MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Reply to: Project MAC
545 Technology Square
Cambridge, Mass. 02139

Telephone: (617) 864-6900 x6201

May 4, 1972

TO: DM/CG/CN Group

FROM: Greg Pfister

SUBJECT: Cover letter for SYS.11.01

Two points:

First, the version of MUDDLE described herein corresponds to the file TS NMUDDL on DM/CG/CN ITS. There is also a DM/CG/CN TS MUDDLE and an AI TS MUDDLE. They are basically the same as the MUDDLE described here, but there are some differences.

Second, the printer for this document has a very wide = (equal sign). Hence the symbol ==? comes out as ═?.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Reply to: Project MAC
545 Technology Square
Cambridge, Mass. 02139

Telephone: (617) 864-6900 x6201

July 31, 1972

MEMORANDUM

TO: MUDDLE Users

FROM: Greg Pfister

SUBJECT: Errata for "A MUDDLE PRIMER" (SYS.11.Ø1)

This memo describes changes which should be made to all copies of "A MUDDLE PRIMER" (SYS.11.Ø1). In many cases, these changes correct quite grievous errors, for which I apologize.

Please note that the changes described reflect only errors in the PRIMER; no attempt has been made to update the PRIMER to include features added since it was published, or even to include features which existed when it was published but were undescribed. Such features, plus features which are planned but not yet implemented, include the following:

1. Bit manipulation
2. Non-garbage-collected storage
3. SEGMENTs in FORMs, and the SUBR APPLY
4. User-defined data types.
5. Use of the Evans & Sutherland display (in a state of flux, but usable)
6. "Image mode" teletype output-implying ARDS and IMLAC graphics (probably will change)
7. User-specified interrupt handling (in a state of flux, but usable)
8. Multiple Processes (in a state of flux, but usable)
9. More flexible I/O -- Binary, block mode, internal channels, etc.; a general, upward compatible, reorganization is under way (not yet usable)

If you wish to learn about any of these, see Chris Reeve, Bruce Daniels, Howard Brodie, or myself (or Ed Black, in the case of the Evans & Sutherland Display). Also, read INFO; MUDDLE RECENT for information about newly added features.

The changes follow.

p.12, Sect. 0.1, 2nd line:

"as TS MUDDLE" should be "as SYS:TS MUDDLE"

p.38, Sect. 4.4.2, 1st line:

change to "<NTH <s.o.> <type FIX>>"

p.39, 3rd line:

change "<NTH <type FIX> <s.o.>>" to "<NTH <s.o.> <type FIX>>"

p.40, Sect. 4.4.5, 2nd line:

change "<NTH <type FIX> <s.o.>>" to "<NTH <s.o.> <type FIX>>"

p.46, Sect. 5.3.3, 2nd paragraph, 4th line:

change "<expresion>" to "<expression>"

p.47, Sect. 5.3.3.1, 2nd line:

"(666666)" should be "(66666)"

p.49, Sect. 5.4.2.1.1, 5th line:

"<SET S <REST 15 "Right is might.">>" should be

"<SET S <REST "Right is might." 15>>"

7th line:

change "<BACK .S 6>" to "<BACK .S 6>ϕ"

p.52, Sect. 5.4.4.2, 3rd line:

delete "ht after "<a UVECTOR>"

p.53, Sect. 5.4.4.2.1, 1st line:

change "<UVECTOR 2>>ϕ" to "<IUVECTOR 2>>ϕ"

p.58, 4th line:

change "<PUT ARF" to "<PUT .ARF"

p.63, Sect. 6.2.9, 4th line:

change "#FALSE ()" to "that FALSE"

Sect. 6.2.10, last line:

change "#FALSE ()" to "the last FALSE it saw"

p.65, Sect. 6.3, 3rd paragraph, last line:

change "#FALSE ()" to "the last FALSE it saw"

p.67, Sect. 7.1, 1st paragraph, last line:

Last line should read: "it to be executed interpretively and return a value."

p.81, Sect. 77, last paragraph:

"vaild" should be "valid"

p.85, 3rd paragraph; 3rd line:

"ACTIVATIION" should be "ACTIVATION"

p.98, Sect. 9.2.5.1, 3rd line:

change "its" to "it's"

p.108, Sect. 10.3.2.1, last line:

change "(1 HI 2)" to "(1 HI 3)"

p.109, 1st line:

change "bind ng" to "binding"

p.111, Sect. 11.1.2, 3rd line:

change "<indicaor>" to "<indicator>"

p.112, Sect. 11.2.1, 1st line:

change "<exp>" to "<exp>>"

Sect. 11.2.2, 1st line:

change "<exp>" to "<exp>>"

p.113, Sect. 11.3, 5th line:

change "<PUT L" to "<PUT .L"

delete next two lines.

9th line:

change "<GET L" to "<GET .L"

12th line:

replace "![4!]" with "(1 2 3 4)"

p.114, 5th line:

replace "0.300000" with "(3 4)"

6th line:

change "<2.L>" to "<2 .L>"

11th line:

change "[A B C;" to "'[A B C;"

Insert new paragraph at end of page:

"The ' in the <SET N... is to keep EVAL from generating a new VECTOR ("Direct representation"), which would not have the comment on it."

p.115, Sect. 12.1, 2nd paragraph, 4th line:

change "PNAMEs of ATOMs to all" to "PNAMEs of all ATOMs to"

3rd paragraph, 4th line:

change "only arise" to "arise only"

p.122, Sect. 12.6, 5th line:

change "LVAL of OBLIST" to "LVAL of the ATOM OBLIST"

Last line should read: ""pops" the LVAL of the ATOM OBLIST and returns the resultant LIST of OBLISTs."

p.126, 4th line:

change "symbols" to "symbol(s)" and add the following sentence at the end of the line:

"If you have many, just write them successively."

9th line:

change "notin" to "not in"

IDENTIFICATION

A MUDDLE PRIMER

Greg Pfister

5 May 1972

DRAFT

MOTIVATION

The following document is intended to be a low level introduction to MUDDLE. It is not intended to take the place of a full reference manual, but rather to bring a naive user to the point where he can use such a manual.

REFERENCES

1. Daniels, Bruce, Micro Muddle Manual, SYS.11.03.

BODY

The primer proper follows.

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 did not use (Sorry, Jerry - I got tired.)

A gold star to Fran Knight for proofreading; ditto to Sue Pitkin for typing. And a pox on the paper-shredding terminal typing this out.

CONTENTS

FORWARD

Trying to explain MUDDLE 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 presentation of MUDDLE in an order apparently requiring the fewest forward references. It is not perfect in that regard; however, if the reader is patient and willing to accept a few, stated things as "magic" until they can be explained better, he will probably not have too many problems understanding what is going on.

This document is by no means meant as a substitute for a MUDDLE reference manual. It is instead intended to provide means for "self-teaching" to the point where a (necessarily) highly self-referential manual can be useful.

Note: all examples below are composed of pairs of lines. The first line of a pair always ends in $ (ALT-MODE); this is the input. The second line is the result of MUDDLE's groveling over the first. If the user were to type all the first lines at a MUDDLE, it would respond with the second.

0. BASIC INTERACTION

The purpose of this chapter is to provide you with that minimal amount of information needed to experiment with MUDDLE while reading this primer. It is strongly recommended that you do experiment, especially upon reaching Chapter 7 (FUNCTION).

0.1. LOADING MUDDLE

First, catch your rabbit. Somehow get the program filed as TS MUDDLE running. In MONIT, incant MUDDLE<cr> and in DDT, use MUDDLE^K . TS MUDDLE will first type out some news relating to MUDDLE, 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 MUDDLE. All it knows how to do is interpret MUDDLE expressions. There is no special "command language"; you communicate with TS MUDDLE — make it do things for you — by actually typing legal MUDDLE expressions which TS MUDDLE then interprets. Everything you can do at a console can be done in a

program, and vice versa, in exactly the same way.

0.2. Typing

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

Typing ALT MODE (or ESC) causes TS MUDDLE to echo $ (dollar sign) and causes the contents of the buffer (the characters which you've typed) to be interpreted as a MUDDLE expression. When this interpretation is done, the result will be printed and TS MUDDLE will wait for more typing. ALT MODE will be represented by the glyph $ in this primer.

Typing RUBOUT (or DEL) 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. If no characters are in the buffer, RUBOUT echoes as carriage-return linefeed.

Typing ^@ (control-commercial at) deletes everything you have typed since the last $, and prints a carriage-return linefeed.

Typing ^L (control-L) 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. It may also, on some consoles, clear the screen.

Typing ^G (control-G) causes MUDDLE to stop whatever it is doing and act like an error occurred. (See "simple error discusion" below.) ^G is generally most useful in aborting infinite loops, semi-infinite typeout, and similar terrible things.

If you end your typing with the pair of characters !$ (exclamation point ALT MODE), all currently open parentheses, brackets, etc., will automatically be closed and interpretation will start. Without the !, MUDDLE will just sit there waiting for you to close them. If you have unbalanced parentheses, brackets, etc., within the expression you typed, MUDDLE will attempt to close them correctly and will tell you that something's wrong.

