HANDLER 1>                    ;"ITS only"
```

References

Note: the form SYS.nn.mm refers to an internal PTD document, available from Programming Technology Division, Laboratory for Computer Science, M.I.T.

1. Berkowitz, Brian, and Chris Reeve, Templates in MDL, SYS.11.26

2. Daniels, Bruce, The MDL Assembler, SYS.11.07

3. Daniels, Bruce, and Ed Black, MDL Graphics User's Manual, SYS.11.04

4. Galley, S. W., Pre-loaded Pure MDL RSUBRs, SYS.11.28

5. Hewitt, Carl, Planner: A Language for Manipulating Models and Proving Theorems in a Robot, Proc. Intl. Joint Conf. on Artif. Intel., May 1969

6. Moon, David A., MACLISP Reference Manual, Laboratory for Computer Science, M.I.T.

7. Reeve, Chris, The MDL Compiler, SYS.11.25

8. Ryan, Neal D., and Michael S. Broos, MDL Library System Guide, SYS.11.15

Topic Index

Parenthesized words refer to other items in this index.

arguments	"OPTIONAL" "TUPLE" "ARGS" (parameter)
arithmetic	+ - * / ABS EXP LOG SIN COS ATAN MIN MAX RANDOM O? 1? ==? L? G? L=? G=? N==?
array	VECTOR UVECTOR TUPLE STRING BYTES TEMPLATE
assignment	SET SETG DEFINE DEFMAC ENVIRONMENT (value parameter binding)
binding	BOUND? GBOUND? ASSIGNED? GASSIGNED? LEGAL? (assignment value parameter)
bits	WORD BITS PUTBITS GETBITS BYTES ANDB ORB XORB EQVB
block	BIND PROG REPEAT OBLIST BLOCK ENDBLOCK OBLIST MOBLIST OBLIST? !-
boolean	FALSE COND AND AND? OR OR? NOT (comparison)
bugs	(errors)
call	FORM APPLY APPLICABLE? EVAL SEGMENT
change	PUT PUTPROP PUTBITS SET SETG
character	CHARACTER STRING ASCII PRINC READCHR NEXTCHR FLATSIZE LISTEN PARSE LPARSE UNPARSE
circular	PUTREST PUT LENGTH? FLATSIZE
comma	GVAL SETG
comments	; FUNCTION ASSOCIATION
comparison	==? N==? =? N=? G? L=? L? G=? O? 1? MAX MIN STRCOMP FLATSIZE LENGTH? (boolean)
conditional	COND AND OR (boolean)

concatenation	SEGMENT STRING CONS
console	(tty)
coroutine	PROCESS
data type	TYPE TYPE? PRIMTYPE TYPEPRIM CHTYPE UTYPE CHUTYPE NEWTYPE PRINTTYPE APPLYTYPE EVALTYPE ALLTYPES VALID-TYPE?
decimal	.
do	(loops execute call)
dump	SAVE (output)
errors	FRAME ARGS FUNCT ERROR ERRORS ERRET RETRY UNWIND
escape	\ ^G ^S
execute	EVAL APPLY QUOTE FSUBR "ARGS" (call)
exit	RETURN ACTIVATION (goto)
file system	FILECOPY FILE-LENGTH RENAME OPEN OPEN-NR CHANNEL NM1 NM2 DEV SNAME
goto	GO TAG UNWIND PROG REPEAT AGAIN RETURN ACTIVATION "ACT" (loops)
graphics	DISPLAY ERASE PICTURE STORAGE IMAGE
identifier	ATOM PNAME SPNAME LINK LOOKUP INSERT REMOVE OBLIST SPECIAL (parameter value)
if	(conditional)
indexing	NTH GET PUT BACK TOP (loops)
input	READ READCHR NEXTCHR READB READSTRING READ-TABLE GC-READ OPEN ACCESS LOAD FLOAD RESTORE FILE-LENGTH RESET
integer	FIX (arithmetic)
interrupts	EVENT HANDLER ON OFF ENABLE DISABLE INT-LEVEL DISMISS INTERRUPT
