The MDL Programming Environment

P. David Lebling

May, 1980

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

Table of Contents

INTRODUCTION

The MDL language is described in "The MDL Programming Language" [3], but in addition to the language itself, there is a rich and varied collection of software written in the language which facilitates the writing of programs and systems of programs in MDL. The information describing this programming environment has been contained in various documents, some out of print or out of date, and in supplemental disk files describing changes and additions. Some of the packages of functions used to deal with MDL code have never been formally documented. This manual brings together some of that scattered documentation.

The document's purpose is to flesh out the description of the language contained in "The MDL Programming Language," giving a fuller description of the program writing and debugging aids available to MDL users, to describe the methods for producing code usable by others, to describe the MDL compiler and the many other techniques for producing and speeding up MDL object code.

The imagined reader of this document is someone who has read "The MDL Programming Language," and now proposes to write programs in MDL, possibly even very large programs. MDL packages that he would find useful in the *process* of doing so are documented here: editors, debuggers, etc. Packages that he might wish to use *within* his program are not included: data-management systems, command interpreters, etc.

This document is of necessity highly self-referent, as many of the components of the MDL programming environment refer to each other and adhere to the same conventions. Additionally, this document assumes that the reader is familiar with the language itself (at least to some degree) and with the ITS, TENEX, or TOPS-20 operating systems.

ACKNOWLEDGMENTS

ACKNOWLEDGMENTS

The programs described in this document are the products of many man-years of effort by many people. Most have been 'touched' by several programmers, added to and improved over the years.

Some of the people responsible for the programs mentioned in this document are: Chris Reeve (MDL, the compiler, GLUE); Brian Berkowitz (MDL, the compiler, TEMPLATE, SUBRFY); Bruce Daniels (MDL, the compiler, PACKAGE, PPRINT, DEBUGR, ASSEM); Tim Anderson (PACKAGE, the Library, FINDATOM, DFL, COMBAT, MUDINQ); Neal Ryan (EDIT, PDUMP, the IPC interface); Marc Blank (MAT, MUDCOM, MONITR, COMBAT, EDIT, CURSOR); David Lebling (CRITIC, EDIT); Michael Broos (the Library); Roger Banks (TRACE); Greg Pfister (PPRINT); Joel Berez (EDIT).

(Most of the documentation subsumed in this manual is from published and unpublished memos of the Programming Technology Division of the M.I.T. Laboratory for Computer Science. As a general rule, updates and revisions to this and other PTD documents concerning MDL are available online in the directory "MUDMAN" at MIT-DMS).

NOTATION

NOTATION

Anything which is written in the MDL language or which is typed on a computer console appears herein in a typewriter font, as in `PPRINT`. A metasyntactic variable -- something to be replaced in actual use by something else -- appears as *channel*, in an italic font. Where a meta-syntactic variable is being used to denote a required argument to some function, it appears as before, but underlined, as *channel*.

In the argument templates of MDL functions, the individual arguments are often given in the form *argument:type*, where *argument* is a 'descriptive' name for the argument, and *type* is its MDL type (or range of types). In such cases, the 'type' *boolean* indicates an argument that is only examined for truth or falsity, and not for any of its other qualities. Such arguments in MDL are often declared '`<OR ATOM FALSE>`'.

Finally, file names are given as though for the ITS operating system:

device:*sname*;*fnm1 fnm2*

The analogous specification for TENEX or TOPS-20 would be

device:<*sname*>*fnm1*.*fnm2*

Note that in the TENEX/TOPS-20 version of MDL, the *fnm2* (which may include the *generation* number, *protection* and *account* fields) is by default `"MUD"` as opposed to `">"` for the ITS version.

1. Overview of the MDL Programming Environment

The parts of the MDL programming environment described in this document are primarily those dealing with the writing, debugging, sharing, and maintenance of code and programs written in MDL. Most of the packages described herein are written in MDL themselves; some are assembly language programs useful to MDL programmers.

The document is divided into chapters dealing with the major issues facing the novice (or even the experienced) MDL programmer.

- 'The Package System' introduces the standard mechanism for lexical blocking and therefore, sharing of MDL code. Understanding its use is fundamental to writing MDL programs.

- 'Program Writing and Debugging Aids' is the largest chapter. It covers mechanisms for loading, dumping, editing, and debugging MDL code, whether interpreted or compiled, in a development or a production environment.

- 'The Library System' discusses the usage of libraries of MDL programs.

- 'The Compiler' includes the specifics of interaction with the MDL compiler, as well as an overview of the theory behind its operation.

- 'Making It Run Faster' covers the various methods for speeding up 'production' MDL code by removing mediated calls and compacting data structures.

- 'The Assembler' documents the MDL assembler and some methods of debugging binary code.

- 'Informational Aids' discusses a few programs, most written in assembly language rather than MDL, which are useful to the MDL programmer.

2. The Package System

The portion of the MDL environment which provides a uniform facility for lexical blocking is known as the Package System. In one sense it is the most basic part of the environment, since it enables many programmers to use each other's code without identifier conflicts.

In addition, the Package System is interfaced to a library facility (see section 4) by which MDL code may be stored and later loaded as needed.

The Package System is so basic to use of the MDL environment that (with a few exceptions) every subsystem or family of MDL functions described in this document is a 'package'.

2.1. The Theory of Lexical Blocking in MDL

Lexical blocking is implemented in MDL by means of OBLISTs and LISTs of OBLISTs. Changes of lexical context are performed using the SUBRs BLOCK and ENDBLOCK. The Package System provides a high-level interface to these low-level constructs.

The primary goal of a lexical blocking scheme is the prevention of identifier conflicts. Specifically, when your program references the variable X, it should be your X and not that of some other program. At the same time, it should not be necessary for a programmer to search every program previously written to verify that an identifier he wishes to use is not already 'taken'.

It should be clear that the simplest solution, a single OBLIST, will not satisfy either of these goals. With only one OBLIST there would necessarily be identifier conflicts, necessitating exhaustive searching for unique identifiers.

Obviously, programmers could put their program's identifiers on an OBLIST unique to that program. Unfortunately, such a solution addresses only half the problem. What happens when some other programmer wishes to use some of this code? He could insert the unique OBLIST for that program into the OBLIST path for his program; but the moment that is done he gets all the identifiers for that program, including local variables, internal data structures, and so on.

Consequently, we move to a situation where each program uses two OBLISTs: one for the identifiers that are local to the program, and one for the identifiers that are to be used by other programs. In the Package System, these are known as the 'internal' OBLIST and the 'entry' OBLIST.

Most of the identifiers in a program are local to it, and want to be placed on the internal OBLIST.

Therefore, in terms of an argument to the BLOCK SUBR, when a program is being loaded into MDL, the OBLIST path wants to be:

```
( internal-oblist
  entry-oblist
  <ROOT> )
```

With this OBLIST path, most ATOMs (identifiers) will be on the internal OBLIST (as READ puts unknown identifiers on <1 .OBLIST>), but the ATOMs for the entries and the ATOMs for the usual SUBRs will be available.

The only issue yet to be addressed is that of using an entry of a different program in your program. This is accomplished by adding the entry OBLISTs of any such programs to the path after ROOT:

```
( internal-oblist
  entry-oblist
  <ROOT>
  other-program-entry-oblist
  yet-another-program-entry-oblist
        .
        .
        .  )
```

As only the entry OBLIST, and not the internal OBLIST, of the program being used is added to the path, the chance of identifier conflict is lessened.

All that remains is to introduce the functions by which these various operations are performed.

2.2. Package System Overview

The functions which make up the Package System are:

- PACKAGE. This indicates the start of a package of functions.
- ENDPACKAGE. This indicates the end of the package of functions.
- ENTRY. This indicates an ATOM which is to be made available outside the definition of this package of functions. All other ATOMs will not be directly available outside the package.
- USE. This indicates a reference by name to another package of functions.
- USE-DATUM. This indicates a reference by name to a data set.
- DROP and L-UNUSE. These undo the effects of USE and USE-DATUM.

These functions are themselves part of a package named "PKG", which is preloaded into MDL.

2.2.1. Sample PACKAGE

A sample MDL PACKAGE is given with comments in order to demonstrate the usage of these functions.

```
<PACKAGE "HOUR-STRING">

;"PACKAGE begins the package called HOUR-STRING."

<ENTRY TIME-STRING>

;"The atom TIME-STRING is an entry to this package;
 it may be referenced by other packages by
 USEing HOUR-STRING."

<USE "DATIME">

;"Indicate that the package DATIME is
  used within the current package."

<DEFINE TIME-STRING ()
        <STRING <UNPARSE <HOURS>> " o'clock">>

;"Define this little function which returns a string
 telling the last hour in a strange format."

<DEFINE HOURS () <1 <RTIME>>>

;"Define an internal function which is available
 only within the HOUR-STRING package, since its
 name is not in any ENTRY statement.
 Note that this function refers to RTIME,
 which is an ENTRY in the DATIME package."

<ENDPACKAGE>

;"The end of this little demonstration package."
```

2.3. PACKAGE

This function delimits the beginning of a package of functions. It takes one required argument, a STRING, which is the name of the package. This STRING uniquely identifies the package within a library of packages (see section 4).

In a PACKAGE those ATOMs which are specified as entries live in a separate OBLIST of their own, called the entry OBLIST. The ATOM naming this OBLIST is on the PACKAGE OBLIST and has the same name as the PACKAGE itself. Thus, an entry 'X' of a PACKAGE 'Y' would have as its 'full-trailer' name: X!-Y!-PACKAGE!- .

PACKAGE blocks (sets up) the current OBLIST path so that the ATOMs which are internal to the PACKAGE

fall into an OBLIST which is not otherwise used. The ATOM naming this OBLIST is on the entry OBLIST of the PACKAGE, and is by default given a name created by putting the character 'I' at the beginning of the PACKAGE's name. An internal ATOM 'Z' in the PACKAGE 'Y' previously mentioned would have as its 'full-trailer' name: Z!-IY!-Y!-PACKAGE!- .

PACKAGE also keeps track of the fact that the particular PACKAGE named has been defined in this MDL process, by putting its name on the PACKAGE OBLIST.

```
<PACKAGE name:string
         iname:string
         size:fix
         isize:fix>
```

PACKAGE takes three optional arguments in addition to the required one (the optional arguments are ignored if *name* is already a PACKAGE):

iname is the name of the internal OBLIST of the PACKAGE; by default it is the name of the PACKAGE with the letter 'I' prefixed.

size is the number of buckets in the entry oblist; by default 19.

isize is the number of buckets in the internal oblist; by default 23.

In addition to PACKAGE, there exists the obsolete function RPACKAGE, documented here only because some programs still use it. The difference between them is that the entry OBLIST for an RPACKAGE is the ROOT OBLIST. The implication of inserting an entry into the ROOT is that this requires that the name of the entry be unique over all PACKAGEs, because the entry is, in effect, being promoted to the status of a SUBR. It is (in rare cases) useful to do this, but the correct way is with the function RENTRY (see section 2.3.1).

2.3.1. ENTRY

The ENTRY function applied to one or more ATOMs declares that these ATOMs are to be put into the OBLIST reserved for entries in this particular PACKAGE. Only ATOMs declared in this way will be accessible (in the normal course of events) to functions outside this PACKAGE.

It is possible to place some entries of a PACKAGE on the ROOT OBLIST using the function RENTRY. It is recommended that instead of using RPACKAGE in those rare cases where entries must go on the ROOT, RENTRY be used instead.

All ENTRY statements should appear immediately after the PACKAGE or RPACKAGE statement. Note: never put a USE statement before the ENTRY statements; if you do, you may get the ERROR message

ALREADY-USED-ELSEWHERE, meaning that the name of an entry is conflicting with an ENTRY in one of the PACKAGEs you USEd. ENTRY will also give an ERROR if it is used outside the body of a PACKAGE.

2.3.2. USE

This function takes as arguments one or more STRINGs which are the names (as given to PACKAGE) of other PACKAGEs. EXTERNAL is a synonym of USE. USE causes the entry OBLISTs of the PACKAGEs named to be spliced into the current OBLIST path. Thus, references to entries of those PACKAGEs may be made after the USE, until the next ENDPACKAGE (or the next DROP or L-UNUSE if USE is being invoked outside a PACKAGE to load a file).

USE is consequently the mechanism for sharing code. If the PACKAGE being used is already loaded, its entries are made available; if not, the PACKAGE is loaded first (see section 4.1 for details on how this is accomplished).

2.3.3. USE-DATUM

USE-DATUM requires one STRING argument, the name of a data set. If the data set is not loaded, USE-DATUM loads it and creates an ATOM of the same name, on the USE-DATUM OBLIST, whose GVAL is the data set. USE-DATUM always EVALs to the data set named, regardless of whether it had to be loaded or not.

2.3.4. DROP and L-UNUSE

These functions take the same arguments as USE and USE-DATUM and undo their effects.

DROP simply splices the named PACKAGEs out of the current OBLIST path. A USE of a DROPped PACKAGE will not reload the PACKAGE but simply splice it back into the OBLIST path.

L-UNUSE splices the PACKAGE out and removes its name from the PACKAGE OBLIST, which will cause the entire PACKAGE to be reloaded if it is USEd again. L-UNUSE of a data set will remove its ATOM from the USE-DATUM OBLIST.

2.3.5. ENDPACKAGE

The ENDPACKAGE function of no arguments terminates the definition of the current PACKAGE and undoes the lexical blocking done by the PACKAGE function. The ENDPACKAGE statement should be the last one in the file.

2.3.6. PACKAGE Restrictions

There are some restrictions on what the user may do inside a PACKAGE. These are enforced by the Library System when the user attempts to submit a PACKAGE to a library.

A PACKAGE should not FLOAD or LOAD any file to obtain parts of itself. All such environment setup should be done with USE and USE-DATUM.

A PACKAGE may not reference any ATOM whose OBLIST path goes through the INITIAL OBLIST. All of a PACKAGE's non-entry ATOMs should fall naturally into the PACKAGE's internal OBLIST.

As mentioned before, the RENTRYs of a PACKAGE have the same OBLIST status as SUBRs, i.e., they must be unique among both all SUBRs and all PACKAGE entries.

2.3.7. ENTRY Name Conflicts

It is possible to have two or more PACKAGEs (not RPACKAGEs) which have entries (not RENTRYs) with the same PNAME. If the user needs both PACKAGEs at the same time, he may USE them both and refer to the ambiguous entries by their 'full trailer' names. All of the non-ambiguous entries in both PACKAGEs may still be referenced by PNAME only.

3. Program Writing and Debugging Aids

This chapter concentrates on editing and debugging aids for MDL programming. The basis for editing and debugging in MDL is twofold: First, MDL is an interpreter, which permits interactive testing and debugging of software. Secondly, MDL programs (even compiled MDL programs) are structures and therefore may be manipulated by other MDL programs.

Packages useful in editing and debugging range from EDIT and PPRINT, which are preloaded, and which form the core of most editing or debugging systems, to more sophisticated aids such as DEBUGR and TRACE, which are more powerful, and useful for more complicated debugging.

It should be noted that, in addition to the editors discussed below, RMODE [5] and EMACS [2], TECO based text editors, understand much of the syntax and many of the conventions of MDL programs.

3.1. Pretty-Printing

The purpose of pretty printing is to clarify the structure of MDL objects by printing them in a more human-readable format than that provided by the SUBRs PRINT, PRIN1, etc. Objects are pretty-printed through the judicious insertion of spaces, tabs, and new-lines between tokens. Pretty-printed objects are readable by the MDL Reader. Pretty printing is an aid to understanding and debugging MDL FUNCTIONs or other objects. You will probably find pretty printing to be extremely helpful, especially if you are working without a listing or with an old listing. In fact, pretty-printing is one way to make a new pretty listing after editing. PPRINT is pre-loaded in most initial MDLs. The name of the package containing PPRINT is "PP".

<PPRINT *any channel*>

pretty-prints *any* on *channel*. The second argument is optional, by default .OUTCHAN . If *any* is an ATOM, PPRINT will enclose it in an application of DEFINE, DEFMAC, SETG, or SET, as seems appropriate. COMMENTs found inside *any* are right-justified. PPRINT cannot output an RSUBR without FIXUPs (that is, one that was READ in while KEEP-FIXUPS (see section 3.4) had no LVAL or had a FALSE LVAL); it will give the ERROR message CAN-NOT-BE-DUMPED. PPRINT returns ,NULL, which is an ATOM whose PNAME is a single rubout, invisible on normal consoles.

<PPRINF *in:string-or-atom-or-list outfile:string*
width:fix eval?:boolean>

pretty-prints all the contents of *in* into *outfile*.

If *in* is an ATOM or a LIST of ATOMs, its VALUE(s) are the objects to be PPRINTed. In this case, *outfile* is by default a file whose first name is produced by taking the PNAME of *in* (or *in*'s first element, if *in* is a LIST).

If *in* is a STRING, if specifies a file containing objects to PPRINT. In this case, *outfile* is by default "TPL:".

width is the maximum width of output lines (although output lines are prevented from being extremely long); it is optional, by default <13 ,OUTCHAN>.

eval? tells PPRINF whether or not to EVAL everything in the file; it is optional, by default a FALSE (don't EVAL). *eval?* is meaningless if *in* is not a STRING.