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

0.3. LOADING A FILE

If you have a MUDDLE program you have written as an ASCII file on some device, you can "load" it by typing the following MUDDLE expression and then typing $:

<FLOAD <F1> <F2> <DEV> <USR>>

Each of the objects in <>'s are surrounded by " (double quotes) and

<F1>	is file name 1	initial default: "INPUT"
<F2>	is file name 2	initial default: ">"
<DEV>	is the device	initial default: "DSK"
<USR>	is the user directory	initial default: your UNAME

The only default which "floats" to what you used last is <USR>.

Once you type $, MUDDLE will process the text in the file exactly as if you had typed it on a console and followed it with $. (Including FLOADs in the file.) When MUDDLE is finished processing the file, it will print "DONE" .

Examples:

Loading the file DSK:GFP;HI UGP001 :

```
<FLCAD "HI" "UGP001" "DSK" "GFP">$
"DONE"
```

Loading the file TEST > from your own disk directory upon first entering MUDDLE:

```
<FLCAD "TEST" ">">$
"DONE"
```

0.4. ERRORS — Simple Considerations

When MUDDLE decides for some reason that something is wrong, the normal order of evaluation is interrupted and an error function is called. This produces the following console output:

```
*ERROR*
<reason>
<function in which error occurred>
LISTENING-AT-LEVEL <an integer> PROCESS <an integer>
```

You may now interact with MUDDLE as usual, typing expressions and having them evaluated. There exist facilities (MUDDLE functions) allowing you to find out what went wrong, restart, or kill whatever was going on. In particular, you can recover from an error — i.e., undo everything but side effects and return to the initial typing phase — by typing the following first line, to which MUDDLE 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>), MUDDLE 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.

1. READ, EVAL, and PRINT

1.1. General

Once you type ⊕, the current contents of the input buffer go through processing by three functions successively: first READ, which passes its output to EVAL, which passes its output to PRINT, whose output is typed on the console. Functionally,

READ: printable representations -> MUDDLE objects
EVAL: MUDDLE objects -> MUDDLE objects
PRINT: MUDDLE objects -> printable representations

I.e.: READ takes ASCII text, such as is typed in at a console, and creates the MUDDLE objects represented by that text. PRINT takes MUDDLE objects, creates ASCII text representations of them, and types them out. EVAL, which is the really important one, performs transformations on MUDDLE objects.

1.1.1. Philosophy

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

MUDDLE objects are best considered as abstract entities with abstract properties. The properties of a particular MUDDLE object depend upon the class of MUDDLE objects to which it belongs. This class is the TYPE of the MUDDLE object. Every MUDDLE object has a TYPE, and every TYPE has its own peculiarities. There are many different TYPEs in MUDDLE; 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.

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

Since EVAL is so ever-present, an abbreviation is in order: "evaluates to <something>" shall be taken as an abbreviation for "when given to EVAL, causes EVAL to return <something>".

As abstract entities, MUDDLE objects are, of course, not "visible". There is, however, a standard way of representing

abstract MUDDLE objects in the real world. The standard way of representing any given TYPE of MUDDLE object will be given below when the TYPE is introduced. These standard representations are what READ understands, and what PRINT produces.

1.2. Example (type FIX)

```
1@
1
```

The following has occurred:

First, READ recognized the character 1 as the representation for a MUDDLE object of type FIX, in particular the one which corresponds to the integer 1. It built the MUDDLE 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 correspondng integer.

1.3. Example (type FLOAT)

```
1.0$
1.00000000
```

What went on was entirely analogous to the preceding example, except that the MUDDLE object was of type FLOAT.

1.4. Example (type ATOM, PNAME)

```
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, i.e., a MUDDLE object of

type ATOM. READ therefore attempted looking the representation up in a table it keeps for such purposes (a LIST of OBLISTs, available as the local value of the ATOM OBLIST — ignore the last if it is gibberish). 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> — likewise ignore) 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.

EVAL, given an ATOM, returned just that ATOM.

PRINT, given an ATOM, typed out its PNAME.

Further on, the methods used to attach values to ATOMs will be described; but first, two more things must be covered.

1.5. Structured Objects

To this point, all the objects we have been concerned with have had no internal structure discernible in MUDDLE. 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 their representations.

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.

2. FUNCTIONAL APPLICATION (type FORM)

2.1. Representation

The MUDDLE type which is used to represent the application of a function to its arguments is the 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 the arguments. (The PRIMTYPE of a FORM is LIST — ignore that until you read Chapter 4.)

2.2. Evaluation

EVAL applied to a FORM does the following:

First, examine the first element of the FORM. If it is an ATOM, look at its "value". 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 the first element's directions in evaluating or not evaluating the arguments, (see below) and then "apply the function" — i.e., EVAL the body of the object gotten from <func>.

2.3. Built-in Functions (type SUBR, type FSUBR)

The built-in functions of MUDDLE come in two varieties: those which have all their arguments EVALd before operating on them (type SUBR, for subroutine) and those which have none of their arguments EVALd (type FSUBR, historically from LISP). See Bruce Daniels' Micro Muddle Manual (ref 1) for a listing of all the functions built into MUDDLE, their type, and a short description.

Unless otherwise stated, every MUDDLE function mentioned is of type SUBR. Also, when it is stated that an argument of a SUBR must be of a particular type, note that this means that EVAL of what is there must be of the particular type.

Other convenient abbreviations which will be used are: "the SUBR <PNAME>" in place of "the SUBR which is the 'value' of the ATOM of PNAME <PNAME>". Similarly, "the FSUBR <PNAME>". In cases where the type of the applicable object either does not matter or is assumed known, "the function <PNAME>" will be used. Yet another: "name of the function <something>" for "PNAME of the

ATOM whose 'value' is the <SUBR or FSUBR> <something>".

2.4. Examples

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

The PNAME of the SUBR which adds numbers is + . All of the usual arithmetic functions are MUDDLE SUBRs: +, -, /, *, MIN, MAX, SIN, COS, SQRT, LOG, EXP. They are all indifferent as to whether their arguments are FLOAT or FIX or a mixture. In the latter case, they exhibit "contagious FLOATing"; one argument of TYPE FLOAT forces the result to TYPE FLOAT.

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

FIX is the PNAME of the SUBR which explicitly returns a FIXed point number corresponding to a FLOATing point number. FLOAT does the opposite.

3. VALUES OF ATOMS

3.1. General

There are two kinds of "value" which may be attached to an ATOM. An ATOM may have either, both, or neither. They interact in no way. 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 functions which reference the local and global values of an ATOM, and some of the characteristics of local vs. global values, follow.

3.2. Global Values

3.2.1. SETG

The global value of an atom may be changed by the SUBR SETG , as in

<SETG <an ATOM> <almost anything>>

where <an ATOM> must EVAL to an ATOM, and <almost anything> can be anything but a segment call (see below — ignore it for now.). The EVAL of the second argument becomes the global value of the EVAL of the first argument. The value returned by the SETG is the new global value of the atom.

3.2.1.1. 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.

3.2.2. GVAL

The SUBR with PNAME GVAL is used to reference the global value of an ATOM.

<GVAL <an ATOM>>

returns as a value the global value of <an ATOM>. If <an ATOM> does not evaluate to an ATOM, or if the ATOM it evaluates to has no global value, an error is generated.

GVAL applied to an ATOM anywhere, in any process, in any function, will always 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:

,<anything> <GVAL <anything>>

3.2.2.1. Examples

Assuming the examples in 3.2.1.1 were carried out in the order given, the following will evaluate as indicated:

```
,FOO$
500
,BAR$
FOO
,,BAR$
500
```

3.2.3. Note on SUBRs and FSUBRs

The GVALs of the ATOMs used to reference MUDDLE "built-in" functions 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.

3.3. SET

The SUBR with PNAME SET is used to put a local value on an ATOM. Applications of SET are of the form

<SET <an atom> <almost anything>>

SET returns EVAL of <almost anything> just like SETG.

3.3.1. 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.

3.4. LVAL

The SUBR used to extract the local value of an ATOM is named LVAL. As with GVAL, there is an abbreviation for an application of LVAL: the character . (period). The following two representations are equivalent, and when EVAL operates on their corresponding MUDDLE objects, they return the current local value of <an ATOM>:

<LVAL <an ATOM>> .<an 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.

3.4.1. Examples

Assume all of the previous examples have been done. Then the following evaluate as indicated:

```
.EAR$
1CO
.FCO$
BAR
,.FOO$
FOO
```

3.5. VALUE

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

1) if the ATOM has a GVAL, returns the GVAL
2) if the ATOM has no GVAL, but has an LVAL, returns the LVAL
3) if the ATOM has neither a GVAL nor an LVAL, error.

3.5.1. Examples

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

3.6. EVAL of a FORM, again.

What really happens when EVAL meets a FORM with an ATOM as its first element is that VALUE of the ATOM is used. If the ATOM does not have any values at all, the VALUE fails and produces an error.

4. TYPES AND STRUCTURED OBJECTS

4.1. General

With one exception which has not yet been implemented, all structured objects in MUDDLE are ordered sets. As such, there is a class of functions which operate on all of them uniformly, as ordered sets. These are grouped together immediately below. On the other hand, the reason for there being different types of structured objects is that there are useful qualities of structured objects which are mutually incompatible. There are, therefore, functions which exist to take full advantage of these mutually incompatible qualities which do not work on all structured objects.

The structural organization of an object, i.e., the way it is stored in memory, is referred to as its "primitive type". While there are many different types of structured objects in MUDDLE, there are very few (for all practical purposes four) primitive types. Each of the four most consciously used primitive types is discussed in Chapter 5, along with those special functions operating on each primitive type. For each of the types discussed in Chapter 5, its primitive type is the same as its type.

In all the following, <s.o.> will be used as a symbol for

any structured object.

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

4.2. SUBRs related to TYPEs

4.2.1. TYPE

<TYPE <anything>>

returns an ATOM whose PNAME corresponds to the TYPE of <anything>. There is no TYPE TYPE. To type a TYPE, just type the appropriate ATOM. Like FIX or FLOAT or ATOM etc.

4.2.2. PRIMTYPE

<PRIMTYPE <anything>>

evaluates to the primitive type of <anything> — not just structured objects. The PRIMTYPE of <anything> is an ATOM which usually also represents a TYPE. The way an object can be manipulated depends solely upon its PRIMTYPE (the way it is evaluated depends upon its TYPE).

4.2.3. CHTYPE

<CHTYPE <something> <a TYPE> >

returns <something> changed to TYPE <a TYPE>. However: an error is generated if the PRIMTYPE of <something> would have to change to accomodate the TYPE change.

4.2.4. ALLTYPES

<ALLTYPES>

returns a VECTOR (see Chapter 5) containing just those ATOMs which can ever be returned by TYPE or PRIMTYPE.

4.3. General Representational Format

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

<its TYPE> <representation as if it were its PRIMTYPE>

READ will understand that format for any structured TYPE.

4.4. Basic Functions

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

4.4.1. LENGTH

<LENGTH <s.o.>>

Evaluates to the number of members of <s.o.>.

4.4.2. NTH

<NTH <type FIX> <s.o.>>

Evaluates to the <type FIX>'th element of <s.o.>. Error if <type FIX> is 0 or less, or greater than <LENGTH <s.o.>>. EVAL understands the application of an object of type FIX as a

"shorthand" call to NTH. I.e., EVAL considers the following two to be identical:

<NTH <type FIX> <s.o.>> <<type FIX> <s.o.>>

4.4.3. REST

<REST <s.o.> <type FIX>>

Evaluates to <s.o.> without its first <type FIX> elements. The second argument is optional, with 1 assumed.

Obscure side effect: REST actually returns <s.o.> "CHTYPEd" (but not through application of CHTYPE) to its PRIMTYPE. E.g., REST of a FORM is a LIST. REST with an explicit second argument of 0 has no effect except for this TYPE change.

4.4.4. PUT

<PUT <s.o.> <type FIX> <anything legal>>

First makes <anything legal> the <type FIX>'th element of <s.o.>, then evaluates to <s.o.>. <anything legal> is anything which can legally be a member of <s.o.>; often, this is synonymous with "any MUDDLE object", but see Chapter 5. Error if <type FIX> is 0 or less, or greater than <LENGTH <s.o.>>.

PUT is actually more general than this. See Chapter 11.

4.4.5. GET

<GET <s.o.> <type FIX>>

Evaluates the same as <NTH <type FIX> <s.o.>>. It is more general than NTH, however, and is included here only for symmetry with PUT. See Chapter 11.

5. BASIC TYPES OF STRUCTURED OBJECTS

5.1. Representations

5.1.1. LIST

(<element 1> <element 2> . . . <element N>)

represents a LIST of N elements.

5.1.2. VECTOR

[<element 1> <element 2> . . . <element N>]

represents a VECTOR of N elements.

5.1.3. UVECTOR

![<element 1> <element 2> . . . <element N> !]

represents a UVECTOR (uniform vector) of N elements. The second ! (exclamation point) is optional.

5.1.4. STRING

"<characters>"

represents a STRING of ASCII text.

5.2. BASIC EVALuation of BASIC STRUCTURES

5.2.1. Basic

EVAL of a STRING is just the original STRING.

EVAL acts exactly the same with LISTs, VECTORs, and UVECTORs. Basically, it generates a new object with elements equal to EVAL of the elements it is given. This is the basic

means of constructing a given structure. However, see "segment evaluation" below.

5.2.2. Many Linked Examples

```
(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]
<PUT .FOO 1 SNEAKY>$
[SNEAKY -3 STRING]
.BAR$
![("meow") ([SNEAKY -3 STRING])!]
<SET FOO <REST <1 <1 .BAR>> 2>>$
"ow"
```

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

5.3. Generation

Since LISTs, VECTORs, UVECTORs, and STRINGs are all generated in a fairly uniform manner, methods of generating them will be covered together here.

5.3.1. Direct Representation

Since EVAL of a LIST, VECTOR, or UVECTOR is a new LIST, VECTOR, or UVECTOR with elements which are EVAL of the original, simply writing down the representation of the object you want will generate it. This method of generation was exclusively used in the examples of 5.2.2.

Note that new STRINGs cannot be generated in this manner, since the contents of a literal STRING are not interpreted.

5.3.2. The SUBRs LIST, VECTOR, UVECTOR, and STRING

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 below.

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

5.3.2.1. 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)"
```

5.3.3. The SUBRs ILIST, IVECTOR, IUVECTOR, and ISTRING

Each of the SUBRs ILIST, IVECTOR, IUVECTOR, and ISTRING creates and returns an object of the obvious TYPE. The format of an application of any of them is:

< <I> <TYPE FIX> <expression> >

where <I> is one of ILIST, IVECTOR, IUVECTOR, or ISTRING. An object of LENGTH <TYPE FIX> is generated, and its elements are set to EVAL of <expression>.

<expression> is optional. When it is not specified, ILIST, IVECTOR, and IUVECTOR return objects filled with objects of TYPE LOSE, which can be passed around and have its TYPE checked, but otherwise is an illegal argument. If <expresion> is not specified in ISTRING, you get a STRING made up of DEL (or RUBOUT) characters, which do not print.

When <expression> is supplied as an agrument, 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 may be used, and don't worry about the ' (single quote); it effectively supresses the initial SUBR EVALuation, and is fully explained in chapter 7.

IUVECTOR and ISTRING again have limitations on what <expression> may EVAL to; again, see below.

5.3.3.1. Examples

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

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

5.4. Unique Properties

5.4.1. LIST (the type)

A LIST may be considered as a "pointer chain". Any MUDDLE object may be a member of a LIST. It is easy to add and remove elements on a LIST, but the higher N is, the harder it is to access the Nth element. The only function which works only on objects of PRIMTYPE LIST is:

5.4.1.1. PUTREST

<PUTREST <a LIST 1> <a LIST 2>>

changes <a LIST 1> so that <REST <a LIST 1>> is <a LIST 2>, then evaluates to <a LIST 1>. Note that this actually changes <a LIST 1>; using it will also change anything having <a LIST 1> as an element or a value. See example below.

5.4.1.1.1. PUTREST Example

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

5.4.2. VECTOR, UVECTOR, and STRING

VECTORs, UVECTORs, and STRINGs may be considered as "arrays". 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 functions may be used only with an

object of PRIMTYPE VECTOR, UVECTOR, or STRING (note — <VUS> is an object whose PRIMTYPE is VECTOR, UVECTOR, or STRING.).

5.4.2.1. BACK

<BACK <VUS> <type FIX>>

This is the opposite of REST. It evaluates to <VUS> with <type FIX> elements back on its front end. If <type FIX> is greater than the number of elements which have been RESTed off, error.

5.4.2.1.1. BACK Examples

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

5.4.2.2. TOP

<TOP <VUS>>

"BACKs up all the way" — i.e., evaluates to <VUS> with all the elements which have been RESTed off back on it.

5.4.2.2.1. TOP Example

```
<TOP .ZOP>#
[1 2 3 4]
```

5.4.3. VECTOR (the type)

Any MUDDLE object may be an element of a VECTOR. A VECTOR takes two words of storage more than an equivalent LIST, but takes it in a contiguous chunk whereas a LIST may be physically spread out. There are no SUBRs or FSUBRs which operate only on VECTORs without also being applicable to UVECTORs and STRINGs.

5.4.4. UVECTOR (the type)

The difference between a UVECTOR and a VECTOR is that every element of a UVECTOR must be of the same type. UVECTORs take half the storage of either VECTORs or LISTs, and like VECTORs, take it in a contiguous chunk.

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 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 if .A EVALs to a type FIX.

The following SUBRs work on UVECTORs alone.

5.4.4.1. UTYPE

<UTYPE <a UVECTOR>>

evaluates to the type of every element in <a UVECTOR>.