iteration	(loops)

leave	(quit)
loading	FLOAD SAVE RESTORE LOAD
location	(pointer)
loops	REPEAT PROG RETURN GO ACTIVATION AGAIN MAPF MAPR STACKFORM ILIST IVECTOR IUVECTOR ISTRING IBYTES IFORM
macro	% %% LINK READ-TABLE PARSE-TABLE DEFMAC EXPAND MACRO
monitor	"READ" "WRITE"
multiprocessing	PROCESS STATE RESUME SUICIDE RESUMER ME MAIN BREAK-SEQ
octal	*
output	PRINT PRIN1 PRINC PRINTB PRINTSTRING IMAGE GC-DUMP ECHOPAIR FLATSIZE SAVE TERPRI CRLF ACCESS RESET BUFOUT NETS
parameter	FUNCTION ATOM LVAL SET SPECIAL UNSPECIAL (identifier value)
parentheses	LIST
parse	PARSE LPARSE PARSE-TABLE UNPARSE
period	LVAL SET READ
pointer	LOCATIVE AT IN SETLOC LIST
predicate	(boolean)
primitives	SUBR FSUBR ROOT GVAL SETG
procedure	FUNCTION DEFINE DEFMAC GVAL CLOSURE
quit	^G ^S QUIT VALRET LOGOUT RETURN (loops)
real	FLOAT (arithmetic)
recursion	(always assumed and built in)
search	MEMQ MEMBER =? ==? (comparison)
sharing	SEGMENT GROW SUBSTRUC

sixbit	UNAME JNAME SEND SEND-WAIT IPC-ON
storage	GC BLOAT BLOAT-STAT FREEZE TUPLE "GC" (structure)
structure	LIST VECTOR UVECTOR STRING BYTES TEMPLATE STRUCTURED? EMPTY? MONAD? LENGTH LENGTH? (concatenation)
subroutine	(procedure primitive)
temporary	"AUX" BIND PROG REPEAT
text	(character)
trailer	!- OBLIST
true	(boolean)
tty	LISTEN ^L ^G ^@ ^D rubout ECHOPAIR TTYECHO TYI "BLOCKED" "UNBLOCKED" ACTIVATE-CHARS (character)
unbinding	(binding)
value	LVAL GVAL VALUE IN SET SETG ENVIRONMENT ASSIGNED? GASSIGNED? BOUND? GBOUND? "BIND" ACTIVATION "ACT" (parameter) RETURN (quit loops)

Token Index

An underscored page number refers to a primary description; an unadorned page number refers to a secondary description.

/	28 151
0?	69
1?	70
1STEP	173
;	24 40
<	24
==?	70 206
=?	70 91
>	24
ABS	28
ACCESS	101 109
ACTIVATE-CHARS	182
ACTIVATION	82 150 181 190 201
AGAIN	83 88 150 174
AGC-FLAG	184
ALLTYPES	46
AND	72 74 183
AND?	72 91
ANDB	64 161
ANY	125
APPLICABLE	125
APPLICABLE?	72
APPLY	47 86
APPLYTYPE	47
ARGS	148 174
ASCII	63
ASOC	123 168 213
ASSIGNED?	74 77 174 185
ASSOCIATIONS	123
AT	117
ATAN	40
ATOM	22 99 142 191 212
AVALUE	123
BACK	58 210
BINARY	165
BIND	82 88
BITS	160

BLOAT	184 193
BLOAT-STAT	194
BLOCK	141 144
BOUND?	77 174 185
BREAK-SEQ	172
BREAKER	172
BUFOUT	101 110 115
BYTE-SIZE	64
BYTES	53 64 64 208
CALLER	163
CHANLIST	102
CHANNEL	64 100 102 102 103 122
CHARACTER	63 99 154
CHTYPE	45 47 206
CHUTYPE	62 211
CLOSE	102
CLOSURE	86
CODE	163
COMMENT	122
COND	73
CONS	58
COS	40
CRLF	99
DEAD	169 169
DECL	124 218
DECL-CHECK	133
DECL?	135
DEFAULT	140
DEFINE	39 147
DEFMAC	156
DEMSIG	198
DEV	101 259
DISABLE	180
DISMISS	174 181 181
DISPLAY	101 113
ECHOPAIR	101 112 146
EMPTY?	72
ENABLE	180
ENDBLOCK	141 144
ENTRY-LOC	165
ENVIRONMENT	37
EQVB	161

Token Index

Token Index

Token Index

Token Index