PPRINF returns either "DONE" or a FALSE if it couldn't open *infile* or *outfile*. PPRINF inserts page boundaries in *outfile*, between objects, every 60 lines or fewer; you may want to move these afterward to more logical places. PPRINF binds KEEP-FIXUPS and REDEFINE to T, and QUICKPRINT (see below) to a FALSE.

3.1.1. PPRINT Control Switches

PPRINT's output is affected by the local values of several ATOMs. Each value is examined only for truth.

.QUICKPRINT

If this ATOM's LVAL is a FALSE, you are in slow mode; otherwise (including the case of no LVAL), you are in fast mode. The behavioral difference is this: in fast mode, there may be COMMENTs in the pretty-printed object(s) which PPRINT misses. Also, fast mode is indeed faster than slow mode. Fast mode is the default, that is, QUICKPRINT is initially true. The modes are really distinguished by the depth of recursion to which PPRINT resorts. In slow mode, it recurses all the way down to every monad in the thing pretty-printed; in fast mode, it goes down only far enough to find something that will fit on a line.

.LOOKAHEAD

PPRINT uses full recursive lookahead to avoid packing things against the right margin and, as a result, not being able to fit things within the right margin. The lookahead results in very good formatting of deeply-nested MAPFed and FUNCTIONs; all but the most bizarre cases should be very legible. However, it can result in noticeable 'pauses' in the printing operation and, in some cases, a net speed slightly less than with limited lookahead. Since this can be a disadvantage when using PPRINT interactively on a heavily-loaded system, the lookahead can be disabled: if the LVAL of LOOKAHEAD is a FALSE, no lookahead will be performed; otherwise it happens. LOOKAHEAD is initially true, that is, lookahead happens by default.

.VERTICAL

If LOOKAHEAD is a FALSE, the formatting can cause too many objects to be squeezed against the right margin. So that particular cases can be made legible, the format when lookahead is not in use can be manually set: if the LVAL of VERTICAL is non-FALSE, PPRINT will indent very little whenever indenting is

called for. (VERTICAL being true means a 'more vertical' format.) VERTICAL is initially FALSE. The value of VERTICAL is ignored when LOOKAHEAD is true; the lookahead effectively chooses different values for VERTICAL for different parts of the object pretty-printed.

3.1.2. Lower-level Pretty Printing

It is sometimes desirable to use some of the functions that PPRINT uses, but in a different way. For example, a specialized pretty-printer for Program Abstracts would want to insert indented field names into the output and pretty-print field values with the same indentation. The names of lower-level pretty-print functions are included in the ROOT OBLIST for such purposes.

<EPRINT *any left-margin:fix*>

pretty-prints *any* on .OUTCHAN to the right of *left-margin*. The second argument is optional, by default <VALUE LEFT-MARGIN> (see below).

<EPRIN1 *any left-margin:fix*>

EPRIN1 is to EPRINT as PRIN1 is to PRINT.

.LEFT-MARGIN

This is the ATOM that EPRINT binds to its second argument. You can SET it outside calls to EPRINT in order to make a permanent left margin. Its initial LVAL is 0.

<INDENT-TO *column:fix channel*>

outputs tabs and/or spaces to advance the output column (<14 *channel*>) to *column*, if it is not already past.

<COLPP *any*
channel
left-margin:fix
right-margin:fix>

pretty-prints *any* on *channel* (by default .OUTCHAN) between the margins *left-margin* (by default <14 *channel*>, the current column) and *right-margin* (by default <13 *channel*>, the rightmost column). All arguments but the first are optional. COLPP returns ,NULL. For example, <COLPP *any* .OUTCHAN 10 70> would leave a 10-character margin at left and right on an 80-column OUTCHAN. Also,

```
<PROG () <PRINT AAAAAAAAAAAAAAA> <COLPP ,FOO>>
```

would result in output like

```
AAAAAAAAAAAAAAA #FUNCTION ((X GGGGGGGGGGGGGGGGGGGGG)
                           <+ X 1>)
```

EPRINT, EPRIN1, and COLPP are affected by the truth of .QUICKPRINT, .LOOKAHEAD, and .VERTICAL.

3.1.3. Ampersand Printing

'Ampersand printing' consists of printing any object on a single line by using the character & (ampersand) to mean 'There's more stuff here.' (This technique is borrowed from the InterLisp editor.)

There are two ways in which & is used by this printer as an abbreviation:

1. An & appearing between some variety of brackets indicates that there is a big object of the indicated TYPE there.

2. The characters . .& or &. . on the left or right of a structure mean that there are more objects to the left or right which have not been printed.

Examples:

```
#FUNCTION ((A B C D) <&>)
```

This is a FUNCTION with four arguments in its argument LIST, and the FUNCTION body contains one FORM which was too big to print in the remainder of the line.

```
<PROG () <KRK <+ .A 5>> <PRINC .Q> <SET BAR <ORG>> <&> &..>
```

This is a large FORM, namely, a PROG. In addition to the elements printed, there are more elements to the right, and there is one FORM which was too big to fit.

Ampersand printing is effected by two pure RSUBRs: &, analogous to PRINT, and &1, analogous to PRIN1. A related RSUBR, &LIS, can be applied to no arguments to put you into an endless READ-EVAL-& loop, instead of the normal READ-EVAL-PRINT loop.

3.1.4. Examining the Stack

<FRM *fix*>

returns the *fix*th FRAME down from the top application of ERROR or LISTEN.

<FRAMES *how-many:fix start:fix*>

pretty-prints *how-many* FRAMEs (by printing the FRAME number (suitable as an argument to FRM), FUNCT, and ARGS of the FRAME), starting with <FRM *start*>. Both arguments are optional; *start* defaults to 0, and *how-many* defaults to a large integer. A FRAME whose FUNCT is an ATOM whose VALUE is an FSUBR is not printed, if the same information is found in the next lower FRAME.

<FR& *how-many:fix start:fix*>

is like FRAMES but uses ampersand printing instead of pretty printing. It is handy for summarizing FUNCTs and ARGS that are large or unprintable (like RSUBRs with no fixups).

<FRATM *how-many:fix start:fix*>

is like FRAMES but gives an abbreviated view of the stack. It prints FUNCTs only, and only for FRAMEs connected with named FUNCTIONs, RSUBRs, and RSUBR-ENTRYs. It is handy when a FRAME contains a non-LEGAL? object.

<FRLVAL *atom*
how-many:fix
start:fix>

prints out the stacked bindings of *atom*, going through *how-many* FRAMEs, starting with <FRM *start*>. The two numeric arguments are optional; *how-many* defaults to a large integer, and *start* defaults to 0. The format of the printing is two columns: the first column is the number of the FRAME in which *atom* has a binding; the second column is the value bound, or a message proclaiming the lack of a value.

<FR&VAL *atom*
how-many:fix
start:fix>

is precisely the same as FRLVAL, except that the values are ampersand printed instead of PRINTed.

Finally, the "FRMSP" PACKAGE contains analogues of many of the preceding functions, but each takes as its first argument a PROCESS, by default <ME>. These are all named by adding a 'P' to the end of the usual name. For example,

<FR&P <MAIN>>

does a <FR&> in the PROCESS MAIN.

There is one additional function of interest in "FRMSP".

<FRTYPE *how-many:fix start:fix*>

is like FRAMES, but gives only the TYPEs of the arguments to each. This is useful in those situations when the stack shows illegal FRAMEs or other unprintable objects.

3.2. The MDL Editor

EDIT allows a MDL user to make incremental changes in MDL structured objects, without leaving MDL and with the ability to save the results in a file, and to set or clear conditional breakpoints of various sorts in objects that will be evaluated, such as FUNCTIONs.

EDIT is an editor/debugger written in, written for, and running under MDL. It comprises the package "EDIT" and several smaller packages which will be mentioned later in this section. EDIT is preloaded in most initial MDLs.

To start editing, apply EDIT to no arguments or to the name of the object you wish to edit: <EDIT>

causes entry into EDIT and opens the last object edited; <EDIT *object*> causes entry into EDIT and opens *object* for editing. Permissible *objects* include:

- ATOMs. The GVAL (preferably) or the LVAL of the ATOM is opened. If it has no value, EDIT returns a FALSE.

- A PRIMTYPE LIST. The PRIMTYPE LIST is opened.

- A FIX. The stack frame with that number is opened (i.e., <ARGS <FRM *fix*>>).

Part of EDIT's efficiency comes from forbidding it to delve into objects that are not of PRIMTYPE LIST, that is, not LISTs, FORMs, FUNCTIONs, etc. Attempts to edit objects of other PRIMTYPEs will result in error messages. These objects can, however, be treated as units when inserting, searching, etc.; or they can be changed into LISTs, edited, and then changed back to their original types.

3.2.1. The Edit 'LISTEN Loop'

3.2.1.1. The Reader

When in EDIT, you are typing at a special, non-standard, input function: The EDIT Reader.

The Reader allows you to type EDIT commands and have them executed, and also to evaluate MDL expressions normally. Its characteristics are as follows:

- As in the normal MDL Reader, nothing is done until you type ESC. DEL, ↑L, ↑D, ↑G, and ↑S also work normally.

- All EDIT commands are terminated when an ESC is encountered in the input stream. In addition, most commands will terminate whenever the maximum number of arguments required has been input or whenever an argument of the wrong type is encountered. In the former case the next object is taken as a new command; in the latter case the object of the wrong type is taken as a new command. EDIT commands may be typed in either upper or lower case.

- If you type something that EDIT does not recognize as a command, normal MDL evaluation and printing are performed on that something. This evaluation will have no effect on your position in the object you are editing.

- While editing a function which is part of a PACKAGE (determined from an examination of the OBLIST containing the ATOM whose value is the function), EDIT causes the OBLIST path to be set up to what it was in the environment of that PACKAGE. This has the advantage of reducing the number of trailers printed, and causes newly entered ATOMs to fall on the correct OBLIST (the internal OBLIST of the PACKAGE). It has the slight disadvantage that it disables the dynamic loader (which depends on unbound variables falling on the INITIAL OBLIST). If the GVAL of E-PKG is a FALSE, this feature is disabled, and the normal OBLIST path is in effect during

editing.

Examples:

```
R 5$
```

Causes execution of EDIT command R, with argument 5.

```
<R 5>$
```

Causes application of the function R to 5.

3.2.1.2. The Ampersand Printer

Your current position is displayed by 'ampersand printing' (see section 3.1.3). This consists of printing any object on a single line by using the character & (ampersand) to mean 'There's more stuff here.'

The ampersand printer used in EDIT is much like the standard one, with the addition that your current position (see below) is displayed by the glyph ▮.

When you initially enter EDIT, you are in a mode called 'non-verbose,' in which ampersand printing is not automatically done following execution of EDIT commands. The V command is used to toggle you in and out of verbose mode (see below).

Examples:

```
#FUNCTION ( ▮ (A B C D) <&>>
```

Indicates that your position is just to the left of a FUNCTION's argument list, and the FUNCTION body contains one FORM which was too big to print.

```
<..& <KRK <+ .A 5>> ▮ <SET BAR <ORG>> <&> &..>
```

Indicates that you are in the middle of a large FORM (e.g., a REPEAT or a PROG), positioned just to the left of the <SET BAR <ORG>>. In addition to the objects printed, there are more objects to both the left and the right, and there is one FORM which was too large to fit on the line.

3.2.2. Edit Commands

3.2.2.1. General

A sequence of EDIT commands is executed as soon as you type ESC. If one command fails, subsequent commands up to the ESC are ignored, and EDIT types out an appropriate error message. A failing EDIT command generally has no effect whatsoever; but see individual descriptions.

Note that *all* arguments to EDIT functions must be legal MDL objects. In particular, you can't search for

<SET , since the <>'s aren't balanced. Nor can you insert it. (But you can, for instance, search for and insert <SET THING 1>.)

If a command expects an argument and doesn't get one, an error message will be printed.

Many EDIT commands take FIXes as arguments. Those that do interpret the ATOM * as an argument to mean 'as many as possible'.

Whenever you are in EDIT, you have a well-defined 'position'. A position is a 'place' *inside* a MDL structure; this 'place' is either *between* two elements of the structure, or *between* an element and either end of the structure, or *inside* an empty structure. All editing, movement, and printing commands operate relative to your current position. The term 'cursor' is used in the following descriptions to refer to an embodiment of a position.

The format used in each of the following command descriptions is:

Command as Typed *English Name*

Description

3.2.2.2. General Commands

? **duh?**

Causes a short summary of all EDIT commands to be typed out. The same summary appears later in this chapter.

?? **huh?**

Similar to the above, but the summary is even shorter, and should fit entirely on the screen of an Imlac terminal.

Q **Quit**

Leave EDIT and return to MDL. (Causes EDIT to return the ATOM T.)

QR *fix* **Quit and Retry**

Quit from EDIT and then retry the frame specified, or by default, the one originally given to an open command or, if none was given, the frame beneath the last ERROR or LISTEN frame.

↑F **Control-F**

This is not really an EDIT command; rather, it is a character, obtained from the input stream at interrupt

level, which is used to return you to the EDIT Reader from some higher level of application, e.g., an ERROR's LISTEN. It is the EDIT equivalent of ERRET with no arguments.

↑F (or ↑S) typed during execution of an EDIT command is similar to normal MDL ↑S but returns to the EDIT Reader instead of the MDL LISTEN loop.

O *object* Open

Equivalent to Q followed by <EDIT *object*>. Positions the cursor just to the left of the first element of the entire object specified.

OT Open This

If the object to the right of the cursor is an ATOM, or a FORM whose first element is an ATOM, and the ATOM's value is openable, then it is opened. This command is useful when tracing a calling sequence through several functions.

3.2.2.3. Movement Commands

UT Up to the Top

Places the cursor at the position it had following an O.

R *fix* Right

Moves the cursor *fix* objects to the right, by default one. If *fix* is too large, i.e., there are not that many positions to the right of the current position, EDIT prints an error comment and the cursor stays where it is.

B Back

Moves the cursor as far to the right as possible.

L *fix* Left

Moves the cursor *fix* positions to the left, by default one. If *fix* is too large, EDIT prints an error message.

F Front

Moves the cursor as far to the left as possible.

DL Down Left

Positions the cursor just to the right of the rightmost element within the object to the left of the cursor, if that object is of PRIMTYPE LIST. Visually, the cursor moves left over one 'close bracket'.

DR Down Right

Positions the cursor just to the left of the leftmost element within the object to the right of the cursor, if that object is of PRIMTYPE LIST. Visually, the cursor moves right over one 'open bracket'. If the cursor is to the left of an element that is not of PRIMTYPE LIST, EDIT prints an error message.

D Down

Equivalent to **DR**.

UR *fix* Up Right

Positions the cursor just to the right of the object the cursor is currently within. Does so *fix* times, by default once.

UL *fix* Up Left

Positions the cursor just to the left of the object the cursor is currently within. Does so *fix* times, by default once.

U *fix* Up

Identical to **UL**.

S *object* Search

Does a depth-first, left-first tree-walk, (i.e., left-to-right) starting with the object to the right of the cursor, until the cursor is just to the right of an object structurally equal (i.e., =?) to its argument. An occurrence of the object will not be found if it is inside anything not of PRIMTYPE LIST. On failure, the cursor does not move. If the argument is omitted, the last object searched for is used.

SR *object* Search Right

Same as **S**.

SL *object* Search Left

Same as S, but the tree-walk is depth-first, right-first (i.e., right-to-left) and you end up to the left of the *object* for which you were searching.

3.2.2.4. Printing Commands

The Empty Command

Causes the normal 'ampersand print' to be done. This is principally useful when you are in 'silent' mode; see the V command.

By the way, an 'empty' command is typed by typing ESC without having typed any visible characters before it.

P **Print**

PPRINTs (not 'ampersand prints') the object to the right of the cursor.

PU **Print Up**

PPRINTs the object the cursor is in. This is similar to doing a U and then a P, although the cursor is not moved.

PT **Print Top**

PPRINTs the whole object you have open.

V **Verbosity**

Toggles the verbosity mode between 'verbose' (most commands cause ampersand printing) and 'silent' (printing of any sort is done only when some explicit print command is used, or when an error occurs). The current state of verbosity is the GVAL of E-VERBOSE.

In silent mode, absolutely *nothing* is printed after each command, not even new-lines or prompts. However, normal MDL evaluation still causes normal MDL printing.

3.2.2.5. Editing Commands

I *any* ... **Insert**

Inserts all its arguments immediately to the right of the cursor. None of its arguments are evaluated; you can insert unevaluated FORMs without using QUOTE. The cursor ends up to the right of the last object inserted.

G *any* ... **Get**

Same as I, but its arguments are evaluated. This is useful in conjunction with the X command (see below).

I: *type:atom fix* **Insert Type**

Grabs *fix* objects to the right of the cursor, inserts them into a newly created object of TYPE *type*, deletes them from the original structure, and inserts the newly created object in their place. In other words, it 'inserts'

the appropriate open and close brackets for *type* at the cursor and *fix* objects to the right.

By default *fix* is one, *type* is LIST. An error message is printed if *fix* is larger than the number of objects to the right of the cursor.