5.4.4.1.1. UTYPE Example

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

5.4.4.2. CHUTYPE

<CHUTYPE <a UVECTOR> <a TYPE>>

changes the UTYPE of <a UVECTOR> to <a TYPE>, simultaneously changing the type of all elements of <a UVECTOR>ht, and returns the new, changed, UVECTOR. This works only when the PRIMTYPE of the elements of <a UVECTOR> can remain the same through the whole process. The PRIMTYPE of LOSE is indeterminate; a UVECTOR of UTYPE LOSE can be CHUTYPEd to anything. (If the anything is structured, elements of the UVECTOR are empty.)

5.4.4.2.1. CHUTYPE Example

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

5.4.5. STRING (the type)

The best mental image of a STRING is a UVECTOR of CHARACTERs — where CHARACTER is the MUDDLE type for a single character. The representation of a CHARACTER, by the way, is

!"<any ASCII character>

That is, the characters !" (exclamation point double quote) preceding a single ASCII character represent the corresponding object of TYPE CHARACTER.

The SUBRs STRING and ISTRING will produce an error if you

attempt to cause them to put other than a CHARACTER or a STRING into a STRING.

There are no functions which uniquely manipulate STRINGs, but one is particularly useful in connection with them:

5.4.5.1. ASCII

<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.

If its argument is of type CHARACTER, ASCII evaluates to the FIXed point number which is its argument's 7-bit ASCII code.

5.5. Segment Evaluation

Segment evaluation is a method of evaluating structured objects which is designed to be a very convenient method for constructing structures from other structures. The only place segment evaluation is legal is within the EVAL of a structured object, and it can only be applied to another structured object. All it consists of is taking the members of the structure segment evaluated and placing them into the structure being constructed.

5.5.1. Type and Representation

Segment evaluation is done only on an object of a particular type, namely type SEGMENT. The representation of an object of type SEGMENT is the following:

!< <el1> <el2> . . . <elN> !>

where the second ! (exclamation point) is optional, and <el1> through <elN> are any legal consituents of a FORM (i.e., just about 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

5.5.2. Evaluation

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

1) It had better be done inside an EVAL of a structure, else error.

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

3) What actually gets inserted into the structure

being built is the elements of the structure returned by the FORM-like evaluation.

5.5.3. SEGMENT Examples

```
<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!]
```

5.5.4. Note on Efficiency

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 LIST, and the segment value of a LIST is the last (rightmost) element being concatenated, that last LIST is not copied. This case is equivalent to the LISP

CONS, and is the reason why LISTs have their structure more easily varied than VECTORs or UVECTORs.

5.5.4.1. Examples

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

This does not copy ARF:

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

These do:

```
(1 !.ARF 2)$
(1 B 3 4 2)
[1 2 !.ARF]$
[1 2 B 3 4]
(1 2 !.ARF !<REST (1)>)$
(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)
```

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

6. TRUTH

6.1. Truth Values

MUDDLE represents "false" with an object of a particular type: type FALSE (unsurprisingly). Type FALSE is structured; its PRIMTYPE is LIST. Thus, you can give excuses by making them elements of the FALSE. Objects of type FALSE are represented by

#FALSE <LIST of its elements>

The empty FORM evaluates to an empty FALSE:

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

In addition, there is a SUBR of PNAME FALSE which takes one argument — a LIST — and CHTYPEs it to FALSE.

Anything which is not FALSE, is, of course, true.

6.2. Predicates

There are numerous MUDDLE functions which can return FALSE or true. See Micro-Muddle Manual (ref 1) 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:

6.2.1. ==?

<==? <e 1> <e 2>>

evaluates to T only if <e 1> is the same object as <e 2>.

6.2.2. =?

<=? <e 1> <e 2>>

evaluates to T if <e 1> and <e 2> are structurally equal — i.e., they "look the same", their printed representations are the same. =? is much slower than ==?. =? should only be used when its characteristics are necessary; they usually are not in any comparisons of unstructured objects.

6.2.3. 0?

<0? <type FIX or FLOAT>>

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

6.2.4. 1?

<1? <type FIX or FLOAT>>

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

6.2.5. G?

<G? <N> <M>>

evaluates to T only if <N> is algebraically greater than <M>. <N> and <M> may indiscriminately be either FIX or FLOAT.

6.2.6. L?

<L? <N> <M>>

evaluates to T only if <N> is algebraically less than <M>. <N> and <M> may indiscriminately be either FIX or FLOAT.

6.2.7. MONAD?

<MONAD? <e>>

evaluates to #FALSE () only if NTH and REST can be performed on its argument without error.

6.2.8. EMPTY?

<EMPTY? <s.o.>>

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

6.2.9. AND

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

AND is an FSUBR. It evaluates its arguments from left to right as they appear in the FORM. As soon as one of them evaluates to FALSE, it returns #FALSE (). If none of them evaluate to FALSE, it returns EVAL of its last argument.

6.2.10. OR

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

OR is an FSUBR. It evaluates its arguments from left to right as they appear in the FORM. As soon as one of them evaluates to non-FALSE, OR returns that non-FALSE value. If this never occurs, it returns #FALSE ().

6.2.11. NOT

<NOT <e>>

evaluates to T only if <e> evaluates to FALSE.

6.2.12. MEMBER

<MEMBER <object> <structured object>>

This SUBR runs down <structured object> from first to last element, comparing each element of <structured object> with <object>. If it finds an element of <structured object> which is =? to <object>, it returns <REST <n - 1> <structured object>>, where the nth element of <structured object> is =? to <object>. I.e., the first element of what it returns is the first element of <structured object> =? to <object>. If no element of <structured object> is =? to <object>, MEMBER returns #FALSE ().

6.2.13. MEMQ

<MEMQ <object > <structured object>>

This SUBR is exactly the same as MEMBER, except that the comparison test is ==? .

6.3. COND

The MUDDLE function which is most used for varying evaluation depending on a truth value is the FSUBR COND. A call to COND has this format:

<COND <L1> <L2> . . . <LN>>

where <L1> through <LN> are LISTs.

COND evaluates as follows, examining its input lists from left to right as they appear in the FORM:

1) If there are no lists left unexamined, return #FALSE ().

2) Evaluate the first element of the first list still unexamined. If it evaluates to FALSE, go back to (1).

3) Evaluate in order the rest of the elements of the current list and return the last thing evaluated.

I.e., COND goes walking down its lists, EVALing the first member of each list looking for a non-FALSE. As soon as it finds a non-FALSE, it forgets about all the other lists and evaluates, in order, the other elements of the current list and returns the last thing it evaluates. If it can't find a non-FALSE, it returns #FALSE().

6.3.1. Examples

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

7. FUNCTION

7.1. General

An object which is of TYPE FUNCTION is of PRIMTYPE LIST. It is what its name implies, i.e., a function. You write it, apply it in a FORM with (or without) arguments, and EVAL causes it to be executed interpretively.

The FSUBR FUNCTION can be used to create objects of type FUNCTION. It is really a very simple routine; it just takes its arguments in a LIST, and CHTYPEs the LIST to type FUNCTION.

In many of the examples below, that which MUDDLE would print in response to the typing of the FUNCTION "definitions" would be both tedious and singularly unenlightening. So, instead of reproducing that output, a line like this:

- - - -

will be used instead.

7.2. Simple Case

In its simplest form, a function has two parts: a LIST of dummy variables, as its first element; and a body — all its other elements. When an application of a FUNCTION in a FORM is evaluated, the dummy variables are bound to the actual arguments, and each MUDDLE object in the body is evaluated in order. The result of the last evaluation is returned as the value of the FUNCTION. Exactly what goes on will be described through the following simple example.

7.2.1. Example

```
<SETG F <FUNCTION (A B) <+ .A .B>>>
#FUNCTION ((A B) <+ .A .B>)
```

The above set the global value of the ATOM of PNAME F to the object of type FUNCTION indicated — a FUNCTION of 2 arguments which just adds them and returns the result. Its argument declaraction is the LIST (A B), and its body consists of the single FORM <+ .A .B>. Note that there is no special representation for type FUNCTION; the default representation is used. Since the latter is a perfectly good way to describe a FUNCTION on input, we could have avoided calling the FSUBR FUNCTION by typing the following, which has exactly the same effect as the above:

```
<SETG F #FUNCTION ((A B) <+ .A .B>)>@
#FUNCTION ((A B) <+ .A .B>)
```

Now, suppose we apply F to something:

```
<F 1 2>@
3
```

What happened was this:

EVAL saw the ATOM F as the first element of a FORM, and found its GVAL to be our FUNCTION. It then examined F's argument declaration, and bound the two ATOMs there — A and B — to the EVAL of the arguments — 1 and 2 respectively; i.e., it made A and B's local values 1 and 2. Having done that, it then executed the body — EVALd the FORM <+ .A .B> — saw that there was nothing more to do, and returned the result of that EVAL after unbinding A and B.