There is no way to directly insert or delete single parentheses, brackets, etc., using EDIT. Instead, use K: (see below) to remove pairs of brackets, and I: to insert them.

I* *indicator:atom* *new-structure* **Imbed**

Imbed looks for all occurrences of *indicator* in *new-structure* and replaces these occurrences with objects taken and deleted from the right of the cursor. It then inserts the result.

If only *new-structure* is given, the *indicator* is the ATOM *. If there aren't enough objects to the right of the cursor to replace each *indicator*, remaining indicators are left untouched and a warning message is printed. If no indicators are found, the new structure is inserted, but a warning message is printed.

I* is generally used to insert one or more structures into another complex structure in one operation, instead of several. For example:

```
<SET X ! <12 .Y>>
I* <COND (<NOT <LENGTH? .Y 11>> *)>$
<SET X <COND (<NOT <LENGTH? .Y 11>> <12 .Y>)> ! >
```

places a protective conditional around an NTH to prevent an out-of-bounds error.

IG *any* ... **Insert into Group**

Inserts into a group. IG is similar to I, but assumes that the object you are in is a group (as produced by GROUP-LOAD). Arguments to IG which are not ATOMs are inserted as in I. Objects which are ATOMs and which have a value insert a FORM which DEFINEs, SETGs, or SETs the ATOM as appropriate. Thus, to add a new function F to a group G, one could type

```
0 G$IG F$Q$
```

K *fix* **Kill**

Deletes *fix* objects to the right of the cursor. Defaults to one. Negative *fix* causes deletion to the left of the cursor.

C *any* **Change**

Changes the one object to the right of the cursor to its single argument. Does not move the cursor. Does not evaluate its argument. C is more efficient than K plus I.

C: *type:atom* Change Type

Changes the type of object to the right of the cursor to *type*. Attempts to do something reasonable for every type change. If you tell it to change a STRING to a LIST, you get a LIST of CHARACTERs. If you attempt to change a structure whose elements are other than CHARACTERs and STRINGs to a STRING, you will get a MDL error.

K: Kill Type

Deletes the brackets around the object to the right of the cursor. I.e., kills the object and inserts its elements into the structure of which it was a part.

SU *new old* Substitute

The Substitute command takes two arguments. All occurrences of *old* from the current location to the end of the open object (actually a search-right is done) are replaced by *new*. Once the search for *old* fails, the command terminates, and the number of substitutions performed is printed. The cursor is left after the last object replaced.

X *atom* Transfer

SETs the *atom* to the object to the right of the cursor. X can be used with K and G to move things around within the object being edited.

SW Swap

Swaps the two objects to right of the cursor, leaving the cursor pointing at the same object. The effect is to move the cursor and the object it points at one object to the right. Repeated SWs move cursor and object further and further to the right.

3.2.2.6. Macro Facility

M *macro* Macro

Takes either a STRING or something which EVALs to a STRING and performs all of the commands in the STRING. For complete assurance that your commands will be done properly, put an ESC between commands.

IT *fix macro* Iterate

This command (also called DO) takes a *fix* and *macro* as if an argument to M. This command will loop through the *macro fix* times or until an error is generated. When the iteration ends, the user is told how many

complete passes have been made of the *macro*.

In both of the above commands, if an EDIT error is generated, the *macro* will be terminated, and the *macro* itself will be printed, with an arrow pointing to the offending command. The cursor will remain at the place where the last legal command left it.

The SU command is, internally:

```
DO * "S old$L$C new$"
```

3.2.2.7. Cursors.

Cursors are locations in objects being EDITed. In addition to the main cursor, which is where editing occurs, other locations (also called cursors) may be remembered. The main cursor may be moved to another cursor in a single operation, potentially saving many motion commands. In large FUNCTIONs cursors may also reduce confusion by distinguishing among several similar areas of code.

UC

Use Cursors

The PACKAGE for dealing with cursors is not normally loaded in an initial MDL, so the UC command loads it and makes the cursor commands available. The PACKAGE loaded is "CURSOR".

CU *atom*

Cursor

CU takes an ATOM argument and SETs the ATOM to an object of type CURSOR, which tries to be clever in the event you change the object. Also, if you use the X command to name a substructure and then move copy it with G or I, the cursors in the substructure will follow to the new location.

There are some restrictions. Cursors in empty LISTs are okay but they will not follow the object to new locations. Also this 'following' feature is effective only at the first G or I after the X. To move the substructure again you have to X again.

I* is somewhat incompatible with CURSORs. Cursors in Imbedded structures will sometimes disappear.

GO *cursor*

Go

GO takes a *cursor* (normally the LVAL of an ATOM previously given as an argument to CU) and GOes to that position. If the *cursor* is illegal (not in the current top-level structure), an error message will be printed and you will remain in your previous position.

KC *atom*

Kill Cursor

Kill the cursor assigned to *atom*.

PC Print Cursors

Prints all cursors in the structure to the right of the main cursor.

PA Print All Cursors

Prints all cursors in the currently open structure.

3.2.2.8. Breakpoints

BK *predicate any ...* Breakpoint

Inserts a breakpoint 'around' the object to the right of the cursor. Takes any number of arguments. Subsequently, whenever that object would have been evaluated, you instead hit a breakpoint function which:

1. Evaluates *predicate*. If the value is FALSE, evaluation continues as if there were no breakpoint. If the value is non-FALSE, or if BK was given no arguments:

2. Types ****BREAK****.

3. For each argument after the first that you gave BK, types

   arg = EVAL of arg

4. Enters LISTEN.

You continue by applying ERRET to one argument, just as from an ERROR; the argument's value is ignored.

Breakpoints are implemented by inserting a BREAKR (a PRIMTYPE LIST with APPLYTYPE FORM) which consists of the function BREAKR and arguments, including the object breakpointed. A breakpoint prints as a glyph similar to the cursor:

object

If the ATOM SHORT-PRINT is assigned and FALSE, the actual BREAKR LIST is printed.

The breakpoint function returns EVAL of the thing it is put 'around,' and there are cases where this does not work. There are always equivalent places that do work.

1. Breakpoint on the first element of a FORM does not work. Put it on the whole FORM.

2. Breakpoint on a LIST which is an argument to a COND does not work. Put it on the first FORM in the LIST.

BA *predicate any ...* Break After

Similar to BK, but puts the breakpoint *after* the object at the cursor. Its action is like that of BK except that the break occurs after the object it is on is EVALed.

This sort of breakpoint prints like the 'before' sort, but with the glyph after the object broken:

object

The *predicate* for a BA breakpoint may check the value returned by EVAL for the object the breakpoint is on. This value is assigned by BREAKR to the ATOM VALUE.

KT

Kill This

Removes the breakpoint (if any) from the object to the right of the cursor.

KB

Kill Breakpoints

Removes all breakpoints in the currently open object.

3.2.2.9. Edit Monitors

There are several commands in EDIT which provide a simple interface to the "MONITOR" PACKAGE. These allow placing of monitors on references to or modifications of LVALs in interpreted MDL code.

For a more complete discussion of the use of monitors, see section 3.7.

UM

Use Monitors

The PACKAGEs for dealing with monitors are not normally loaded in an initial MDL, so the UM command loads them and makes the three commands for creating monitors available. The PACKAGEs loaded are "MONITR", which is the general monitor PACKAGE, and "EMONIT", which is the interface between EDIT and "MONITR".

RW *atom predicate any ...*

Read-write Monitor

The most general type of monitor that can be set is a read-write monitor. It will catch any reference to or attempt to modify the LVAL of the *atom* specified. The restrictions on placement of breakpoints also apply to monitors, with the addition that a monitor on an LVAL must be placed after that LVAL has become ASSIGNED?.

The second, third (and so on) arguments to RW are the same as those for BK. The *predicate* may be dependent on either the new or old value of the variable: These are available as the LVALs of NEWVAL and OLDVAL, respectively.

When a monitor is triggered, it prints the type of monitor, the variable being monitored, and any other information requested by the user, and then calls **LISTEN**.

A monitor prints as yet another glyph:

M[*atom*]*object*

where *atom* is the **ATOM** being monitored, and *object* is the object on which the call to **MONITOR** is placed.

Edit monitors are objects of type **BREAKR**, and thus they are killed by the same commands that kill normal breakpoints: **KB**, **KT**, and so on.

RM *atom predicate any ...* **Read Monitor**

RM is analogous to **RW**, but is only triggered by reading the variable.

WM *atom predicate any ...* **Write Monitor**

WM is analogous to **RW**, but is only triggered by writing the variable.

3.2.2.10. User-defined Edit Commands

It is possible to add user-defined commands to **EDIT**. The value of **EDIT-TABLE** should be a **VECTOR** of **STRING**s (commands) and **APPLICABLE** objects. **EDIT** will search **EDIT-TABLE** before its own command table. If a match is found, the **APPLICABLE** will be applied to three arguments: the command string, the **LOCATIVE** containing the item currently being edited (the immediately surrounding object) and the position in that item.

Note that user-defined commands should not be added except by constructing a new value of **EDIT-TABLE** from the commands to be added *and* the old value. Otherwise, any existing user-defined commands may be lost when new ones are added.

The Monitor commands described in section 3.2.2.9 are effectively 'installed' user-defined commands. They add elements to **EDIT-TABLE** when loaded by the **UM** command.

3.2.3. Examples

3.2.3.1. Simple Editing

Suppose you have the **FUNCTION**

```
#FUNCTION (('A) <EVAL .A>)
```

as the global value of the ATOM SIMP, and you wish to change it to

```
#FUNCTION (("BIND" B 'A) (<EVAL .A .B> .A))
```

using EDIT. The following example does just that: it includes doing the editing and applying of SIMP to an argument. Console input and output are shown below exactly as they would be in non-silent mode. (Console input consists of those characters to the left of every $). Note that there is nothing in SIMP which is big enough to warrant use of an &.

```
<EDIT SIMP>$
V$
#FUNCTION ( ■ ('A) <EVAL .A>)
D$
( ■ 'A)
I "BIND" B$
("BIND" B ■ 'A)
S .A$
<EVAL .A ■ >
I .B$
<EVAL .A .B ■ >
UR$
#FUNCTION (("BIND" B 'A) <EVAL .A .B> ■ )
I .A$
#FUNCTION (("BIND" B 'A) <EVAL .A .B> .A ■ )
L 2$
#FUNCTION (("BIND" B 'A) ■ <EVAL .A .B> .A)
I: LIST 2$
#FUNCTION (("BIND" B 'A) ■ (<EVAL .A .B> .A))
<SIMP <+ 1 2>>$
(3 <+ 1 2>)
#FUNCTION (("BIND" B 'A) ■ (<EVAL .A .B> .A))
Q$T
```

3.2.3.2. X and G Commands

In this example we have the FUNCTION

```
<DEFINE F (X)
        <G .X 10>
        <H 23 <- .X 1>>>$
```

By applying the X and G commands to the appropriate FORMs, we are able to swap the FORMs within the FUNCTION.

```
<DEFINE F (X)
        <G .X 10>
        <H 23 <- .X 1>>>$
F
<EDIT F>$
$
#FUNCTION ( ■ (X) <G .X 10> <H 23 <- .X 1>>)
R$$
#FUNCTION ((X) ■ <G .X 10> <H 23 <- .X 1>>)
X MOVER$
#FUNCTION ((X) ■ <G .X 10> <H 23 <- .X 1>>)
K$$
#FUNCTION ((X) ■ <H 23 <- .X 1>>)
R$$
#FUNCTION ((X) <H 23 <- .X 1>> ■ )
G .MOVER$$
#FUNCTION ((X)  <H 23 <- .X 1>> <G .X 10> ■ )
Q$1
.MOVER$
<G .X 10>
```

3.2.3.3. Unconditional Breakpoints

To insert unconditional breakpoints into the FUNCTION in the next example, do the following:

1. Define FIB and test the FUNCTION a few times.

2. Enter EDIT and position the cursor appropriately.

3. Insert the breakpoint.

4. Leave EDIT and run the FUNCTION again for the value 3. The breakpoint is exercised 5 times during this run.

```
<DEFINE FIB (X)
        <COND (<L=? .X 1> .X)
              (ELSE <+ <FIB <- .X 2>> <FIB <- .X 1>!$
FIB
<FIB 5>$
5
<FIB 6>$
8
<FIB 10>$
55
<EDIT FIB>$
R$$
#FUNCTION ((X) ■ <&>)
BK T .X$Q$T
<FIB 3>$
**BREAK**
.X = 3
LISTENING-AT-LEVEL 2 PROCESS 1
<ERRET T>$
**BREAK**
.X = 1
LISTENING-AT-LEVEL 2 PROCESS 1
<ERRET T>$
**BREAK**
.X = 2
LISTENING-AT-LEVEL 2 PROCESS 1
<ERRET T>$
**BREAK**
.X = 0
LISTENING-AT-LEVEL 2 PROCESS 1
<ERRET T>$
**BREAK**
.X = 1
LISTENING-AT-LEVEL 2 PROCESS 1
<ERRET T>$
2
```

3.2.3.4. Conditional Breakpoints

We continue from the previous example and demonstrate conditional breakpoints with the following:

1. Enter `EDIT` and kill the breakpoint from the previous example.

2. Position the cursor and insert a conditional breakpoint with a *predicate* of `<0? .X>`.

3. Leave `EDIT` and run the `FUNCTION` again for the value **10.**

4. Enter `EDIT` and remove the breakpoint.

```
<EDIT>$
$
#FUNCTION ((X) ■ ■<&>)
KB$$
#FUNCTION ((X) ■ <&>)
BK <0? .X> <TIME>$Q$T
<FIB 10>$
**BREAK**
<TIME> = 14.794538
LISTENING-AT-LEVEL 2 PROCESS 1
.X$
0
<ERRET T>$
**BREAK**
<TIME> = 15.252382
LISTENING-AT-LEVEL 2 PROCESS 1
.X$
0
<ERRET T>$
**BREAK**
<TIME> = 15.716037
LISTENING-AT-LEVEL 2 PROCESS 1
```

and so on. Eventually we reach the last breakpoint, and re-enter **EDIT**

```
<EDIT>$
$
#FUNCTION ((X) ■ ■<&>)
KB$Q$T
<ERRET T>$
55
```

3.2.4. Edit Command Summary

NAME	ARGS	MEANING
?	*none*	type out short summary
??	*none*	type out this summary
O	*any*	Open object or the value of an atom
OT	*none*	Open object at the cursor
Q	*none*	Quit and return to MDL
QR	*fix*	Quit and Retry frame
V	*none*	toggle Verbosity

Movement commands

L	*fix*	move Left *fix* objects
R	*fix*	move Right *fix* objects
U	*fix*	move Up *fix* levels
D	*none*	move Down one level
B	*none*	move to Back of object
F	*none*	move to Front of object
UR	*fix*	move Up *fix* objects and to the Right
DL	*fix*	move Down *fix* objects and to the Left
UT	*none*	Up Top -- go to the place you were after you did O

Editing commands

I	*any...*	Insert arguments to the right of cursor
I:	*type,fix*	make next *n* objects into a *type*
I*	*atom,object*	Imbed command: replace all occurrences of *atom* (default *) in *object* with objects to right of cursor
IG	*any...*	Insert into group
SU	*new,old*	SUbstitute *new* for *old*
X	*atom*	set the atom to the object to right of cursor
G	*any...*	Get EVAL of arguments, insert to right of cursor
SW	*none*	SWap the two objects to the right of cursor
C	*any*	Change the next object to arg
C:	*type*	Change the type of the next object to *type*
K	*fix*	Kill (delete) the next *fix* objects
K:	*none*	Kill (remove) the 'brackets' around the next object

Search Commands

S/SR	*any*	Search (Right) until match (=?) is found for *any*
SL	*any*	Search Left as above

Macro Commands

M	*string*	execute the string as if typed to EDIT
IT/DO	*fix,string*	ITerate the execute *string fix* times

Printing commands

P	*none*	PPRINT the next object
PU	*none*	PPRINT the next Upper level
PT	*none*	PPRINT the whole object open

Cursor commands

UC	*none*	Use Cursors
CU	*atom*	set *atom* to CUrrent cursor position
GO	*cursor*	GO to the specified cursor position
PC	*none*	Print Cursor positions in the current object
PA	*none*	Print All cursor positions in the top-level object
KC	*atom*	Kill the Cursor assigned to the atom

Debugging commands

BK	*pred,any...*	set BreaKpoint at next object; if *pred* evaluates to FALSE, don't break; rest of arguments are printed out at break
BA	*pred,any...*	set Breakpoint After next object
KB	*none*	Kill all Breakpoints in open object
KT	*none*	Kill This breakpoint in the object to the right of cursor

Monitor commands

UM	*none*	Use Monitors
RW	*atom,pred,any...*	set Read-Write monitor on *atom*
RM	*atom,pred,any...*	set Read Monitor on *atom*
WM	*atom,pred,any...*	set Write Monitor on *atom*

↑F and ↑S return you to EDIT from a higher level.

The ATOM * may be used as a *fix* argument whose value is the largest legal value for that command.

3.3. Debugging and the Interpreter

Before continuing the discussion of the various packages that are used in the debugging of MDL code, we will expand on the discussion of ERROR, FRAME, (and so on) in Chapter 16 of [3]. To summarize that chapter, whenever an ATOM is bound or a FUNCTION or RSUBR is MCALLed in MDL, information is added to the contol stack. This information, normally 'invisible', may be examined using the functions described in a previous section (FRAMES, FR&, FRLVAL, etc.). An invocation of ERROR puts MDL into a LISTEN-like loop.

Successive ERRORs stack up and are reflected in the LISTENING-AT-LEVEL message printed whenever ERROR or LISTEN is called.

In addition to being examined, the stack may be modified as part of the debugging procedure. For example, the SUBRs SET and LVAL take an optional second argument which may be (among several possible TYPEs) a FRAME. EVALing

```
<SET X 10 <FRM n>>
```

would change the LVAL of X in the nearest binding lower in the stack than the FRAME *n* FRAMEs lower than the most recent call to ERROR or LISTEN. Similarly

```
<LVAL X <FRM n>>
```

examines the LVAL in a particular FRAME.

The most common use of the MDL interpreter in debugging is to invoke the SUBR ERRET. With no arguments, it drops all the way to the bottom of the stack and then calls LISTEN: It says 'I give up' (although side effects are not undone). More commonly, ERRET is given a single argument, which causes the last invocation of ERROR or LISTEN to return that argument. For example, suppose a program contains , FOO but FOO has no GVAL. MDL would respond

```
*ERROR*
UNASSIGNED-VARIABLE
FOO
GVAL
LISTENING-AT-LEVEL 2 PROCESS 1
```

You could give up, saying <ERRET>, but it is often more reasonable to say 'Oh, yes, FOO was supposed to be 1000', and then

```
<ERRET 1000>
```

Still better is

```
<ERRET <SETG FOO 1000>>
```

which will prevent future ERRORs from the same cause.

Finally, ERRET may be given a second argument of a FRAME, which means to return the first argument as the value of the invocation of that FRAME. In the previous example, the programmer might look at the stack (with FR& or FRAMES) and see

```
1  GVAL          [FOO]
2  EVAL          [,FOO]
3  EVAL          [<+ .X .Y ,FOO>]
4  EVAL          [<LOSER .A .B>]
5  EVAL          [</ ,GOOD-GVAL <LOSER .A .B>>]
6  EVAL          [<WINNER 1.0 2.0>]
7  LISTEN        []
```

After some thought, he may just say 'Well, LOSER apparently needs some debugging, but for now I'm interested in WINNER', in which case he can 'fake' a reasonable return from LOSER by typing

```
<ERRET 342.0 <FRM 4>>
```

which returns 342.0 exactly as though LOSER had returned it.

More complex errors are sometimes more difficult to fix, requiring the use of EDIT (at least). In the above example, the programmer might decide to debug LOSER after all. There are two ways to go about this: First, if the problem is localized, the FRAME itself may be edited (which is to say, the *contents* of the FRAME may be edited). Changes will show up in the FUNCTION from which the FRAME's contents were derived. The newly corrected FRAME may then be RETRYed. For example,

```
<EDIT 3>$
...various editing commands
QR$
```

Second, the function itself may be edited. In the process, it may be so changed that the FORM which caused the ERROR no longer even exists. Often, the easiest solution is to retry the invocation of the EDITed FUNCTION from scratch: in this case

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

As always, the major restriction to remember is that side-effects are not undone by RETRY.

3.4. Loading and Dumping

GROUP-LOAD and GROUP-DUMP are used to load and dump files of MDL programs in such a way that the contents of the file are made available in a MDL structure called a *group*. Many other PACKAGEs in the MDL environment operate on or change groups: Among them are "EDIT", "GLUE", "PDUMP", and the MDL compiler.

GROUP-LOAD and GROUP-DUMP are almost as widely used as FLOAD as a way of dealing with groups of MDL functions. Consequently, they are already loaded in most initial MDLs, as part of the package "GRLOAD".

```
<GROUP-LOAD file-name:string
            group-name:atom>
```

file-name:string is the file to load.

group-name:atom is the name to give the group. It is optional and by default the ATOM formed by PARSE of the first name of the file to load. The group will be stored as the LVAL of *group-name*.

GROUP-DUMP is the opposite of GROUP-LOAD. It outputs the group from the MDL to the file given as its first argument. Functions unchanged since the last GROUP-LOAD are copied from the original input file. Functions that have been edited are output using the routine given as the third argument to GROUP-DUMP.

```
<GROUP-DUMP file-name:string
            group-name:atom
            print-routine
            kill-breakpoints?>
```

file-name:string is the only required argument. It is the file to which to output the group.

group-name:atom is optional, and defaults as it does for GROUP-LOAD, but of course gives an ERROR if the group doesn't already exist.

print-routine is optional, and defaults to ,PPRINT unless the group contained NBIN format RSUBRs, in which case ,PRINC is used.

kill-breakpoints? is optional, by default T, in which case GROUP-DUMP kills all EDIT breakpoints and monitors in objects being dumped. Giving a fourth argument of a FALSE to GROUP-DUMP prevents this.

On the surface, it appears that little happens in the process of loading a file and making it into a group. However, a great deal of information about the group has been stored away in associations for later use. Some of this information is of use to the MDL programmer:

1. On an association between *group-name* and the ATOM CHANNEL is stored a LIST giving the name of the file that was GROUP-LOADed to form the group. Removing this association before GROUP-DUMPing has the effect of making the entire group be output from core rather than copied from the original source.

2. On an association between *group-name* and the ATOM MAGIC-RSUBR the ATOM T is stored if the group contained any RSUBRs in fast (NBIN) format. It is this association which is used to determine the default *print-routine* in GROUP-DUMP.

3. The OBLIST path in effect at any time during the load is available. The original path is stored on an association between *group-name* and the ATOM BLOCK. Within the group, the path changes are stored in an association between the group RESTed to the point of change and the ATOM BLOCK.

4. If the second element of a FUNCTION definition is not an ATOM, the actual FUNCTION name gotten by EVAL of that element is stored as an association between the original element and the

ATOM VALUE.

5. The location of a function within the input file is stored as a LIST of the starting and ending offsets (in characters) of the function, under an association between a locative to the GVAL of the FUNCTION name and the indicator DEFINE. This association is removed by EDIT (and other editors) to indicate that the FUNCTION has been changed.

There are additionally several switches that affect the operation of GROUP-LOAD:

.KEEP-FIXUPS

If the LVAL of KEEP-FIXUPS is true (and GROUP-LOAD binds it that way during loading), the fixups of RSUBRs GROUP-LOADed will be kept.

.EXPFLOAD

If the LVAL of EXPFLOAD is true, FLOADs will be expanded. That is, the objects in the file FLOADed will be added to the group in place of the FLOAD. The initial setting of EXPFLOAD is a FALSE.

.EXPSPLICE

If the LVAL of EXPSPLICE is true, any objects returned within SPLICEs will be inserted directly into the group as described above. The initial setting of EXPSPLICE is a FALSE.

3.5. The One-step Debugger

The MDL One-step debugger allows the user to step through the evaluation of any MDL expression one 'operation' at a time. Between steps, variables may be examined or changed, functions edited, and so on. This is possible because the debugger runs in a different MDL PROCESS than the expression being stepped, and a MDL PROCESS may 1STEP another [3]. To load the Debugger, <USE "DEBUGR">.

The MDL Debugger can be in any of three states. In the initial state, OFF, no one-stepping occurs and the Debugger does not listen for any special interrupt characters. The Debugger is, therefore, completely inactive. By typing <DEBUG> to MDL, you leave the OFF state and enter the READY state. In the READY state no one-stepping occurs, however the Debugger does listen for interrupt characters. By typing the interrupt character ↑B, you enter the ON state and one-stepping begins. In addition, if you were stopped at an EDIT breakpoint when the ↑B was typed, the breakpoint will automatically be exited and evaluation continued in the one-stepping state.

While in the ON state, the Debugger will proceed through the execution of any MDL objects one step at a time. In essence, the Debugger stops just before and just after every call to EVAL. At each step the Debugger will indicate its current condition as follows. If EVAL is recursively entered at level, *n*, with input, *object*, the display will be:

n=> *object*

(where *object* is ampersand printed). If EVAL is returning from level, *n*, with result, *object*, the display will be:

n<= *object*

(where *object* is ampersand printed).

The Debugger will stop at each such step and wait for directions. There are four interrupt characters that may be typed to proceed further in the program: ↑N, ↑O, ↑R and ↑A. They each take an optional prefix argument that serves as a repeat count.

↑N

causes the Debugger to perform the next step of the current evaluation.

↑O

causes the current object to be completely evaluated without any one-stepping and then stops with the result of that evaluation. ↑O is useful for stepping over COND predicates that you know will not succeed, or more generally, uninteresting parts of a program.

↑A

is similar to ↑O, but specific to the evaluation of the argument list of a FUNCTION, PROG, or REPEAT. Typing ↑A during such evaluation allows the rest of the argument list to be evaluated without one-stepping and then stops before evaluating the body of a FUNCTION, PROG, or REPEAT or returning of a result.

↑R

is most effectively used in a REPEAT or PROG loop. Typing ↑R causes evaluation to proceed until control returns to the point in the body of the REPEAT/PROG at which ↑R was typed. It thus allows you to go once around a loop.

It should be noticed that, when stopped at one of these steps, you can examine and modify program variables, do a FRAMES or FR&, EDIT FUNCTIONs and set breakpoints, and in general perform any valid MDL operations. Also, when you stop, the LVAL of the ATOM LAST-OUT will be set to the object the Debugger last typed out. This is useful if the & performed by the Debugger did not show a particular detail that you are interested in.

Use the interrupt character ↑E to leave the ON state and return to the READY state. Use the interrupt character ↑Q to leave either the ON state or the READY state and return to the OFF state. When leaving the ON state as described, the execution currently being one-stepped will be finished in the usual manner.

The function REPAIR attempts to fix any errors in the Debugger that you might happen to invoke. These errors are easily distinguished since they never occur in MDL's MAIN PROCESS. Therefore, you will see:

LISTENING-AT-LEVEL *m* PROCESS *n*

(where *n* is not 1). REPAIR turns off the Debugger and returns you to running in the MAIN PROCESS (no longer one-stepping). Because REPAIR turns off the Debugger, you must do <DEBUG> again if you wish to try any further one-stepping.

3.5.1. MDL Debugger Command Summary

<USE "DEBUGR"> loads the Debugger.

<DEBUG> makes the Debugger ready.

↑B *begins* one-stepping.

↑N performs the *next* step of the computation.

↑O steps completely *over* the next computation, then stops and continues one-stepping.

↑A evaluates the *arguments* of the current object then stops and continues one-stepping through the body.

↑R continues evaluation until you *return* to this point.

↑E *ends* one-stepping.

↑Q *quits* one-stepping and makes the Debugger *unready* (turned off).

<HELP> prints a command summary.

<REPAIR> attempts to repair any Debugger errors you might invoke.

3.5.2. MDL Debugger Special Features

The following flags have special importance to the Debugger:

,INDENT-INC

is the amount by which to indent for each level (by default 2 spaces).

,INDENT-MOD

The indentation-level is the real level taken modulo this number. The default is 10. Indentation 'restarts' when level gets here. If you don't like this feature, make the number large.

,INDENT-DIF

is the minimum amount of free space to reserve on each line that indentation must not touch (by default 20). Therefore at level L the indentation is exactly:

```
<MIN <* ,INDENT-INC <MOD .L ,INDENT-MOD>>
     <- <13 ,OUTCHAN> ,INDENT-DIF>>
```

```
,OUT-FAST
```

if true the Debugger will not stop when leaving a level with a result. The default is T.

```
,OUT-UNIQUE
```

if both this and previous flag are true successive 'outs' of the same item will not be displayed (defaults to T).

```
,SELF-FAST
```

if true the Debugger will not stop when entering a level with an object which EVALs to itself (e.g. ATOMs, FIXes, STRINGs). The default is T. The display will be:

n: *object*

```
,FORM-FAST
```

if true the Debugger will not stop when entering a level with any of the 'short' FORMs (e.g. <>, .FOO, ,BAR, 'ANYTHING). The default is T. The display will be:

n: .FOO = *lval*

Any of these flags can be SETGed by you to tailor the Debugger to your own tastes.

3.6. Execution Tracing

The "TRACE" PACKAGE provides a facility for observing the arguments and returned values of selected FUNCTIONs and RSUBRs. It is possible to print the arguments on entry to the function, print the value returned, and to break on entry to and exit from the function. All actions may be performed conditionally. To load TRACE, type

```
<USE "TRACE">
```

3.6.1. Using TRACE

TRACE is invoked by

```
<TRACE what options>
```

what is either an ATOM or a LIST of ATOMs, naming the things to be traced. These may include SUBRs, FUNCTIONs, and RSUBRs; however, anything which is traced must EVAL all of its arguments. *options* specifies the behavior of TRACE with respect to the specified function. There are five switches, as follows:

```
IN-BREAK
```

means break (cause a MDL ERROR) before calling the function. Normally off.

IN-PRINT

means & function arguments on entry. Normally on.

OUT-PRINT

means & function value on exit. Normally on.

OUT-BREAK

break after executing the function call. Normally off.

VERBOSE

means & the arguments to the function one per line. This is useful if the arguments are long. Normally off.

To cause a given option to be unconditionally on, include its name (an ATOM) in the *options* TUPLE. To cause an option to be unconditionally off, include a two-element LIST, composed of the option name and a FALSE. If the second element of the LIST is neither FALSE nor an ATOM, it will be EVALed each time TRACE examines the setting of the given option for the function. This allows conditional breakpoints, for example.

Thus:

```
<TRACE FOO (OUT-PRINT <>)>
```

will cause FOO's arguments to be printed on entry, but the value will not be printed.

```
<TRACE FOO (OUT-PRINT '<G? <TIME> 4.0>)>
```

will cause printing of the value after four seconds of cpu time have been used. Printing of the arguments will occur each time FOO is called.

UNTRACE turns off tracing of the specified functions:

```
<UNTRACE what:atom-or-list>
```

What defaults to a LIST of all functions which have been traced.

3.6.2. Understanding TRACE

TRACE works by CHTYPEing the specified functions to new types which have an APPLYTYPE associated with them. This means that one cannot trace calls to RSUBRs or RSUBR-ENTRYs which are already linked. In addition, it means that UNTRACE must be used to get the old value back. To determine the status of a function with respect to tracing, say

```
<GET applicable TRACE>
```

This returns FALSE if *applicable* is not traced; otherwise, it returns an object which describes the settings of the various options. The object has a PRINTTYPE which associates the name of each option with its setting:

```
<GET ,FOO TRACE>$

FOO
IN-BREAK:  #FALSE ()
IN-PRINT:  T
OUT-PRINT:  <G? <TIME> 4.0>
OUT-BREAK:  #FALSE ()
VERBOSE:  #FALSE ()
```

Individual settings for a particular function may be changed by PUTting into this structure:

```
<PUT <GET ,FOO TRACE> ,IN-BREAK T>
```

causes a break whenever FOO is called.

3.7. Monitors

A common problem in debugging is the mysterious 'clobbering' of some value or element of a data structure. MDL has imbedded in it a mechanism for triggering interrupts on references, either for reading or writing, to values of variables and elements of structures.

The "MONITOR" PACKAGE is designed to be a readily accessible user interface to these "READ" and "WRITE" interrupts in the MDL interpreter.

To obtain "MONITOR",

```
<USE "MONITOR">
```

There are three basic kinds of 'things' which can be monitored: values of ATOMs, elements of STRUCTUREDs (the TYPE of the element is not important), and ASSOCIATIONs.

For ATOMs, the LVAL or the GVAL may be monitored. If the LVAL is to be monitored, the ATOM must be ASSIGNED?. For the GVAL, the ATOM must be GBOUND?. If these conditions cannot be met, a monitor cannot be generated.

For STRUCTUREDs, the monitor is on the *n*th element, where *n* is specified when the monitor is created. Remember, the monitor is on a slot of the STRUCTURED, not on the contents of that slot!

For ASSOCIATIONs, the monitor is on the association itself.

3.7.1. Monitor Internals

This section expands on the discussion of monitors in the MDL document itself [3].

MDL defines two types of monitors: Read and Write. These are implemented in the language by two

interrupts, READ!-INTERRUPTS and WRITE!-INTERRUPTS, respectively. In addition, the "MONITOR" PACKAGE can allow read-write monitors. The "MONITOR" PACKAGE is at base a set of functions to create and handle these interrupts. A monitor is triggered in the following cases:

Read monitor:
- For LVALs -- via LVAL
- For GVALs -- via GVAL
- For STRUCTUREDs -- via NTH
- For ASSOCIATIONs -- via GET and GETPROP

Write monitor:
- For LVALs -- via SET or "AUX" bindings
- For GVALs -- via SETG
- For STRUCTUREDs -- via PUT, SUBSTRUC
- For ASSOCIATIONs -- via PUT and PUTPROP

Note that PUTRESTs of LISTs which may alter the *n*th element of a LIST, do not access the old *n*th element of the LIST and therefore do not cause a write monitor to trigger.