The relationship of "binding" and LVALs is this: binding a "dummy variable" consists of pushing its current LVAL onto a stack and then giving it a new LVAL. "Unbinding" consists of popping that stack. This comes under the heading of "dynamic blocking"; its application to recursion is clear.

The fact that the EVAL is done before the binding means that this works:

```
<SET A 1>$
1
<F 1 <+ .A 1>>$
3
```

Note that if we were to SET A to something within F, it would have no effect on the LVAL of A after returning from F, since the value we SET it to would be "popped". That produces, in some cases, a problem. Suppose we wish to write a FUNCTION which takes an ATOM as an argument, and increments the LVAL of that ATOM. We could write it like this:

```
<SETG INC <FUNCTION (A) <SET .A <+ 1 ..A>>>>$
#FUNCTION ((A) <SET .A <+ 1 ..A>>)
```

We give this FUNCTION an ATOM, which becomes the LVAL of A. So we SET that ATOM — i.e., .A — to 1 plus the LVAL of the ATOM — i.e., 1 plus ..A . In many cases, this works fine:

```
<SET ATM 0>$
0
<INC ATM>$
1
.ATM
1
```

However, if we happen to call it with the ATOM of PNAME A, we lose. Try it. (Exercise for the reader.) There are two ways of getting around this last problem, both of which will eventually be mentioned. A third, inelegant, and imperfect method is to note that since MUDDLE sets no limit on the length of identifiers, we could use an intentionally long and stupid identifier in place of A, thus lessening the chances of conflict. Of course, somebody else, trying to do the same thing, is bound to use just that identifier.

7.2.2. Factorial and Comments

I clearly can't not use factorial as an example — so, the basic recursive factorial FUNCTION follows. I might as well introduce comments at the same time.

```
<SETG FACT <FUNCTION (A)        ;"Basic factorial function."

        <COND (<L? .A 2> 1)     ;"If arg is less than 2, return 1."

              (<* .A <FACT <- .A 1>>>) ;"Else recurse." >>>@

#FUNCTION ((A) <COND (<L? .A 2> 1) (<* .A <FACT <- .A 1>>>)>)
```

Note that carriage-returns, line-feeds, tabs, spaces, etc. just mean "separator". In particular, they have nothing to do with delineating comments.

Also note that in the second clause of the COND, its first element is non-FALSE; it's either FIX or FLOAT. Alternatively, you could stuff an ATOM in there — like T or ELSE for mnemonic reasons — or anything but a FALSE.

Finally note that the indicator for a comment is the character ; (semicolon). When READ sees a semicolon, it attaches the next MUDDLE object to the last structural element seen as that element's COMMENT property (see Chapter 11). The first comment above is attached to FACT's argument list, the second to the first clause of the COND, and the third to the second clause of the COND. Comments are thus remembered along with the object with which they are associated, but have no effect on either EVAL or PRINT. They can, however, be gotten back (again, see Chapter 11). In the example above, the MUDDLE objects which are comments are STRINGs; this is usually the case,

but is clearly not necessary. They could be ATOMs, LISTs, FUNCTIONs (to name some useful possibilities) or anything else.

7.3. "OPTIONAL"

MUDDLE provides very convenient means for allowing optional arguments. Inserting the STRING "OPTIONAL" in the argument declaration allows the specification of optional arguments with default values. The syntax of the "OPTIONAL" part of the argument declaration is as follows:

"OPTIONAL" <AoL1> <AoL2> . . . <AoLN>

First, there is the STRING "OPTIONAL". Then there is any number of either ATOMs or two element LISTs, 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 MUDDLE expression. If there are required arguments, these 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, i.e., is not the first element of a two element LIST, that ATOM becomes "bound", but no local value is assigned to it. A local value can be assigned to it by using SET. Until an ATOM is assigned, any attempt to reference it other than as an argument to the predicate SUBRs BOUND? and ASSIGNED? (which return T under the obvious condition) will produce an error.

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

By the way, there is one other predicate similar to BOUND? and ASSIGNED?, namely GASSIGNED? . The latter returns T if its argument, which (as in BOUND? and ASSIGNED?) must be an ATOM, has a global value.

Since an ATOM can be BOUND? but not ASSIGNED?, and applying ASSIGNED? to an unbound ATOM produces an error, some care must be used if you wish to find out whether an ATOM has a local value.

The following, by virtue of the fact that AND is an FSUBR, will return T if <A> has a local value, FALSE if it does not, and never produce an error:

```
<AND <BOUND? <A>> <ASSIGNED? <A>>>
```

7.3.1. "OPTIONAL" Example

```
<SETG INC1 <FUNCTION (A "OPTIONAL" (N 1))
        <SET .A <+ ..A .N>>>>@
#FUNCTION ((A "OPTIONAL" (N 1)) <SET .A <+ ..A .N>>)
<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 MUDDLE expression; there is no reason why the optional argument need not be hairy — e.g., a call to a FUNCTION which reads a file on an I/O device.

7.4. "TUPLE" and TYPE TUPLE

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 LIST or VECTOR as its argument, with the arbitrary number of arguments as elements of the LIST or VECTOR (or UVECTOR, for that matter). This can, however, lead to

inelegant looking FORMS. The STRING "TUPLE" appearing in the argument declaration allows you to avoid that. It must follow explicit and optional arguments (if there are any of either) and must be followed by an ATOM.

The effect of "TUPLE" appearing in an argument declaration is the following: Any arguments left in the FORM after satisfying explicit and optional arguments are EVALd and made sequential members of an object of TYPE and PRIMTYPE TUPLE. The TUPLE is then bound to the ATOM following "TUPLE" in the argument declaration. 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 most other bindings in a stack. This does not effect 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.)

7.4.1. "TUPLE" Example

```
<SETG NTHARG <FUNCTION (N "TUPLE" T)
                       ;"Snarf all but first argument into T."
    <COND (<1? .N> 1)  ;"If N is 1, return 1st arg, i.e., .N,
                         i.e., 1"
          (<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, bitch."
          (ELSE <.N .T>)>>>
```

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 truly necessary in the last clause because the Nth argument might be a 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. Now make it work with no arguments.

7.5. "AUX" and "EXTRA"

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

"AUX" (or "EXTRA") must appear in the argument declaration after any information about explicit arguments. It is followed by ATOMs or 2-element LISTs as if it were "OPTIONAL". ATOMs in the 2-element LISTs are bound to the EVAL of the second element in the LIST. Atoms not in such lists are initially bound to an object of type UNASSIGNED, namely #UNASSIGNED 0 .

All binding specified in an argument declaration is done sequentially from left to right, so initialization expressions for "AUX" can refer to objects which have just been bound. For example, this works:

```
<SETG AUXEX
      <FUNCTION ("TUPLE" T
                 "AUX" (A <LENGTH .T>) (B <* 2 .A>))
                (.A .B)>>$

- - - -

<AUXEX 1 2 "FOO">$
(3 6)
```

7.6. QUOTE

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

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

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

```
<SETG Q2 <FUNCTION (A 'B) (.A .B)>>$
- - - -
<Q2 <+ 1 2> <+ 1 2>>$
(3 <+ 1 2>)
```

7.7. "ARGS"

The indicator "ARGS" can appear in an argument declaration with precisely the same syntax as "TUPLE". "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 <the FORM applying this FUNCTION> <FIX>>

with an appropriate <FIX>. The type change to LIST is a result of the REST. Examples:

```
<SETG QIT <FUNCTION (N "ARGS" L) <.N .L>>>@
- - - -
<QIT 2 <+ 3 4> <LENGTH ,QALL> FOO>@
<LENGTH ,QALL>

<SETG FUNCTION
      <FUNCTION ("ARGS" ARGL_AND_BODY)
                <CHTYPE .ARGL_AND_BODY FUNCTION>>>@
- - - -
<FUNCTION (A B) <+ .A .B>>@
#FUNCTION ((A B) <+ .A .B>)
```

The last example is a perfectly vaild definition of FUNCTION.

7.8. "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 whose application is being evaluated — i.e., 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. Please note that such techniques will not work if the calling code has been compiled. (Neither will "CALL".)

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

7.9. EVAL and "BIND"

Obtaining unevaluated arguments, e.g., 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 takes one optional argument, of TYPE ENVIRONMENT. An ENVIRONMENT consists basically of all the information needed at any given time by EVAL. Now, binding changes the ENVIRONMENT; so 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 declaration, provides this information. It binds the ATOM immediately following it to the ENVIRONMENT existing "at call time" — i.e., just before any binding is done for its FUNCTION. Example:

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

7.10. ACTIVATION, "NAME", "ACT", AGAIN, and EXIT

EVALuation of a FUNCTION, after the argument declaration has been taken care of, normally conists of EVALuating each of the objects in the body in the order given, and returning value of the last thing EVALd. If you want to vary this sequence, you need to know, at least, where the FUNCTION begins. Not surprisingly, if you think about it, 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; they can be generated, and bound to an ATOM, in either of the two

following ways:

1) Put an ATOM immediately before the argument declaration. ACTIVATION of the FUNCTION will be bound to that ATOM.
2) As the last thing in the argument declaration, insert either of the STRINGs "NAME" or "ACT" and follow them with an ATOM. The ATOM will be bound to the ACTIVATION of the FUNCTION.

Each ACTIVATION refers explicitly to a particular application of a FUNCTION. E.g., 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 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 applications; when the application no longer exists, neither does the ACTIVATION.)

ACTIVATIONs are used by the SUBRs AGAIN and EXIT.

AGAIN takes 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 body of the FUNCTION in the application specified by the ACTIVATION. The application is not re-evaluated completely; in particular, no re-binding (of arguments, "AUX" variables, etc.) is done.

EXIT takes 2 arguments: an ACTIVATION and an arbitrary expression. It causes the FUNCTION EVALuation whose ACTIVATION it is given to terminate and return EVAL of EXIT's second argument. I.e., EXIT means "quit doing this and return that", where "this" is the ACTIVATION — its first argument — and "that" is the expression — its second argument. Example:

```
<SETG MY+ <FUNCTION ("TUPLE" T "AUX" (M 0) "NAME" NM)
        <COND   (<EMPTY? .T> <EXIT .NM .M>)>
        <SET M <+ .M <1 .T>>>
        <SET T <REST .T>>
        <AGAIN .NM>>>$
- - - -
<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) — e.g., via an application of F2 in F1 with F1's ACTIVATIION as an argument. If F2 EXITs with F1's ACTIVATION, F2 and F1 terminate immediately, and F1 returns the EXIT's second argument. Good for error exits.

AGAIN can clearly pull a similar trick.

8. PROG and REPEAT

8.1 General

PROG and REPEAT are almost identical FSUBRs which make it possible to arbitrarily vary the order of EVALuation — i.e., have "jumps". Syntax of a PROG is:

<PROG <optional activation> <prog list> <body>>

where

<optional activation> is an optional ATOM, which is bound to the ACTIVATION of the PROG.

<prog list> is a LIST which looks exactly like that segment of a FUNCTION's argument declaration which follows an "AUX", and serves exactly the same purpose. It is not optional. If you need no temporary variables, make it ().

<body> is an arbitrary number of arbitrary MUDDLE expressions.

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

8.2. Basic PROG EVALuation

Upon entering a PROG, an ACTIVATION is always generated. If there is an ATOM in the right place, it is also bound to that ATOM. The variables in the <prog list> (if any) are then bound as indicated in the <prog list>. Each 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.

8.3. AGAIN and RETURN

Within a PROG, you always have a defined ACTIVATION, whether you bind it to an ATOM or not.

If AGAIN is used with no arguments within a PROG, it uses the ACTIVATION of the closest surrounding PROG, and re-starts the PROG without rebinding the <prog list> variables, just like it works in a FUNCTION. With an argument, it can, of course re-start any PROG or FUNCTION within which it is embedded at run time.

To leave a PROG without evaluating any more of it, use the SUBR RETURN. RETURN takes one argument and causes the first PROG in which it is embedded to return EVAL of that ARGUMENT. EXIT can also be used, with an explicit ACTIVATION of course, to do the same thing.

8.4. REPEAT EVALuation

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 EXIT (or GO with a TAG — see below).

8.5. GO and TAG

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, starting after <prog list>, 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 position in a PROG from within any FUNCTION called inside the PROG. 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.

9. I/O

9.1. General - Basic

All I/O FUNCTIONs in MUDDLE take an optional argument which directs their attention to specific I/O channels. The I/O FUNCTIONs will first be described without their optional arguments. In this situation, they all refer to the initial default of TTY. When given an optional argument, that argument follows any arguments indicated here.

9.1.1. Input

All of the following input routines, when directed at the TTY, hang until $ (ALT MODE) is typed and allow normal use of rubout and ^L.

9.1.1.1. READ

<READ>

This returns the entire MUDDLE object whose representation is next in the input stream. Successive <READ>s return successive objects.

9.1.1.2. READCHR

<READCHR>

This returns the next CHARACTER in the input stream. Successive <READCHR>s return successive CHARACTERs.

9.1.1.3. NEXTCHR

<NEXTCHR>

This returns the CHARACTER which READCHR will return the next time READCHR is called. Multiple <NEXTCHR>s, with no READs or READCHRs between them, all return the same thing.

9.1.2. Output

9.1.2.1. PRINT

<PRINT <obj>>

This outputs, in order,

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

and then returns EVAL of its argument. This is precisely the SUBR PRINT mentioned in chapter 1.

9.1.2.2. PRIN1

<PRIN1 <obj>>

outputs just the representation of EVAL of <obj>. Returns EVAL of its argument.

9.1.2.3. PRINC

<PRINC <obj>>

acts exactly like PRIN1, except that if its argument is a STRING or a CHARACTER, it suppresses the surrounding "s or initial !" respectively.

9.1.2.4. FLATSIZE

<FLATSIZE <obj> <FIX>>

does not actually cause any output to occur. Instead, it first finds out how many characters PRINT would take to print <obj>, and then compares that number with <FIX>. If <FIX> is less than the number of characters needed, FLATSIZE returns #FALSE (); otherwise, it returns the number of characters needed to PRINT

<obj>.

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

9.2. CHANNELs

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

9.2.1. OPEN

<OPEN <dir> <F1> <F2> <dev> <usr>>

OPEN is a SUBR which creates and returns a CHANNEL (in both the ITS and MUDDLE senses of the word). All its arguments must be of TYPE STRING, and all are OPTIONAL. If the attempted opening of an ITS I/O channel fails, OPEN returns #FALSE (). Argument descriptions:

<dir> must be "READ" for input or "PRINT" for output.
Default: "READ".

<F1> is the first file name. Default: "INPUT" if <dir> is "READ", "OUTPUT" if <dir> is "PRINT".

<F2> is the second file name. Default: ">".

<dev> is the device. Default: "DSK"

<usr> is the user directory. Default: your current one.

9.2.2. CLOSE

<CLOSE <a CHANNEL>>

closes <a CHANNEL> and returns its input, with its "state" changed to "closed".

9.2.3. CHANLIST

<CHANLIST>

returns a LIST whose elements are all the currently open CHANNELs.

9.2.4. 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. See, however, Input Errors below.

By the way, a good hack 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.

9.2.5. 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 (e.g., 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).

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>

9.2.5.1. Output CHANNELs

The contents of a CHANNEL used for output are:

* 1:	FIX	channel number (0 means channel not open)
* 2:	STRING	direction (for output, its "PRINT")
* 3:	STRING	device name argument
* 4:	STRING	first file name argument
* 5:	STRING	second file name argument
* 6:	STRING	directory name argument
* 7:	STRING	real device name
* 8:	STRING	real first file name
* 9:	STRING	real second file 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 (not yet used)
18:	FIX	radix for number conversion

9.2.5.2. Input CHANNELs

The contents of a CHANNEL used for input from a file-oriented device is the same as the contents of the corresponding CHANNEL used for output, except that element 2 (direction) contains "READ".

A CHANNEL used for input for a console variety device (e.g., "TTY") has, in addition, element 15 set to a UVECTOR of UTYPE LOSE which is used as an input buffer.

9.3. Input Errors

An explicit CHANNEL for input is the second optional argument of all SUBRs used for input. The first optional argument is an error routine — i.e., something for the input SUBR to EVAL if it detects an error. A typical error argument is a QUOTEd FORM which calls an error routine of yours. If not given, the standard error SUBR — ERROR — is used. (Since attempting to read past the End-of-file is an error, if you really don't expect errors you can use an application of RETURN or EXIT as an "error" routine to bounce you out of a read loop. You can usually use <CLOSE <the channel>> as the thing RETURN or EXIT returns and kill two birds with one stone.)

Note that input from the TTY "hangs" until ALT-MODE is typed; then you start getting successive items or characters.

9.3.1. Example

The following FUNCTION outputs to .OUTCHAN a file read according to its arguments. The static variables which are initially SET to the funny strings hold the default arguments. (The funny strings are the initial defaults.)

```
<SET DF1 "______">
<SET DF2 "______">
<SET DDEV "DSK">
<SET DUSR "">

<SETG PF
  <FUNCTION ("OPTIONAL" (F1 .DF1) (F2 .DF2)
                        (DEV .DDEV) (USR .DUSR)
            "AUX" (CHN <OPEN "READ"
                             <SET DF1 .F1> ;"Set up defaults"
                             <SET DF2 .F2> ;"for next call."
                             <SET DDEV .DEV>
                             <SET DUSR .USR>>))
    <COND (.CHN          ;"If CHN is FALSE, bad OPEN, else O.K."
           <REPEAT ()
              <PRINC <READCHR '<RETURN <CLOSE .CHN>> .CHN>>
            >            ;"Until EOF, keep reading and printing
                              a character at a time."
           DONE.         ;"Then return this ATOM.")
          (ELSE #FALSE ("BAD FILE NAME")
                        ;"Return a FALSE so user can test it
                       easily if used inside another FUNCTION."
           )>>>
```

9.4. Other I/O Functions

9.4.1. LOAD

<LOAD <input CHANNEL> <OB>>

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

<OB> is optional, and may be used to specify a LIST of OBLISTs for the READ. Its default is .OBLIST. See far below.