Internally, MDL performs monitoring on LOCATIVEs to STRUCTUREDs. In fact, LVAL and GVAL are really pointers to an internal structure. This need not concern the user except in the case of LVALs of ATOMs. In this case, MDL will monitor a LOCATIVE to *that* (exactly that unique) binding of the ATOM. When that binding becomes invalid, or more precisely,

```
<NOT <LEGAL? locative>>
```

a function in the "MONITOR" PACKAGE will make the monitor vanish. Illegal monitors print as #MONITOR [ILLEGAL] (if you ever get a pointer to one). Remember that if you want to monitor the LVAL of an ATOM bound in a FUNCTION (or PROG, etc.), you must create a new monitor each time, as a new binding is created each time. One way to do this is to edit into the FUNCTION a call to MONITOR (see below) after the ATOM becomes ASSIGNED?. Fortunately, EDIT (see section 3.2.2.9) has commands to do exactly that.

3.7.2. Creating MONITORs

Creation of all monitors is done through a call to MONITOR (which returns an object of TYPE MONITOR), as follows:

```
<MONITOR type:string
         object
         where
         predicate
         todo:tuple >
```

where:

type is one of "READ", "WRITE", or "RW".

object is either an ATOM or a STRUCTURED, or an ASSOCIATION item.

where is either LVAL or GVAL (if *object* is an ATOM) or a FIX. (if *object* is a STRUCTURED), or an ASSOCIATION INDICATOR.

predicate is something which is EVALed to determine whether the monitor is to be triggered; this defaults to T. The "MONITOR" PACKAGE defines three variables which can be referenced in the test:

OLDVAL is the old value of the object monitored.

NEWVAL is the new value of the object monitored.

MONOBJ is the object monitored (ATOM or STRUCTURED).

Here *value* means LVAL, GVAL, or element. Obviously, NEWVAL is not set for "READ" monitors.

todo is any number of things to be EVALed and PRINTed when the monitor is triggered.

Note that *predicate* and *todo* are identical to the analogous arguments of the EDIT BK command.

3.7.3. Monitor Events

When a monitor is triggered, the following is printed (remember the *predicate* is evaluated before this), and then LISTEN is called. To continue, <ERRET T>.

Read:

```
**READ of where of object**
Value: oldval
todo1 = result1
todo2 = result2
...
```

Write:

```
**WRITE of where of object**
Old value: oldval
New value: newval
todo1 = result1
todo2 = result2
...
```

A slightly different first line format is used for associations.

3.7.4. Killing Monitors

Killing a MONITOR is accomplished by calling KILL-MONITOR as follows:

<KILL-MONITOR *monitor*>

or

<KILL-MONITOR *type object where*>

In the latter case, *type*, *object*, and *where* are as given in the original call to MONITOR.

To kill all MONITORs, use

<KILL-ALL-MONITORS>.

3.7.5. Other Monitor Routines

<MONOBJ *monitor*>

returns the *object* monitored.

<MONSPEC *monitor*>

returns the *where* of the MONITOR.

<CLEAN-MONITORS>

flushes invalid MONITORs from the MONITOR LIST. This is done internally and need not be called routinely.

,MONITORS

is a LIST of all current MONITORs.

3.7.6. What You Can't Do with Monitors

You can't monitor the LVAL of something BOUND? but not ASSIGNED?. E.g.,

```
<DEFINE WRONG ("AUX" BAR)
        <MONITOR "READ" BAR LVAL>
        ..&.. >
```

You can't expect compiled code to cause monitors to be triggered. Naturally, you can't place monitors in compiled code; however, a compiled reference to a monitored ATOM will not usually cause the monitor to trigger either.

3.8. FINDATOM

The "FINDATOM" PACKAGE is intended to reduce the problems caused by multiple OBLISTs and lengthy ATOM names in MDL. It allows one to find all ATOMs whose PNAMEs match some specification, which need not be exact; in addition, one may place constraints on the values of the ATOMs found.

FINDATOM is invoked as:

```
<FINDATOM specstr:string
          searchlist
          constraints
          outobl:list>
```

specstr is a STRING describing the PNAMEs of the ATOMs one wishes to find. Three special characters are recognized in this STRING:

*: matches anything, including an empty string

=: matches any single character

†Q: quotes the following character

Search strings may be an arbitrary concatenation of normal and special characters. For example:

"*SDM*": matches any ATOM containing "SDM" anywhere in its PNAME.

"*=SDM*": matches any ATOM containing "SDM" in its PNAME, provided that at least one character precedes the "SDM".

"†Q*": matches any ATOM with PNAME "*".

"*": matches any ATOM.

If †Q is the only special character in the string, it need not be quoted: "†Q" searches for ATOMs with PNAME "†Q".

searchlist specifies the OBLISTs to search. Possible values are:

#FALSE (): search all OBLISTs in .OBLIST

#FALSE (*oblists-or-forms*): search all but the OBLISTs specified.

oblist: search only this OBLIST.

list-of-oblists: search only the OBLISTs in this list.

else: search all OBLISTs. This is the default.

constraints is a TUPLE describing the value of each ATOM found. It may consist of any number of valid TYPE names, along with arbitrary structures and the following special objects:

T: if present, overrides any other constraints; if no other constraints are specified, this is assumed. Any ATOM matching *specstr* will be accepted.

ANY: overrides any constraint other than T. Any ATOM matching *specstr* which has a value (either GVAL or LVAL) will be accepted.

<>: any ATOM which has no value will be accepted. Note that giving both ANY and <> is equivalent to giving T.

LINK: any LINK will be accepted.

If other constraints are provided, they work as follows: all valid TYPE names given (ones for whom VALID-TYPE? returns T) are stored in a structure; when a value is encountered, its TYPE is MEMQed on this structure. If the ATOM does not succeed here, it is next checked against the 'arbitrary structures.'

Anything in *constraints* which is neither one of the above 'special objects' nor a valid type is treated as a DECL specification. All such objects are put in a FORM starting with OR, which has the effect of generating a single DECL specification. When a value is found, DECL? is called with the value as its first argument and the generated FORM as its second. If DECL? returns T, meaning that the FORM is valid as a DECL for the VALUE, the ATOM is accepted.

Examples:

```
ATOM FALSE '<LIST [REST FIX]>
```

specifies that any ATOM accepted must have either a GVAL or an LVAL which is of type ATOM or FALSE, or which is a LIST of FIXes.

```
'<OR ATOM FALSE> '<LIST [REST OBLIST]>
```

specifies that any ATOM accepted must match the DECL

```
<OR <OR ATOM FALSE> <LIST [REST OBLIST]>>
```

outobl, if present, is a LIST of OBLISTs which is the LVAL of OBLIST when FINDATOM prints things. Thus, one may force all ATOMs to be printed with full trailers by providing an empty LIST here. The last argument given to FINDATOM, provided it is a LIST, is assumed to be *outobl*.

FINDATOM prints the name of each ATOM it accepts, followed by the STRING "Gassigned" and the type of GVAL if the ATOM has one; this will be followed by the STRING "Assigned" and the type of the

LVAL if the ATOM has one. It prints the number of ATOMs found when it finishes.

3.9. "PINFO"

"PINFO" is an informational PACKAGE. It is used to examine the OBLISTs of the PACKAGEs loaded into an MDL. There are two major entries in PINFO.

<PCK-INFO *package:string*
internal?:boolean>

Both arguments to PCK-INFO are optional. If neither argument is given, the names of the PACKAGEs loaded into the MDL are listed. If a *package* is given, the contents of the package's ENTRY OBLIST are listed, as well as information about the VALUE of each ENTRY. If *internal?* is provided and non-FALSE the contents of the internal OBLIST are also listed. PCK-INFO prints an error message if package is not loaded.

<PCK-USES *package:string*>

lists the names of PACKAGEs USEd by package or returns a FALSE if package is not loaded.

3.10. Debugging in a Run-time Environment

A fairly common occurrence when running 'debugged' code is to find that if was not after all *completely* debugged. It is useful to be able to load interpreted versions of some FUNCTIONs in a PACKAGE into the compiled environment for debugging. "DFL", "RDFL", and "UNLINK" are PACKAGEs written to simplify this procedure.

3.10.1. DFL

The "DFL" ('Debugging Fload') PACKAGE is a set of routines for loading and dumping of small numbers of FUNCTIONs from a larger file. It is useful in debugging already running systems, or ones which have not been GROUP-LOADed. To get "DFL"

<USE "DFL">

The main entry of the "DFL" PACKAGE is DFL:

<DFL *func-names file-name:string unlink?:boolean*>

where all arguments are optional and

func-names is the name(s) of the DEFINEd FUNCTION(s) to be obtained from this file. It may be an ATOM, a STRING, or a structure of ATOMs or STRINGs; if ATOMs are given, their SPNAMEs are used. The default is the argument last given to DFL or RDFL.

file-name is the file to obtain the FUNCTION(s) from. The default is the last file DFLed or RDFLed. An ATOM may be given, in which case its SPNAME is used for the first file name.

unlink? If this is true, and if one or more of the values replaced by the DFLed FUNCTIONs were RSUBRs or

RSUBR-ENTRYs, the reference VECTORs of *all* RSUBRs, including pure ones, will be searched for occurrences of the old value; such occurrences will be replaced by the ATOM. This is the inverse of RSUBR-LINKing. Pure structures will be unpurified; this does not change their address in core, but simply makes the page they live in read/write.

In the normal case, if an RSUBR or RSUBR-ENTRY is being replaced, unlinking will occur automatically in garbage-collector space only if RSUBR-LINK is T. Also, remember that unlinking is not the same as substituting: only RSUBRs stored at top level in reference VECTORs are found; if the old value itself was in a structure (such as a dispatch table), it will not be replaced.

3.10.2. RDFL

RDFL is similar to DFL but is for reloading RSUBRs rather than FUNCTIONs. RDFL is contained in the PACKAGE "RDFL".

<RDFL *func-names file-name unlink? glue?*>

The first three arguments are as for DFL. The only difference between RDFL and DFL (barring the effect of the fourth argument) is that RDFL searches in the file for '<SETG ' rather than '<DEFINE '.

glue? If non-FALSE, RDFL will READ and EVAL the next object in the file following each RSUBR read. This will in the normal case obtain the 'glue bits' for the RSUBR (see section 6.1). The default for *glue?* is

<AND <ASSIGNED? GLUE!- > .GLUE!- >

This is the FORM used in NBIN files to determine whether glue bits should be kept.

Note that RDFL will work to reload any SETGed object, not just RSUBRs.

RDFLing an RSUBR-ENTRY does not work and may well be fatal: you must RDFL the RSUBR in which the RSUBR-ENTRY is an entry, as well.

3.10.3. UN-DFL

UN-DFL is for writing out DFLed FUNCTIONs after EDITing.

<UN-DFL *atoms filnam force?*>

atoms is an ATOM or a list of ATOMs, which will be UN-DFLed. The FUNCTIONs defined must all be from the same file, or UN-DFL will not work. UN-DFL can only UN-DFL things which were previously loaded by DFL.

filnam The default is the file the ATOMs originally came from.

force? Normally, UN-DFL will object if there is a version between the file the FUNCTIONs came from and the file which UN-DFL will create: it thinks it will likely destroy useful information. Providing an ATOM here causes this scruple to be ignored. It is almost always unwise to do so. For example:

```
<DFL (FOO BAR)> <UN-DFL FOO> <UN-DFL BAR>
```

will cause UN-DFL to fail. Moral: DFL and UN-DFL your FUNCTIONs together.

3.10.4. UNLINK

The "UNLINK" PACKAGE contains three entries: UNLINK, PURE?, and UNPURIFY. UNLINK is sometimes called by DFL; PURE? and UNPURIFY are good ways to figuratively defeat the safety 'interlock' of MDL.

UNLINK is used to unlink RSUBRs after they have been linked. (See the discussion of RSUBR-LINK in [3]).

```
<UNLINK atoms pure?>
```

atoms is a list of the ATOMs to be unlinked, or a FALSE, meaning unlink every RSUBR in the MDL, or a group-name, meaning unlink calls to all FUNCTIONs and RSUBRs in the group.

pure? is optional and defaults to FALSE, but if true, even pure RSUBRs will be searched. UNLINK examines all the OBLISTs in the MDL, looking for RSUBRs; if an RSUBR exists only in a structure, and not at top level in any RSUBR's reference VECTOR, it will not be found.

```
<UNPURIFY pure-object:any>
```

PURE? takes an object and determines if the right half of the value word is greater than the number contained in the MDL location PURBOT, which is the lowest pure location in MDL. Ergo, 'Is the object I gave you pure?' It is only meaningful for structures.

```
<UNPURIFY pure-object:any>
```

UNPURIFY takes a single argument, which must be of PRIMTYPE VECTOR or UVECTOR (i.e., it must have an AOBJN pointer for its value word). It causes the pages in which that object lives to become impure, and returns T.

Because there is no way on ITS to make a read-only page an impure page directly, the following algorithm is used by UNPURIFY:

1. Is the object pure, according to PURE? If not, leave.

2. Is UNPURIFY-PAGE!-IUNLINK GASSIGNED? If not, get a page from the interpreter, and SETG the aforementioned ATOM to its number. I.e., the page is more or less permanently taken for use of UNPURIFY.

3. For each page occupied by the object: a) If the page is already impure, do nothing; b) otherwise,

map the page on top of UNPURIFY-PAGE; c) create a new, impure page where the old page was. d) copy the contents of UNPURIFY-PAGE back to the old, now impure page.

Thus, no pointers are changed: as far as MDL is concerned, in fact, nothing has changed. The unpurified pages are still pure, according to its page map. However, you may freely change the unpurified object.

If your change to the newly unpurified object consists of PUTing a pointer into garbage-collected space into the object, you may lose completely unless the pointer points to a frozen object. The MDL garbage collector does *not* examine unpurified objects. UNLINK can only use UNPURIFY because all ATOMs referenced by pure RSUBRs are indeed frozen.

For the above reason, use of UNPURIFY is not recommended for the general user.

3.11. CRITIC

"CRITIC" is a PACKAGE designed to aid the user in debugging (and perhaps increasing the efficiency of) his programs. It accumulates and prints in a readable format information about the interactions of the various FUNCTIONs (and LVALs and GVALs) in a group. It also warns the user about various conditions it considers to be either non-optimal or erroneous, such as incorrect use of SPECIAL, forgetting to QUOTE some structure, and so on. Like most critics, it is sometimes wrong, but it tries to perform a useful service. To load "CRITIC" say

```
<USE "CRITIC">
```

There are two major entries, one of which prints more information than the other.

```
<CRITIC group-name
        output-file>
```

where *group-name* is the ATOM returned by a GROUP-LOAD, and the optional *output-file* is a STRING giving the name of the file to output to (by default with second file name "CRITIC"). This can also be a CHANNEL if you are planning to do several CRITICs into one file. CRITIC prints information about interactions among the FUNCTIONs in a group (as described below).

```
<CRITIC-NOTES group-name
              output-file>
```

is similar but only prints 'errors' and 'warnings' -- things that might be problems with the FUNCTIONs in the group.

The output format (for each FUNCTION and for the group as a whole) is as follows:

function (*object number of function in group*)

Called-by: a list of all the functions which call *function*

Calls: a list of all the functions called by *function*
SETG: external globals SETGed by *function*
GVAL: external globals referenced by *function*
SET: external variables SET by *function*
LVAL: external variables referenced by *function*
SPECIAL: variables declared SPECIAL by *function*
USE-DATUM: DATUMs used by *function*

The above table is printed by CRITIC but not by CRITIC-NOTES. 'External' as used above means 'External to *function*'.

CRITIC-NOTES and CRITIC both print information about possible defects or errors in each FUNCTION. These can be any or all of the following (explanations follow where needed).

3.11.1. Global problems with the Group

FLOAD in file.

This is pretty minor: FLOADs at top level are discouraged if you can avoid them.

BLOCK or ENDBLOCK at top level in PACKAGE.

PACKAGEs should not have to resort to this.

atom-name: **MANIFESTed structure.**

The ATOM given is a structure but was MANIFESTed. Since a MANIFEST is copied within the reference VECTOR of any RSUBR that uses it, it is usually not a good idea.

ENTRYs not bound, assumed locals: *atom-list*

The ATOMs given were made ENTRYs in the PACKAGE, but were not bound, so CRITIC has assumed they are locals, for lack of something better to do.

Packages USEd but never referenced: *package-names*

These PACKAGEs were in USE statements but no ATOM was ever found which fell on their OBLISTs. There will sometimes be incorrect entries in this list if you USE a PACKAGE which sets up a funny ENTRY OBLIST (RPACKAGEs included) or no OBLISTs at all.

Internal functions unused: *atom-list*

These are FUNCTIONs DEFINEd but apparently never referenced and not entries. There will sometimes be incorrect entries in this list if you have FUNCTIONs invoked only by funny dispatching methods, such as

APPLYing or EVALing an element of a structure.

`Internal globals unused:` *atom-list*

ATOMs SETGed at top level but never referenced.

`Internal manifests unused:` *atom-list*

ATOMs SETGed and MANIFESTed at top level but never referenced.

3.11.2. Parameter list problems

`ATOM` *atom-name* `used twice in parameter list.`

The ATOM named was bound twice in the same parameter LIST within the FUNCTION. MDL doesn't worry about this, but you might.

`Untasteful re-use of ATOM` *atom-name* `in ROOT.`

An ATOM was bound which happened to be in the ROOT OBLIST and happened to have a GVAL that is a SUBR or FSUBR. This is reported because the ATOM will have to be unpurified, which is expensive.

`"BIND" illegally located.`

A "BIND" was found other than at the beginning of a parameter LIST.

`"CALL"/"ARGS" illegally located.`

A "CALL" or "ARGS" was found after the "AUX" in a parameter LIST.

`"OPTIONAL" illegally located.`

"OPTIONAL" was found after "AUX" in a parameter LIST.

`"TUPLE" illegally located.`

"TUPLE" was found after "AUX" in a parameter LIST.

atom `"AUX" illegally QUOTEd.`

The ATOM named was given as a quoted argument in the "AUX" part of the parameter LIST.

`External locals set but unbound and unDECLed:` *atom-list*

`External locals set but unbound:` *atom-list*

Two different classes of hacking an external local. In both cases it means that the ATOMs did not appear to

be improperly SPECIALed, since no one bound them higher in the call tree (or at top level). These are most often indications of misspelling or forgetting to put a temporary in the parameter LIST.

`External locals used but unbound and unDECLed:` *atom-list*

`External locals used but unbound:` *atom-list*

A reference to an external local which was not bound anywhere is probably a misspelling of a SPECIAL bound elsewhere or the result of forgetting to put the ATOMs in the FUNCTION's parameter LIST.

`External locals set but unDECLed:` *atom-list*

`External locals used but unDECLed:` *atom-list*

An external used but not DECLed usually means that the compiler will produce poorer code.

3.11.3. Unused ATOMs

`Argument unused:` *atom-list*

The arguments listed were never referenced.

`Unused:` *atom-list*

The ATOMs listed were bound at top level of the FUNCTION and never referenced.

`Unused in PROG:` *atom-list*

Similar to the above, but the ATOMs were bound within a PROG.

`Unused in REPEAT:` *atom-list*

Similar to the above, but the ATOMs were bound within a REPEAT.

`Unused in FUNCTION:` *atom-list*

Similar to the above, but the ATOMs were bound within a nameless FUNCTION, such as the second argument to a MAPF/MAPR.

`Unused SPECIALs:` *atom-list*

The same as above (including ' ... in FUNCTION', etc.), except that the ATOM was SPECIAL. This message results from really looking down the call tree, so it is more accurate about this problem than the compiler, which only looks at the FUNCTION in which the ATOM is bound.

3.11.4. Function calling errors

`Calls undefined function` *atom*.

The `FUNCTION` calls an undefined `FUNCTION` (undefined at the time `CRITIC` ran).

`Calls` *function* `with too few arguments.`

`Calls` *function* `with too many arguments.`

`External FUNCTION` *function*

The `FUNCTION` named is called but doesn't seem to fall on any of the `OBLIST`s associated with the group.

3.11.5. SPECIAL/UNSPECIAL problems

`SPECIALs never used as SPECIALs:` *atom-list*

The `ATOM`s were made `SPECIAL` but never used outside the `FUNCTION` in which they were bound.

atom-name `is unused or should be SPECIAL.`

A very specific error which means that the `ATOM` given (always one of `INCHAN`, `OUTCHAN`, or `OBLIST`) was bound but never referenced within the `FUNCTION`, and was not `SPECIAL`: Either you bound it for effect and forgot to `SPECIAL` it, or you didn't need to bind it.

atom `unbound in paths:` *path-list*

If the `FUNCTION` is called by one of the paths given, the *atom* will be unbound. A path is just a list of calls `CRITIC` has found are possible, such as `(FOO BAR BLECH)`, meaning '`FOO` is called by `BAR` which is called by `BLECH`'.

`The ATOM` *atom* `used in` *fcn1* `should be special in` *fcn2*.

This note will appear with both `FUNCTION`s mentioned. It means that *atom* is referenced in *fcn1* and the nearest `FUNCTION` that binds it and calls down to *fcn1* is *fcn2*.

3.11.6. DECLing problems

`RSUBR has no DECL.`

`FUNCTION has no DECL.`

`Parameters not DECLed:` *atom-list*

The `ATOM`s given were bound but not `DECL`ed in the parameter list of a `FUNCTION`, `PROG`, or `REPEAT`.

`No DECL in DECL for:` *atom-list*

The ATOMs in the *atom-list* given had no associated declarations.

`NEWTYPE not DECLed:` *type-name*

A NEWTYPE of a structured type was made but no DECL argument was included. In a structured NEWTYPE, including a DECL of the interior can greatly increase the efficiency of compiled code.

`Illegal DECL:` *atom-list decl reason*

The DECL pair given had illegal syntax for the reason given. These can include:

"Not a legal type": An object appeared in a DECL that was not an ATOM, FORM, or SEGMENT.

"Type-name not a type: *atom*": Something other than a type-name or special symbol (such as ANY) appeared where a type was expected. This is sometimes caused by not having your environment completely set up when CRITIC is run.

"FORM/SEGMENT too short": A FORM/SEGMENT construction of only one element was found.

"SPECIAL/UNSPECIAL with three or more elements"

"Bad PRIMTYPE type": The type given in a PRIMTYPE was not a type-name.

"PRIMTYPE with three or more elements"

"Bad type of structured type": The type-name given as the type of a structured type was not a type. For example, <FOO FIX> where FOO is not a type.

"Bad BYTES specification": A BYTES specification was not of the form <BYTES *fix fix*>, or the byte size was greater than 36.

"BYTES DECL too short": A BYTES construction of only one element was encountered.

"BYTES DECL too long": A BYTES construction of more than three elements was encountered.

"VECTOR in OR specification": An NTH/REST/OPT construction was found at top level of an OR.

"Nth/REST/OPT too short": A one-element NTH/REST/OPT.

"Only REST or OPT may follow OPT": Something other than a REST or OPT was found after an OPT.

"REST must terminate DECL": Something was found after a REST in the DECL.

3.11.7. Miscellaneous

`Possibly should be QUOTEd:` *structure*

The structure given will be =? to itself if EVALed. CRITIC lists these under the assumption that you might have forgotten to QUOTE a structure that should have been. It says "possibly" because you obviously want to build new structure sometimes. One way to do this without offending CRITIC is to build new structure with explicit calls to LIST, VECTOR, etc.

3.12. Program Environments

The ENV PACKAGE makes it easier to load programs into different environments. It allows certain actions to be taken during loading only if a given 'feature' is present. ENV has three ENTRYs, and is preloaded.

`<FEATURES` *features:tuple*`>`

If given no arguments, FEATURES returns the current feature LIST. If its first argument is not a FALSE, the arguments are added to the feature LIST. If the first argument is FALSE, the remaining arguments are removed from the feature LIST. Thus,

```
<FEATURES "COMPILER">
```

says that we are currently in a compiler. All of the 'feature' arguments may be either STRINGs or ATOMs; internally features are stored as STRINGs to avoid OBLIST problems.

`<FEATURE?` *features:tuple*`>`

returns T if any of its arguments is on the feature LIST.

`<EVAL-WHEN` *features*
consequences:tuple`>`

uses the first argument to decide whether to evaluate the remaining arguments.

features specifies which feature(s) to look for. It may be a single feature or a LIST of features. In the latter case, if the first element is a FALSE, what is checked for is the absence of the features listed. Note that this argument is often a LIST created out of arguments to FEATURE?.

consequences are things to be evaluated only if the features are present (or absent, in the FALSE case).

For example,

```
<EVAL-WHEN GLUE <SETG FOO 1>>
```

would perform the SETG only if it's evaluated in a GLUE (or some other environment defining that feature).

```
<EVAL-WHEN (<> COMPILER) <SETG BAR 2>>
```

would not perform the SETG in the compiler environment.

Unfortunately, the ENV PACKAGE is a relatively recent innovation, and so many programs do not set up appropriate environments.

4. The Library System

A coherent unified library system serves to facilitate the sharing of algorithms and data by imposing a discipline appropriate for the particular environment. The MDL Library System provides:

- A uniform access method for referring to functions and data outside of the current logical group;
- Lexical blocking, eliminating difficulties arising from overlap of names between different logical groups;
- Automatic loading of functions for the user who knows only the name of the function which is wanted;
- A facility whereby functions which may be necessary only in unusual situations are loaded only in the event that they are needed.

The MDL Library System may be divided into distinct parts. These are:

- The Package System, the collection of routines used to provide lexical blocking for a logical group (see section 2);
- The 'explicit' loading facility, the routines used to explicitly indicate that references are being made to a particular logical group;
- The 'implicit' (or 'dynamic') loading facility, the machinery for automatically loading functions when they are needed during console interaction.

4.1. Program Libraries

In the previous discussion of the Package System and USE (see section 2.3.2), we glossed over the mechanism by which a PACKAGE is loaded when another PACKAGE (or the user at his terminal) refers to it. We will now give the details.

There are two types of loading common in MDL programming: 'explicit' loading, such as USE may initiate, and 'implicit' or 'dynamic' loading, initiated by attempting to call or examine a function that is not currently loaded.

In the case of 'explicit' loading, it is necessary somehow to map the name of a PACKAGE into a file name which contains the body of that PACKAGE. The mechanism for doing so must be flexible enough to allow both 'installed' programs (those that have been debugged and submitted to the library) and developmental programs to be loaded. It must also be tailorable for special needs, such as libraries for specific systems and personal libraries for individual users.

In the case of 'implicit' loading, the further mapping from the specific ENTRY of a PACKAGE referenced to the PACKAGE itself must be performed. It must deal with the case of two or more PACKAGEs each containing an ENTRY with the same PNAME.

For programs that are 'public' or 'installed', both of these mappings are performed by a library. A library is a file which contains pointers between the names of ENTRYs of PACKAGEs and the PACKAGEs containing them, and from PACKAGE and DATUM names to the files containing them.

The standard library is named LIBMUD and lives on a directory named LIBMUD (on ITS) or MDLLIB (on Tenex/Tops-20), but other libraries, personal or special purpose, may also exist; the mechanisms for creating and maintaining them are the same in both cases.

4.1.1. Library Searching

When a PACKAGE is USEd, MDL first checks to see if the PACKAGE is already loaded, by looking up the PACKAGE name on the PACKAGE OBLIST. If the PACKAGE is not yet loaded, MDL must search for the file containing the body of the PACKAGE.

When MDL searches, it does so under the direction of a search path stored as the LVAL of the ATOM L-SEARCH-PATH. This LVAL is a LIST, each element of which specifies 'a place to look' for the PACKAGE. These elements may be:

"file-name"

A STRING refers to a library file; "LIBMUD;LIBMUD" for example.

[]

An empty VECTOR refers to the <SNAME> directory. The directory will be searched for files whose names are the name of the PACKAGE being loaded (truncated to six characters on ITS) and second names from the LVAL of the ATOM L-SECOND-NAMES, which is a VECTOR of STRINGs which are possible second names for the file.

[*dir:string-or-false*]

A non-empty VECTOR specifies a directory. The first element of the VECTOR gives the directory as a STRING or a FALSE, the latter case meaning <SNAME>. If that is the only element, L-SECOND-NAMES specifies the file names to look for. If there are other elements, they should be STRINGs to use in place of L-SECOND-NAMES.

A search path may consist of any number of such elements. The loader will examine them sequentially, attempting to find the PACKAGE being loaded.

The initial LVAL of L-SEARCH-PATH (on ITS) is

```
("LIBMUD" "LIBMUD;LIBMUD" [] ["MBPROG"] ["MPROG" ">"])
```

and on Tenex/TOPS-20, it is

```
("LIBMUD" "<MDLLIB>LIBMUD" [] ["MDLLIB"])
```

This instructs the loader to first search the user's personal library (if it exists), then the 'public' library. Next, search the user's directory for a file whose first name is the PACKAGE name, and whose second name is specified by L-SECOND-NAMES. If that fails, perform the same search on the library directory, and finally (on ITS), look for a source version of the PACKAGE on the source directory.

The initial LVAL of L-SECOND-NAMES (on ITS) is

```
["FBIN" "GBIN" "NBIN" ">"]
```

and on Tenex/TOPS-20, it is

```
["FBIN" "GBIN" "NBIN" "MUD"]
```

To give a simple example of how this mechanism may be tailored for individual needs, consider a programmer debugging a subsystem. If he wants his debugging versions of various PACKAGEs to be loaded before the installed versions, he CONSes a new element onto L-SEARCH-PATH so that it contains

```
([] "LIBMUD" "LIBMUD;LIBMUD" [] ["MBPROG"] ["MPROG" ">"])
```

(assuming the files with his debugging versions are on the <SNAME> directory).

4.1.2. Dynamic Loading

To ease the use of 'top level' routines from the console, a feature is provided whereby the Library System can load a PACKAGE of functions automatically when one of the functions which is an ENTRY in that PACKAGE is invoked by name. This facility is not available for use by other PACKAGEs of functions, which must refer explicitly, via USE, to PACKAGEs which they require: while a human can resolve the difficulty of possible multiple PACKAGEs with ENTRYs of the same name, a program cannot.

When an error is generated because a FORM is evaluated, and the first element of that FORM is an ATOM which has no value, and the particular ATOM is in the INITIAL OBLIST, an error handler established by the Library System determines if there are any PACKAGEs in the current libraries which contain an ENTRY with the same name as the PNAME of that ATOM. If there is one such PACKAGE, it is loaded, and the evaluation which got the error is continued with the correct value. If there is more than one such PACKAGE, the possible choices are displayed, the user is asked which is the desired PACKAGE, and it is loaded. If there are no PACKAGEs with ENTRYs of the correct name, the error is not handled, and so it will fall into the standard error mechanism. This same procedure is also invoked when GVAL is applied to an ATOM on the INITIAL

OBLIST and the ATOM has no value.

4.1.3. USE-DEFER

It is sometimes desirable to have available functions that are rarely invoked, but are nonetheless available. (One example would be certain error handling routines.)

The USE-DEFER function sets up the OBLIST path so that, when a reference is made to an ENTRY in the specified file, the correct ATOM is found, but the PACKAGE is not actually loaded at that time. When a function at a later time tries to call the function which is the value of one of the entries in this PACKAGE, the whole PACKAGE will be automatically loaded. USE-DEFER has two constraints which USE does not. First, the PACKAGE must be in one of the currently active libraries; it may not simply be a file as in the case of USE. Second, no reference may be made to ATOMs which are entries but do not have values which are applicable. In other words, ATOMs which are entries because they are data (rather than functions) may not be referenced when USE-DEFER is employed instead of USE.

Because USE-DEFER utilizes the dynamic loader, which utilizes the ERROR interrupts, USE-DEFER will not work in a demon or any other MDL program which sets up its own error handlers. All such MDL programs should SETG the ATOM L-NO-DEFER to a non-FALSE, which (as explained previously) will cause USE-DEFER to behave exactly like USE. Then, PACKAGEs containing a USE-DEFER can be used without modification in demons and the like.

4.1.4. USE-TOTAL

USE-TOTAL is analogous to USE, but instead of splicing in only the ENTRY OBLIST of the PACKAGE, it additionally splices in the internal OBLIST. This is useful in some debugging situations, as it reduces the number of trailers printed and also makes the internal identifiers of the PACKAGE more accessible.

4.1.5. Translations

It is occasionally useful to have more than one copy of a particular PACKAGE loaded at once. One example that comes to mind is the case of debugging a debugging PACKAGE. The Library System contains a mechanism for 'translating' a PACKAGE name into another one. More specifically, it is possible to tell USE: 'If you ever load the PACKAGE named *foo*, pretend it was named *bar* instead.' Note that this does not change the searching and loading procedure described above, only the names of the OBLISTs and so on used to store the ATOMs in the PACKAGE.

<TRANSLATE *old:string new:string-or-false*>

causes the PACKAGE *old*, when it is USEd, to behave as if it were named *new*. If *new* is FALSE, it means that *old* should be loaded as though it were not a PACKAGE at all; its ATOMs will appear on the DEFAULT OBLIST or <1 .OBLIST> (normally INITIAL).

<UNTRANSLATE *old:string*>

causes any translation of *old* to be removed.

<TRANSLATIONS>

lists all translations currently in existence.

,L-TRANSLATIONS

is a LIST containing all the translations.

4.1.6. The Library Data File

In addition to its ability to map between PACKAGEs, ENTRYs, and the files which contain them, the library serves another purpose. If a user is compiling a function which USEs a given PACKAGE, that PACKAGE is not usually going to be run. All that is necessary is to examine the calling sequences of its functions, and make sure that all 'side-effects' (such as the definition of new TYPEs) occur. If only these necessary parts of the PACKAGE are loaded, a great saving of time and space is effected.

The library data file provides a way of achieving this end. When a PACKAGE is added to the library, more information than the list of ENTRYs and the file containing the PACKAGE is collected. In particular, MANIFEST GVALs, NEWTYPE definitions, some MACROs, and RSUBR DECLs are stored. Since this is the information used by the compiler, one can save a great deal of space and time by using information from the library where possible.