9.4.2. FLOAD

<FLOAD <F1> <F2> <DEV> <USR> <OB>>

acts just like LOAD, except that it takes arguments like OPEN, OPENs the CHANNEL itself for reading, and CLOSEs the CHANNEL when done. <OB> is optional, as in LOAD.

9.4.3. ECHOPAIR

<ECHOPAIR <TTY input CHANNEL> <TTY output CHANNEL>>

returns its first argument, after making the two CHANNELs "know about each other" so that RUBOUT, ^@, ^L, etc., will work correctly between them.

10. Locatives

10.1. General

There is in MUDDLE a facility for obtaining and working directly with objects which roughly correspond to "pointers" in assembly language or "lvals" in BCPL or PAL. In MUDDLE, these are generically known as locatives (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 always refer to positions in structures. It is not possible to obtain a locative to something (e.g., an ATOM) which is not part of any structure.

It is possible to obtain a locative to any position in any structured object in MUDDLE — even the LVALS and GVALs of ATOMs, a structuring which is normally "hidden".

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

10.2. Obtaining Locatives

10.2.1. LLOC

<LLOC <an ATOM>>

returns a locative (TYPE LOCD) to the LVAL of <an ATOM>. If <an ATOM> has no LVAL, error. The locative returned by LLOC is independent of future re-bindings of <an ATOM>. I.e., IN (see below) of that locative will return the same thing even if the <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 a LVAL which has become unbound will fetch up an error. (It breaks just like a TUPLE . . ..)

10.2.2. GLOC

<GLOC <an ATOM>>

returns a locative (TYPE LOCD) to the GVAL of <an ATOM>. If <an ATOM> has no GVAL, error.

10.2.3. AT

<AT <s.o.> <TYPE FIX>>

where <s.o.> is any structured object, returns a locative to the <TYPE FIX>th position in <s.o.>. The exact TYPE of the locative returned depends on the TYPE of <s.o.>; e.g., LOCL for LIST, LOCV for VECTOR, etc. If <TYPE FIX> is greater than <LENGTH <s.o.>> or less than 1, error.

If the second argument — <TYPE FIX> — is not given, 1 is used.

10.3. 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 returnd by LLOC, GLOC, or AT.

10.3.1. IN

<IN <locative>>

returns the object <locative> points to. 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 TUPLE as mentioned in Chapter 7.

10.3.1.1. IN Examples

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

10.3.2. SETLOC

<SETLOC <locative> <anything>>

returns <anything>, after having made <anything> the contents of that position in a structure pointed to by <locative>. The structure itself is not otherwise disturbed. Error if IN wouldn't work on <locative> or if you try to put the wrong TYPE.

into a UVECTOR.

10.3.2.1. SETLOC Examples

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

10.4. Note on locatives

You may have noticed that locatives are, strictly speaking, unnecessary; you can do everything locatives allow by appropriate use of, e.g., 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 piece of code 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 bind ng can save a lot of fooling around with EVAL and ENVIRONMENTs.

11. Association

There is an associative data storage and retrieval system embedded in MUDDLE which is similar to, but less general than, that of LEAP (or SAIL). It is used via the four SUBRs described below.

11.1. Associative storage

11.1.1. PUTPROP

<PUTPROP <object> <indicator> <value>>

returns <object>, having associated <value> with <object> under the indicator <indicator>.

11.1.2. PUT

<PUT <object> <indicator> <value>>

returns <object>, after having done the following:

If <object> was structured and <indicacr> was of TYPE FIX, it does <SETLOC <AT <object> <indicator>> <value>>.

Otherwise, it acts like PUTPROP.

11.1.3. Removing Associations

If PUTPROP is used without its <value> argument, it removes any association existing between its <object> 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, may be used in the same way.

If either <object> or <indicator> cease to exist (i.e., no one was pointing to them, so they were garbage collected) then the association between them ceases to exist (is garbage collected).

11.2. Associative Retrieval

11.2.1. GETPROP

<GETPROP <object> <indicator> <exp>

if there is a <value> associated with <object> under <indicator>, returns that <value>. If there is no such association, returns EVAL of <exp>.

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

NOTE: <object> and <indicator> in GETPROP must be the same MUDDLE objects used to establish the association; i.e., they must be ==? to the objects used by PUTPROP or PUT.

11.2.2. GET

<GET <object> <indicator> <exp>

is the inverse of PUT, using NTH or GETPROP depending on the test outlined in 11.1.2.. <exp> is optional and used as in GETPROP.

11.3. Examples of Association

```
<SET L (1 2 3 4)>$
(1 2 3 4)
<SET N 0>$
0
<PUT L FOO "L is a list.">$
"L is a list."
.I$
(1 2 3 4)
<GET L FOO>$
"L is a list."
<PUTPROP .L 3 ![4]>$
![4!]
<GETPROP .L 3>$
![4!]
<GET .L 3>$
3
<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>>:

```
<PUT <REST .L 2> PERCENT 0.3>$
0.300000
<GET <2.L> PERCENT>$
#FALSE()
<GET <REST .L 2> PERCENT>$
0.300000
```

Remember comments?

```
<SET N [A B C;"third element"D E]>$
[A B C D E]
<GET <REST .N 3> COMMENT>$
"third element."
```

12. Lexical Blocking

Lexical, or static, blocking is another means of preventing identifier collisions in MUDDLE. (The first was dynamic blocking.) By using a subset of the MUDDLE lexical blocking facilities, the "block structure" of such languages as ALGOL, PL/1, SAIL, etc., can be simulated, should you wish to do so. (Write your full implementation of ALGOL 68 in MUDDLE!)

12.1. Basic Considerations

Since what follows appears to be rather complex, a short discussion of the basic problem lexical blocking solves and MUDDLE'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 ATOMs to all 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 Joe Hacker's library program?

Dynamic blocking (pushdown of LVALs) solves many such problems. However, there are some it does not solve — such as state variables (whether impure or pure). Major problems with a system having only dynamic blocking usually only arise when

attempts are made to share large numbers of significant programs among many people.

The solution used in MUDDLE 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 he wants.

Such a table is a MUDDLE object of TYPE OBLIST. All the complication which follows arises out of a desire to provide a powerful, easily used method of working with OBLISTs, with good defaults.

12.2. OBLISTs

An OBLIST is actually a UVECTOR of 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.

12.2.1. 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 <an OBLIST> OBLIST>

or

<GET <an OBLIST> OBLIST>

returns the name of <an OBLIST>.

Similarly, every name of an OBLIST is associated with its OBLIST, again under the indicator OBLIST, so that

<GETPROP <an OBLIST name> OBLIST>

or

<GET <an OBLIST name> OBLIST>

returns the OBLIST whose name is <an 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, 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.

12.2.2. MOBLIST

<MOBLIST <ATOM> <FIX>>

creates and returns a new OBLIST, containing no ATOMs, whose name is <ATOM>. <FIX> is the size of the OBLIST created — the number of hashing buckets. <FIX> is optional, with default 151, which is far more than you usually need or should use. If specified, <FIX> should be a prime number, since the hashing works better then.

12.2.3. OBLIST?

<OBLIST? <ATOM>>

returns #FALSE () if <ATOM> is not on any OBLIST. If <ATOM> is on an OBLIST, it returns that OBLIST.

12.3. READ and OBLISTs

12.3.1. Trailers

READ can be explicitly told to look an ATOM up on a particular OBLIST by giving the ATOM a trailer. A trailer consists of the characters !- (exclamation point dash) following the ATOM, immediately followed by the name of the OBLIST.

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 no separators (like space, tab, carriage-return, etc.). There is a default name (see below) 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.

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.

Defaults very often make trailers unnecessary. See below.

12.3.2. READ and Defaults

If trailer notation is not used (the "normal" case), READ looks up the PNAME of the ATOM in a LIST of OBLISTs, specifically 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 inserts it into <1 .OBLIST>.

12.4. 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. I.e.: if the ATOM is the first ATOM of that PNAME which READ would find in its normal lookup on the current LIST of OBLISTs, no trailer is output. If it is not, !- is put out and the NAME of the OBLIST is recursively PRINTed.

Warning: there is an obscure case, which does not occur in normal practice, for which the PRINT trailer recursion does not terminate. If an ATOM must have a trailer printed, and the name of the OBLIST is an ATOM in that very same OBLIST, death.

12.5. Initial State

Upon starting a MUDDLE, .OBLIST contains 2 OBLISTs. <1 .OBLIST> initially contains no ATOMs, and <2 .OBLIST> contains all the ATOMs whose GVALs are SUBRs or FSUBRs. It is difficult to lose track of <2 .OBLIST>; 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 fact that !-<space> is used.

The name of the initial <1 .OBLIST> is INITIAL (really INITIAL!-).

An error restores .OBLIST, in the sense that the initial OBLISTs it contained are now its members again, in their initial order. However, any changes that were made to those OBLISTs — e.g., new ATOMs added — remain. <ERRET> does the same thing.

One other OBLIST exists in a virgin MUDDLE: an OBLIST whose name is ERRORS!- . This OBLIST contains ATOMs whose PNAMEs are used as error messages. It is not initially on .OBLIST, so errors usually cause a lot of !-ERRORS trailers to be printed.

12.6. 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 OBLIST and making its argument the current LVAL. You usually want <ROOT> to be a member of <LIST of OBLISTs>, normally its last.

<ENDBLOCK>

"pops" the LVAL of OBLIST and returns the resultant OBLIST.

12.7. SUBRs Associated With Lexical Blocking

12.7.1. READ (again)

<READ <error routine> <CHANNEL> <LIST of OBLISTs>>

This is the full configuration of READ. <LIST of OBLISTs> is used as stated in 12.3. to look up ATOMs and insert them in OBLISTs. If it is not specified, .OBLIST is used.

12.7.2. LOOKUP

<LOOKUP <STRING> <OBLIST>>

If an ATOM of PNAME <STRING> is in <OBLIST>, returns that ATOM; otherwise returns #FALSE().

12.7.3. 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().

12.7.4. INTERN

<INTERN <ATOM> <OBLIST>>

puts <ATOM> into <OBLIST> and returns it. If there is already an ATOM with the same PNAME as <ATOM> in OBLIST, error.

12.7.5. 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 ().

12.7.6. PNAME

<PNAME <ATOM>>

returns a STRING (not unique) which is <ATOM>'s PNAME.

12.8. Example of Normal Use: Death of the INC Problem

On the following page 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.