If ,L-USE-DATFILE is true, USE of a PACKAGE will load from the data file if possible. It is impossible if the PACKAGE has changed since the data file entry was created. In those cases, the PACKAGE itself is loaded instead. If ,L-ALWAYS-DATFILE is true, an ERROR will result if the data file entry is outdated; one can ERRET T to cause the real PACKAGE to be loaded.

USE-DATFILE is just like USE, except that it temporarily SETGs L-USE-DATFILE and L-ALWAYS-DATFILE to T.

The data file contains, for each PACKAGE, information for each interesting ENTRY: MANIFEST GVALs, NEWTYPE definitions, RSUBR DECLs, and MACROs. It also has, of course, the lists of ENTRYs and RENTRYs needed by the dynamic loader. It does not contain other structures, nor does it contain functions. When a

PACKAGE is loaded from the data file, it is effectively USE-DEFERed; if you end up needing to run part of the PACKAGE, it will be loaded dynamically.

Some PACKAGEs can not have data file entries. If a PACKAGE defines MACROs that use data not stored in the data file (if the MACRO calls a FUNCTION, for example), the PACKAGE will not get a data file entry: it would normally end up being loaded from the file anyway.

It is possible for a data file entry to become obsolete (if a new version of a PACKAGE is created without the library entry being updated). For this reason, the library is examined periodically for such entries and an attempt is made to update the appropriate entries.

4.1.7. Run-time Switches

There are a number of variables which may be set dynamically to tailor the Library System's performance.

.L-SEARCH-PATH

as described above (see section 4.1.1) is a LIST specifying the libraries and directories to look in, and the files to look for when trying to load a PACKAGE. This variable is used by USE, USE-DEFER, USE-DATUM, and the dynamic loader.

.L-SECOND-NAMES

as described above (see section 4.1.1) is a VECTOR of the second names of files to look for when attempting to load a PACKAGE from a directory.

.L-NOISY

If the GVAL of L-NOISY is non-FALSE, the names of PACKAGEs and DATUMs are printed whenever they are loaded, dynamically or otherwise. This feature may be turned off by SETGing L-NOISY to #FALSE (). L-NOISY has an initial GVAL of T.

.L-NO-MAGIC

Dynamic loading may be disabled by SETGing L-NO-MAGIC to a non-FALSE. L-NO-MAGIC has an initial GVAL of a FALSE.

.L-ALWAYS-INQUIRE

If the GVAL of L-ALWAYS-INQUIRE is non-FALSE, the dynamic loader will always ask the user before it loads anything. The GVAL of L-ALWAYS-INQUIRE is initially a FALSE.

.L-NO-DEFER

If the GVAL of L-NO-DEFER is non-FALSE, USE-DEFER will work exactly like USE. L-NO-DEFER is initially SETGed to #FALSE ().

4.1.8. Library Utility Functions

A number of functions exist which allow the user to examine libraries, list their contents, and retrieve their entries. All of the functions below except L-PATH and L-OBL accept an optional STRING argument, a library specification. If it is defaulted, they operate on the public library, specified by the string "LIBMUD;LIBMUD" or "<MDLLIB>LIBMUD".

<L-LOAD *package:string library:string*>

L-LOAD requires a STRING (the name of a PACKAGE or DATUM) and attempts to load it from *library* (if given) or the current libraries, as per L-SEARCH-PATH.

<L-FIND *function-name:string library:string*>

L-FIND requires a STRING (the name of an ENTRY), returning a UVECTOR of two-element VECTORS of the form:

[*package-in-which-function-exists:string*
library-in-which-package-exists:string]

This finds all of the entries which have the same PNAME but are in different PACKAGEs.

The remaining functions are in the PACKAGE "L", rather than in the PACKAGE "PKG". For each of these, the optional *library* argument is by default *the* library; that is, "LIBMUD;LIBMUD" or "<MDLLIB>LIBMUD".

<L-FILE *package:string library:string*>

L-FILE requires a STRING (the name of a PACKAGE or DATUM) and returns a STRING which is the file specification of the file, pointed to by the library, which contains the body of that PACKAGE or DATUM.

<L-WHERE *package:string library:string*>

L-WHERE is similar to L-FILE but returns a VECTOR of STRINGs which is the actual complete file specification of the file containing the PACKAGE (i.e., the 'real' slots in a CHANNEL open to the file).

<L-LISTE *library:string*>

L-LISTE prints the names of all of the entries of all of the PACKAGEs in the library.

<L-LISTP *library:string*>

L-LISTP prints the names of all of the PACKAGEs and DATUMs in the library.

<L-COUNTE *library:string*>

L-COUNTE returns a FIX, the number of entries defined by all of the PACKAGEs in the library.

<L-COUNTP *library:string*>

L-COUNTP returns a FIX, the number of PACKAGEs and DATUMs in the library.

<L-LISTPE *package-string library:string*>

L-LISTPE requires a STRING (the name of a PACKAGE) and prints the names of all of its entries.

<L-PATH>

L-PATH prints a list of the names of all of the OBLISTs in the user's current OBLIST path.

<L-OBL *atom*>

L-OBL requires an ATOM and returns an ATOM, the name of the first ATOM's OBLIST. L-OBL is in fact

<GET <OBLIST? *atom*> OBLIST>

4.1.9. Internal Library Functions

There are several internal functions used for searching libraries (which is, after all, all the Library System ever does).

<PACKAGE-FIND *package:string library:string*>

searches *library* for *package*. If there is no such PACKAGE or DATUM in *library*, it returns a FALSE. Otherwise, it returns a STRING, which is the name of the file containing *package*.

<ENTRY-FIND *entry:string-or-atom library:string*>

searches *library* for PACKAGEs containing *entry*. It returns a FALSE if there are none, otherwise a LIST some multiple of four elements long, where each set of four elements describes a package containing an ENTRY with that PNAME. These elements are:

package:string is the PACKAGE being described.

file-name:string is the file-name containing the package.

rpackage?:atom-or-false indicates, if non-FALSE, that the package is in fact an RPACKAGE.

rentry?:atom-or-false indicates, if non-FALSE, that the entry is an RENTRY.

<DEFER-FIND *package:string library:string*>

returns a FALSE if the PACKAGE or DATUM is not found, or a VECTOR of five elements describing the PACKAGE.

rpackage?:atom-or-false indicates, as above, whether the package is an RPACKAGE.

name:string is the name of the package.

file-name:string is the file containing the package.

entries:list is a LIST of the PNAMEs of the ENTRYs of the package.

rentries:list is a LIST of the PNAMEs of the RENTRYs of the package.

This is all the information about the package that the library contains.

4.1.10. Library Maintenance

The PACKAGE called "LUP" contains functions used to modify libraries, and to add, update and delete PACKAGEs and DATUMs. It should be noted that libraries do not contain the bodies of PACKAGEs and DATUMs. Rather, they point to files which contain these.

<LUP-ACT *library:string*>

requires one argument, a library specification STRING, and activates the library so specified. If the library doesn't exist, it is created. In order to protect the library from loss due to system or MDL crashes, activating a library for modification copies the library data files and locks the library so that no one else may modify it. Modifications are made to the copies, which are renamed back over the originals only when the library is explicitly deactivated. Obviously, PACKAGEs added to a library aren't available, even to the person adding them, until the library is deactivated.

<LUP-DCT>

deactivates the currently active library.

<LUP-ADD-PACK *package-file:string*
update?:boolean
datfile-entry?:boolean>

package-file is a file specification of the file containing the body of the PACKAGE to be added. LUP-ADD-PACK will find the PACKAGE statement within the file (or complain if it can't).

update? is optional, and if non-FALSE, it allows the PACKAGE to update an older version of itself, something which is not otherwise allowed. Note that, since the library points to the file which contains the body of the PACKAGE, that file should not be deleted later, else the library won't be able to find it.

datfile-entry? is by default T, but if it is FALSE, no entry will be created in the datfile for this PACKAGE. Since datfile entries are generally useful only in the compiler (and similar environments), it doesn't do much good to have them for PACKAGEs that are only called from top level (e.g., FINDATOM).

When adding a PACKAGE to the public library, the PACKAGE's object file should be copied to the appropriate library directory ("LIBRM*n*" on ITS, or "<MDLLIB>" on Tops-20) and the library pointed at that copy of the file. If no library is activated when LUP-ADD-PACK runs, it will activate "LIBMUD;LIBMUD" or "<MDLLIB>LIBMUD".

<LUP-ADD-DATUM *name:string*
file:string
update?:boolean>

is analogous to LUP-ADD-PACK, adding a DATUM to the active library. LUP-ADD-DATUM requires two STRING arguments, the name of the DATUM and the specification of the file which contains the body of the DATUM. LUP-ADD-DATUM will accept the same optional argument that LUP-ADD-PACK accepts, with the same meaning and default. The same restrictions concerning the file which contains the DATUM also apply.

<LUP-DEL *package:string*>

LUP-DEL requires one STRING argument, the name of a PACKAGE or data set, and deletes that PACKAGE or DATUM from the currently active library. LUP-DEL does not touch the file containing the body of the PACKAGE or DATUM.

<LUP-MOVE *package:string* *file:string*>

causes the file pointer of *package* to be changed to point to *file*. This is a faster operation than re-adding the PACKAGE, and it is intended for situations in which an existing library file has been moved for some reason.

<LIB-GC *library:string*>

garbage-collects the library in question, if this is required. Garbage-collection is occasionally useful since it causes all the elements of each hash bucket to live near each other in the library file, thus improving performance during searches. It also allocates some free storage in each page of the file.

4.2. The Pure-mapping Library

The basic idea behind MDL pure mapping is to separate out the code part of RSUBRs in compiled programs. The RSUBRs themselves are kept in a file known as an FBIN (see 6.3). These RSUBRs do not contain the code but instead point to a file which contains the code. This scheme has several advantages. First, the code can be dynamically mapped in when needed. This allows MDL to use more code than will fit in the virtual address space of the machine it is running on. Secondly, since the code is pure it can be shared between several MDLs using it. Finally, the FBIN file itself is smaller than a corresponding NBIN file and therefore FLOADs more rapidly.

In the most basic implementation of FBINs, there are three files: the FBIN, the SAV file (which contains the code), and the FIXUP file, which contains the information necessary to update the SAV FILE for new releases of MDL. As is obvious, this entails a lot of files, and potentially a lot of file directories. The MDL Pure-mapping Library reduces this storage overhead by collecting all of the SAV and FIXUP files together.

The scheme uses two large data bases, each contained in one file. The data bases are called 'SAV' and 'FIXUP'. These files store all currently existent SAVs and FIXUPs for all existing versions of MDL. Each data

base is structured like a file system. There is a main 'directory' that points to a number of other 'directories', each of which points to a number of 'files' inside the data base. In this section the word 'file' or 'directory' in quotes refers to an object inside a data base. The files containing the data bases are named (on ITS) "MUDSAV;SAV FILE" and "MUDSAV;FIXUP FILE". On Tenex/TOPS-20, they are "<MDL>SAV.FILE" and "<MDL>FIXUP.FILE".

4.2.1. The Demon

While all MDLs can read from the Pure-mapping Library, there is only one program which can write into it. This is a maintainer demon which runs once a day to keep the Library updated. This demon can add 'files', delete 'files', and add 'subdirectories' to both data bases.

To facilitate updating of the Library there is a directory on which to put files to be added as well as files to indicate what is to be deleted. This is the "MUDTMP" directory on ITS and the "<MDLLIB>" directory on Tenex/TOPS-20. Any file on it with the second name of SAV*nnn* or FIX*nnn* (where *nnn* is a 2 or 3 digit MDL release number) will be added to the appropriate data base. If the files "DELETE SAVS" or "DELETE FIXUPS" exist, then they will be used to delete 'files' from the data bases. These files must be ASCII files of the form

filename 1 [SPACE] *filename 2* [CRLF]

An example of a valid delete file is as follows

```
NCODGE SAV53
1NCODGE SAV53
```

The demon will ignore any deletion requests for 'files' not in the data base.

The demon does its work in several passes. The basic passes are the delete pass, the planning pass, the update pass, and the salvage pass. The delete pass deletes 'files' if either a "DELETE SAVS" or "DELETE FIXUPS" file exists on its working directory. The planning pass builds a plan file by examining the working directory and calculating where new 'files' will be placed in the data bases. The planning pass builds two files using a special internal format. These files will be used by the update pass to add 'files' to the data bases. The planning pass also enlarges the data base files as much as necessary to accomodate the new 'files'. The update phase reads the plan files and adds new SAV and FIXUP 'files' to the data bases. If a 'directory' overflows, a new 'directory' is added during this pass, and all the 'directories' are recreated (i.e., all the 'files' have to be rehashed, since they were originally placed in a 'directory' according to a hashing algorithm based on the number of 'directories'). The salvage pass is used to pick up any free storage that has been lost through system crashes or lost through holes created during the updating of the data bases.

Throughout the entire processing of the data bases attempts are made to keep the data bases in a consistent state. 'Directories' are updated only after 'files' are guaranteed to be in the data bases. The plan files described are used to keep the data bases consistent in case the system crashes while the demon is in the update pass.

A major goal in the design of the data bases is to allow recovery in case of demon errors or system disk crashes. To this end the data bases are backed up on tape every other week. (It would be dumped more often but the file is currently over two million words long). This of course leaves the problem that 'files' added to the data bases between dumps could be lost in a disk crash. To aid in recovery from such a crash, all 'files' added between dumps are copied to the "MUDRST" directory (on ITS) or the "<MDL.SV>" directory (on Tenex/TOPS-20). Moreover a file is kept listing all the 'files' added during the previous week. This file is called "ADDED FILES". All this information is deleted once the data base is dumped to tape.

4.2.2. User Programs

Occasionally it is useful for a user to list the data base 'directories', to see if certain 'files' are in it, and copy 'files' out of the data base. DBMAIN is a program which allows the user to do these things.

The following are functions available to the user.

4.2.2.1. Listing Functions

<CLISTF *data-base:string*>

is used to list all the 'files' in a data base. It takes one optional argument which is the name of the data base (either "SAV" or "FIXUP"). If no argument is supplied, "SAV" is used by default. (This is always the default whenever a function takes an optional argument specifying the data base.) CLISTF prints each 'file', its length, and where it is located. The format of a line of listing is as follows:

fn1 fn2 size block

where *fn1* is the first 'file' name, *fn2* is the second 'file' name, *size* is the length of the 'file' in blocks (1024. words for SAVs, 256. words for FIXUPs), and *block* is the block at which the 'file' starts. This is the format used whenever listing 'files'.

<LISTF *data-base:string directories*>

is used to list all the 'directories' of an entire data base. It takes two optional arguments, the *data-base* to be listed, and a specification of which '*directories*' to list. The '*directories*' may be:

a FIX: list the 'directory' specified by the FIX;

a LIST of FIXs: list the 'directories' specified in the LIST;

the ATOM ALL: list all the 'directories' (this is the default).

<FLIST *data-base:string*>

lists free areas of storage in the data base. It lists the free storage in the form:

length block

where *length* is the length of the area of free storage and *block* is the block number of the starting block. This function takes one optional argument which is the name of the data base to be examined. At the end of the listing it will tell the total amount of free storage.

4.2.2.2. Find Functions

<FIND-FILE *file:string data-base:string*>

is used to find a specific 'file'. It takes as its argument a 'file' specification and prints the 'file' name along with the information printed by the listing functions if the 'file' exists, otherwise it returns an object of type FALSE. The 'file' specification must be of the form:

"*dir*;*fn1 fn2*"

where *dir* is either SAV or FIXUP and *fn1* and *fn2* are the first and second 'file' names respectively.

<SPEC-FIND *fn1:string data-base:string*>

is used to find all 'files' with the same basic name, disregarding the leading digit(s) which are added to make 'file' names unique. It takes one required argument which is the *fn1* to look for. It takes an optional second argument which is the *data-base* to look in. For example the call

```
<SPEC-FIND "MAIL">
```

might print:

```
MAIL SAV53 8  140
1MAIL SAV53 8  360
```

4.2.2.3. Other Functions

<DELETE *file:string data-base:string*>

allows the user to delete a 'file' from a data base. It takes the same type of 'file' specification that FIND-FILE takes. The 'file' you specify will be deleted the next time the demon that maintains the data base runs.

<GET-FILE *file:string output:string data-base:string*>

allows the user to retrieve a 'file' from the data base. It takes two arguments. The first is the 'file' specification of the *file* to retrieve out of the data base and the second is the *output* file you wish to copy it to.

<STATUS>

gives the information about the state of the data bases. It tells the number of 'files' and the amount of free storage in each data base. STATUS takes no arguments.

4.2.3. Using DBMAIN

There are several ways to use DBMAIN. It can be used by typing

:DBMAIN *function arg1 ... argn*

to DDT. The *jcl-line* is of the form *function arg1 ... argn*, where *function* is the name of the function to be used. For example

```
:DBMAIN FLIST "FIXUP"
```

will list the free storage block for the "FIXUP" data base. DBMAIN will kill itself after finishing and can be killed earlier by typing ↑S.

The *jcl-line* mentioned above can be modified to allow output to be routed to a file. This can be done by preceding the normal *jcl-line* with a string specifying the file name of the output file.

```
:DBMAIN "LISTOF SAVS" CLISTF
```

will produce a listing of the files in the SAV data base and will print this information to the file "LISTOF SAVS".