```
<MOBLIST INCO 1>
;"Create an OBLIST to hold your external symbols."

INC!-INCO
;"Put your external symbols into that OBLIST."

<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 notin
any OBLIST of the BLOCK; therefore, trailer notation of !-INCO
will not work."

<SETG INC                ;"INC is found in the INCO OBLIST."
      <FUNCTION (A)      ;"A is not found, and is therefore put
                          into INCI by READ."
         <SET .A <+..A 1>>>>

<ENDBLOCK>
```

This example is rather trivial, but it contains all the issues, of which there are three:

The idea is that you should create two OBLISTs, one to hold ATOMs which you wish users to know of (INCO), and the other to hold internal ATOMs which are not normally of interest to the

user (INCI). The case above has one ATOM in each category.

INCO is explicitly used without trailers so that externally used 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, following the ENDBLOCK with

```
<REMOVE "INCI" <GET INCO OBLIST>>
```

will 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" — i.e., be able to type it in — by saying something of the form

```
<INTERN <1<1 ,INC!-INCO>> <1 .OBLIST>>
```

thereby grabbing A out of the structure of INC and re-inserting it into an OBLIST.

12.9. Extensions

There are some extensions to the basic lexical blocking machinery which are planned. Their intent is to facilitate the use and dynamic loading of "packages" of routines. These descriptions should be taken as tentative.

12.9.1. The User Oblist Oblist (UOO)

The initial .OBLIST will eventually contain three OBLISTs: INITIAL, an OBLIST named UOO!- , and ROOT in that order. UOO (from User Oblist Oblist) is intended to be a "root" for trees of user-defined OBLISTs, similar to the MULTICS udd for directories. It is not possible to actually enforce this use of UOO; however, using it as a "root" will be convenient, due to the next feature.

12.9.2. Automatic OBLIST Generation

Suppose trailer notation is used, and an ATOM in the trailer is not an OBLIST name. Eventually, this will cause READ to generate an OBLIST of that name, place the name in UOO, and place the originally trailered ATOM into the new OBLIST. If the routine or package defining the original ATOM is now loaded later, it need only look in UOO to resolve previous references.

13. Errors, FRAMEs, etc.

13.1. LISTEN

This SUBR takes any number of aguments. It PRINTs them, then PRINTS

LISTENING-AT-LEVEL <M> PROCESS <N>

where <M> is an integer (FIX) which is incremented each time LISTEN is called recursively, and <N> is an integer identifying the process in which the LISTEN was EVALed. LISTEN then drops into an infinite READ-EVAL-PRINT loop which may be left via ERRET (see below).

13.2. ERROR

This SUBR is identical to LISTEN, except that it first PRINTs *ERROR*.

When any SUBR or FSUBR detects an anomolous condition, (e.g., its arguments are of the wrong TYPE) it calls ERROR with

two arguments:

1) an ATOM whose PNAME describes the problem

2) the ATOM whose VALUE the SUBR or FSUBR is and then returns whatever the call to ERROR returns.

13.3. TYPE FRAME

A FRAME is the object placed on a process' stack whenever a SUBR or FSUBR is applied. It contains information describing what was applied, plus an object of TYPE ARGUMENTS. The latter is a TUPLE-like object, often referred to as the ARGUMENT TUPLE, whose elements are the arguments to the SUBR or FSUBR applied. If a SUBR is applied, the ARGUMENTS of its FRAME have been EVAL'd by the time the FRAME is generated.

A FRAME is an anomolous TYPE in the following ways:

1) It cannot be typed in. It can only be generated by an application.

2) It does not type out in any standard format, but rather as

STACK-FRAME-FOR-

followed by the PNAME of the SUBR or FSUBR applied.

13.3.1. ARGS

<ARGS <a FRAME>>

Returns the ARGUMENT TUPLE of <a FRAME>.

13.3.2. FUNCT

<FUNCT <a FRAME>>

returns the ATOM whose VALUE is being applied in <a FRAME>.

13.3.3. FRAME (the SUBR)

<FRAME <a FRAME>>

returns the FRAME stacked before <a FRAME>. If called with no arguments, FRAME returns the topmost FRAME used in an application of ERROR.

13.3.4. Examples

Say you have gotten an ERROR. You may now type at ERROR's LISTEN loop and get things EVALd. E.g.,

```
<FUNCT <FRAME>>$
ERROR
<FUNCT <FRAME <FRAME>>>$
<the ATOM whose VALUE is the (F)SUBR
  which called ERROR>
<ARGS <FRAME<FRAME>>>$
<the arguments of the (F)SUBR which
  called ERROR>
```

13.4. ERRET

<ERRET <anything> <a FRAME>

This SUBR

1) causes the stack to be stripped down to the level of <a FRAME>.
2) Then returns <anything>.

The net result is that the application which generated <a FRAME> is forced to return <anything>.

The second argument to ERRET is optional, with default <FRAME>.

If ERRET is called with no arguments, it drops you all the way down to the bottom of the stack — the level 1 LISTEN loop.

13.4.1. Examples

```
<* 3 <+ a 1>>$
*ERROR*
FIRST-ARG-WRONG-TYPE
+
LISTEN-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. E.g.,

```
<SETG F
      <FUNCTION (A "AUX" (B "a string"))
                (.B <REST .A 2>) ;"Return this LIST."
  >>$
- - - -
<F  (1)>$
*ERROR*
OUT-OF-BOUNDS
REST
LISTENING-AT-LEVEL 2 PROCESS 1
<ARGS <FRAME <FRAME>>>$
[(1) 1]
.A$
(1)
.B$
"a string"
<ERRET (5)>  ; "Make the REST return (5)"$
("a string" (5))
```

13.5. Control-G (^G)

Typing Control-G (^G) at MUDDLE causes it to act just as if an error had occurred in whatever was currently being done. You can then examine the contents of variables as above, continue by applying ERRET to one argument (which is ignored), or flush

everything by applying ERRET to no arguments.

14. Other Things

The following don't seem to fit very well into any sectioning I can come up with. So here they are.

14.1. STACKFORM

This rather strange FSUBR is used to build a FORM on the STACK (more efficient than garbage-collected storage) and then EVAL it. An application of STACKFORM looks like this:

<STACKFORM <ap> <e> <ce>>

where

<e> is an arbitrary expression

<ap> EVALs to something which can be applied (SUBR, FSUBR, etc.)

<ce> is another arbitrary expression which should be capable of returning a FALSE.

Evaluation of an application of a STACKFORM proceeds as follows:

1) Evaluate <ap> and place the result on the stack.

2) Evaluate <ce>, and then:

2.1) If <ce> evaluated to non-FALSE, evaluate <e> and place the result on the stack. Then go back to the start of (2).

2.2) If <ce> evaluated to a FALSE, apply the stacked <ap> to the stacked <e>s and return the result.

14.1.1. Example

The following SUBR reads characters from .INCHAN until an ⌀ is read, and then returns what was read as one STRING.

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

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

14.2. % and %%

The tokens % and %% are interpreted by READ in such a way as to give a "macro" capability to MUDDLE similar to PL/1'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 %. I.e., % means "don't really READ this, use EVAL of it instead."

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.

14.2.1. Example

```
<SETG SETUP <FUNCTION () <SET A 0>>>$
- - - -
<SETG NXT <FUNCTION () <SET A <+ .A 1>>>>$
- - - -
[%%<SETUP> %<NXT> %<NXT> (%%<SETUP>) %<NXT>]$
[1 2 () 1]
```