4.2.4. Garbage Collection

One problem of the MDL Pure-mapping Library is that many useless SAV and FIXUP 'files' remain as new revisions of user programs are created. To alleviate this problem there is a garbage collection system for the data bases.

The major goal of this scheme is to determine which 'files' in the data bases are no longer useful. To do this all files in the system are scanned to see what SAV files are still pointed to (*not* including those pointed to only from within ITS archive files). A SAV 'file' can be pointed to from FBIN files and SAVE files. A SAVE file contains pointers in its PURVEC (Pure VECTOR). All FBIN files should begin with something of the form

'<PCODE *file:string*>

where *file* is the name of the SAV 'file' associated with this FBIN. If an FBIN has more than one SAV 'file' associated with it then there can be several PCODE FORMs at the beginning of the file. For purposes of garbage collection, this FORM (or FORMs) *must* be retained whenever an FBIN file is edited. If these PCODE FORMs disappear, their pointers to the SAV 'files' will go with them, and the SAV 'files' might be garbage collected.

Garbage collections proceed by looking at every file on the disk, building a list of all 'files' pointed to. The program then examines the data bases and any 'files' which are not pointed to are deleted.

It is possible that deletions can fragment the free area in the data bases. If compaction becomes necessary,

there exists a routine to do in-place compaction of the data bases.

4.2.5. Internal Structure

The "SAV" and "FIXUP" data bases have similar formats. The 'files' in the data base are pointed to by entries in what is essentially a hash table. Associated with each data base is a main 'directory' (the hash table). This 'directory' is located in the first 1024 words of the file. This main 'directory' points to other 'directories' in the data base (the hashing buckets). Each of these 'directories' is 1024 words long. The first 'file' name is used to determine which 'directory' the 'file' is on. The structure of the main 'directory' is as follows.

word 0/ number *n* of entries in the main 'directory'
words 1-n/ block number of each 'directory'

There can be up to 1023 'directories' and each of these can contain approximately 500 'files'. This provides a virtually unlimited 'directory'.

Word 0 of each 'directory' gives its length in words. From Word 1 on are 'directory' entries. All entries have the same two word format. The first word contains the the first 'file' name in SIXBIT. The second word contains the following fields:

length of the 'file' in blocks (a block for a SAV 'file' is 1024 words long while a block for a FIXUP 'file' is 256 words long) (bits 1-6)

version revision of MDL this 'file' belongs to (bits 8-17)

block in the data base where this 'file' starts (bits 18-35)

The 'directories' are sorted by strict numerical order (e.g., AAA SAV53 comes before 1AAA SAV53).

Each data base contains a free storage table. This table occupies the second 1024 words of the data base. The first word of the table is the number of entries in the free storage table. The remaining entries define areas of free storage. These are of the form

length, , *block*

where *length* is the number of blocks for this free area, and *block* is the block number at which it starts.

There are two major differences between the "SAV" data base and the "FIXUP" data base. The first deals with block sizes. In the "SAV" data base the block size is 1024 words. In the "FIXUP" data base the block size is 256 words. This smaller size allows for more compaction of these small 'files'.

The second major difference is that while there can be many versions of the same 'file' in the "SAV" data base (e.g. NCODGE SAV53 and NCODGE SAV54), there can only be one version in the "FIXUP" data base.

This will be the FIXUP 'file' most recently added. The corresponding SAV 'file' for this FIXUP 'file' should exist to allow the SAV file to be updated for future MDL revisions.

5. The Compiler

The purpose of the MDL compiler is to transform interpreted MDL code into assembly language. The compiler comes in several incarnations for various purposes.

PCOMP is a program which runs the 'installed' compiler -- that is, the one which is most debugged, supported, and otherwise official. The 'P' stands for 'purified,' incidentally.

NPCOMP is a program which runs a newer, less well-debugged compiler, if there is one. NPCOMP is often where development work of one sort or another is being debugged.

The 'Batch Compiler,' often called COMBAT, though strictly speaking the name refers to a different program (see section 5.2) is a program that compiles, at night, those compilations that have been queued for it.

The remainder of this chapter describes the specifics of interaction with the compiler, including a section on its internals.

5.1. Interfacing to the Compiler

The operation of the MDL compiler is controlled by a few very high-level functions and a sometimes bewildering array of ATOMs whose values are switches and data. This section will describe each such ATOM and its use. The reader should bear in mind that in the normal case he will be using COMBAT to set up his compilations and thus will not have to deal directly with these ATOMs and calls.

5.1.1. Compiler Functions

<COMPILE *source:function-or-list output:channel*>

is the lowest level call to the compiler. It compiles exactly one FUNCTION (or a LIST of them) and prints the generated code on the CHANNEL given as the second argument. COMPILE is used primarily for compiler debugging.

<FILE-COMPILE *input:string output:string*>

FILE-COMPILE attempts to provide a convenient interface between the user and the compiler. The user simply gives FILE-COMPILE the name of an input file, and it can do all the rest. The user may specify other information about output files, compiler modes, etc., but if he doesn't, reasonable assumptions are made.

FILE-COMPILE works in the following way. First it reads in the input file and collects into a LIST the names of all of the defined FUNCTIONs that it finds. It sorts this LIST based on which FUNCTIONs call which other FUNCTIONs. The FUNCTIONs which call no other FUNCTIONs are at the beginning of the LIST, followed by those that only call FUNCTIONs that call no other FUNCTIONs, and so on. Groups of FUNCTIONs that are mutually recursive are collected in LISTs subordinate to the main LIST.

Each FUNCTION will produce a separate RSUBR. COMPILE is called successively on each member of the LIST of FUNCTIONs. LISTs of mutually recursive FUNCTIONs are also passed to COMPILE.

After each FUNCTION or LIST of FUNCTIONs is compiled, the resulting RSUBR is written into a temporary file to enable more convenient crash recovery. This file is written in such a way that, no matter when the system crashes, the contents of the temporary file are guaranteed to be in a consistent state.

When all is compiled, FILE-COMPILE writes out an output file which is identical to the input file except that all FUNCTIONs have been replaced with their compiled counterparts. If any of the FUNCTIONs did not compile due to programmer errors or compiler bugs, those FUNCTIONs are left unchanged in the output file.

During its operation, FILE-COMPILE maintains a "RECORD" file which contains all of the messages, warnings and error messages produced by the compiler. It may optionally produce a listing of the object code produced, in MDL assembler format. This is primarily useful for compiler debugging. (Note that a somewhat less complete listing may be made at a later time. See section 7.3.)

On ITS, FILE-COMPILE usually runs as a demon called COMBAT ZONE. In this case another interface called FCOMP resides above FILE-COMPILE. This interface reads files that are compilation specifications and passes them to FILE-COMPILE.

<FCOMP %.INCHAN *input-file output-file*>

As most compiler usage is based on COMBAT plan files, FCOMP is the most-seen driver of the compiler. (Note that the % in front of .INCHAN causes the CHANNEL the PLAN file is being read from to be passed as one argument to FCOMP.)

<STATUS>

is an informational function; it tells how far the compilation of a given group has progressed, which FUNCTION is being worked on, and how many FUNCTIONs remain to be compiled. It also prints the accumulated real time and cpu time since the beginning of the compilation. Obviously, you must ↑G the compilation to use it, but see section 8.3.

5.1.2. Compiler Switches

The calls to the various compiler drivers are rather short, for the simple reason that the controlling information is passed to the compiler as the LVALs of a set of ATOMs.

<SET DEBUG-COMPILE!- *boolean*>

(by default FALSE) causes the compiler to generate extra information about what it's doing. This information is in the form of 'warnings' produced when the compiler was forced to generate less than optimal code. For example, invocations of the arithmetic SUBRs can be open-compiled if the variables used can be determined to be exclusively FIXes. The debugging compiler will warn you if it is forced to resort to less efficient arithmetic calls.

<SET PRECOMPILED!- *file:string*>

Often, a file of FUNCTIONs has been compiled before, and now only a few FUNCTIONs have been updated and need to be compiled again. Most of the file is already correctly compiled; it is quite wasteful to recompile the entire thing. If a PRECOMPILED is given, the file is loaded before compilation; any functions which have corresponding RSUBRs in the precompilation, and which are not on the REDO list, are not recompiled. It is appropriate to specify the temporary file as a precompilation if your previous compilation was interrupted by a system crash.

<SET REDO!- *list-of-atoms*>

REDO is a LIST of FUNCTION names to be recompiled, regardless of whether or not they are compiled in the precompilation. In conjunction with PRECOMPILED and PACKAGE-MODE, REDO allows compilation of precisely those FUNCTIONs which have been changed since the last compilation. Note that COMBAT will set up these values more-or-less automatically in most situations.

<SET PACKAGE-MODE!- *string*>

This should be the name of a PACKAGE, which is assumed to be the PACKAGE being compiled. FUNCTION names in the REDO LIST will be looked up in the appropriate PACKAGE OBLISTs if this flag is set, thereby saving some typing of trailers.

<SET TEMPNAME!- *file:string*>

The compiler writes intermediate results to the temporary file, which is normally the file "*sname*;*fnm* >" on ITS, where *fnm* is the first name of the input file. It is rarely (if ever) necessary to change that default.

<SET SOURCE!- *file:string*>

Setting this switch causes the compiler to write out the assembler input it generates. This is sometimes useful for compiler debugging. On ITS, such output normally goes to "*sname*;*fnm* SOURCE", where *fnm* is the first name of the input file.

<SET SPECIAL!- *boolean*>

The compiler normally assumes that variables which aren't declared SPECIAL aren't SPECIAL. This means that they will be available only to the RSUBR in which they are declared: SPECIAL variables are bound on the control stack, just as all variables are in interpreted code. If this flag is T (by default FALSE), all variables will be assumed to be SPECIAL unless declared otherwise. This is analogous to SPECIAL-MODE being

SPECIAL, and it is not recommended that any code be written using this convention.

<SET EXPFLOAD!- *boolean*>

If true, FLOADs in the file being compiled will be expanded at load time: what was FLOADed before will be treated as part of the file. EXPFLOAD is examined by GROUP-LOAD, and not the compiler itself. The default is FALSE.

<SET EXPSPLICE!- *boolean*>

If true, objects of type SPLICE (primtype LIST) which are encountered in the course of EVALing the forms processed by GROUP-LOAD will be spliced directly into the group; it is therefore a lot like EXPFLOAD. EXPSPLICE is examined by GROUP-LOAD, and not the compiler itself. The default is therefore FALSE. Its only known use has been to make functions at load time and have them compiled.

<SET CAREFUL!- *boolean*>

Defaults to T. If FALSE, the compiler will omit most of the bounds-checking code it normally generates for NTHs, PUTs, and so on. This obviously will make the compiled code run faster, but also makes debugging the compiled code nearly impossible.

<SET REASONABLE!- *boolean*>

Defaults to T. If FALSE, the compiler will generate reasonable code only if *everything ever* called from the functions being compiled is loaded into the compiler. A call to a function not loaded produces an EVAL of a FORM, thereby ensuring that such constructs as "CALL" in the called function will work correctly. This is admittedly pretty unreasonable (if not paranoid), whence the name of the switch.

<SET GLUE!- *boolean*>

Defaults to T. If FALSE, the compiler will not generate GLUE bits. As you always want GLUE bits, there is *no* reason to *ever* change this.

<SET MACRO-COMPILE!- *boolean*>

Defaults to FALSE. If non-FALSE, the compiler will compile MACROs into RSUBRs. This doesn't change anything produced by macro expansions, but does cause the expansion to speed up. Since the compiler expands the macro and then compiles the expansion, this is rarely useful.

<SET MACRO-FLUSH!- *boolean*>

Defaults to FALSE. If non-FALSE, MACROs which appear in the file being compiled will not appear in the resulting NBIN. This saves space, at the expense of making debugging harder.

<SET MAX-SPACE!- *boolean*>

Defaults to FALSE. If non-FALSE, the compiler flushes from core most of each RSUBR once it has been compiled; only the DECL is needed to help compile other functions. Since the entire RSUBR is written out in the temporary file, no information is lost. This can, for compilations which are too large, result in considerable improvements in speed, primarily because more space is available in the MDL and less time is

spent in the garbage collector.

<SET HAIRY-ANALYSIS!- *boolean*>

Defaults to T. If this is not set, the compiler will not perform the complex type checking it usually does. If HAIRY-ANALYSIS is FALSE, the code will be generated faster, as type-analysis is expensive, but will not execute as fast.

5.2. COMBAT

The usual method of dealing with the compiler is through the program COMBAT, whose specialty is the preparation of 'plan files' to be loaded by the compiler. COMBAT is a program which knows about each of the previously described compiler switches and the interactions among them. It has an easy-to-use interface, an ability to store commonly used 'plan files' as *compilation types*, and in general is designed to make using the MDL compiler a less-cumbersome task.

5.2.1. User interface

COMBAT's user interface is patterned after, though not identical to, a CALICO interface [1]. In particular, it expects in response to any given prompt a particular type of input from the user, which may be a file name, a 'symbol', or text. Ordinarily, the type of input expected is indicated by the 'syntactic prompt' which follows the normal prompt; this is one of '(FILESPEC)', '(SYM)', and '(TEXT)'. The 'Toggle verbosity' compilation type turns the printing of the syntactic prompt on and off, and causes a tailor file to be written out when used.

A number of special characters are defined for any of these types of input

↑@: Clears the input buffer, as in MDL.

↑D: Redisplays the input buffer, as in MDL.

↑L: Clears the screen and redisplays the input buffer, as in MDL.

↑G: When given as the first character of an answer, allows one to get the answer from a user-defined type. See the section on tailoring.

↑Q: Has special effects when a compilation plan is being made (see below). See also the section on file name input.

↑R: Causes COMBAT to 'back up'. Typically this means go to the previous question asked, but in certain modes it may have a slightly different effect. When a MUDCOM is running, this kills it and backs up to the last question asked.

↑S: Abnormally ends whatever is being done, and returns to the 'Type of compilation' question. If a MUDCOM is running, it will be killed. When a long compilation plan ('How to run' is 'Many') is being

made, the portions already made will be saved. See the 'Flush many' compilation type.

?: When given as the first character of an answer, this causes a more detailed description of what is expected to be printed, along with the current default and how to obtain it.

\: This quotes whatever character follows it, including DEL, ESC, etc. It does *not* have the effect of quoting strange characters in file names; see the section on file name input. \, used as a quote character, never echoes, and cannot be rubbed out.

In addition, when the syntactic prompt is (SYM), ↑F is useful (see below).

5.2.1.1. Symbolic input

If you are familiar with CALICO, this section can probably be skipped. When entering symbolic input, one need only type the characters required to uniquely specify the desired choice: the interface will complete the response, and in addition can display the available choices at any point.

SPACE completes the response as far as it can. If the response is uniquely specified, it will be displayed in its entirety, followed by '!'; if more than one choice is still possible, then the portion of those choices which is unambiguously specified will be displayed, followed by '&'. For instance, if 'Expand floads' and 'Expand splices' are among the choices, and 'Ex SPACE' has been typed, 'Expand &' will be displayed if the 'Ex' reduces the choices to those two.

In some cases, if SPACE is the first character typed, it will select the default (first) choice and terminate.

When ↑F is typed, all remaining choices will be displayed.

To terminate responses in this mode, either ESC or CRLF may be used. In either case, the current response is completed as far as it can be. If only one choice then remains, the answer is terminated and the single choice will be used. If more than one choice is possible, it is just as if SPACE had been typed.

Typing ESC or CRLF before any other characters have been entered causes the default answer to be used.

5.2.1.2. File names

File names are expected in the standard *dev*:*sname*;*fname1* *fname2* format on ITS; on Tenex/TOPS-20, standard file name recognition is used. Typically, typing simply ESC or CRLF answers 'no' to the question, while SPACE ESC says 'use the default'. In certain special cases ('Input file' and 'Output file'), when some answer to the question is imperative, the default will be used in either case. File names should *not* be surrounded by quotes in this mode; they are not MDL STRINGs!

It is rather painful to get funny characters (such as SPACE) into file names. When the file-name parser sees a ↑Q, it uses the following character in the name being generated regardless. Unfortunately, the ↑Q must be quoted to get it past the reader, since it has special effects in the normal case. Thus, the file name given to MDL as "TAA; FOO >" has to be typed to COMBAT as TAA;\↑Q FOO >.

5.2.1.3. Text

Text is just that: *relatively* arbitrary characters, terminated by ESC. Since CRLF is allowed in text, it does not terminate input. Text type input is used in a number of cases where it isn't quite appropriate, such as the 'Redo list' and 'Package mode' questions. If it is known that the expected response is a LIST or STRING, as in those cases, the appropriate brackets or quotes should *not* be typed.

5.2.2. Combat Questions

This section discusses the questions that can be asked of the user during the preparation of a COMBAT plan file, which is FLOADed by the COMBAT demon or by PCOMP to effect a compilation. The perceptive reader will notice a strong resemblance to section 5.1.2, in which the switches relevant to the compiler are listed. Questions asked by the pre-existing compilation types ('Verbose' and 'Short') are so indicated. All questions are available in user-defined compilation types (see section 5.2.5).

'Sname': sets the default directory for questions that want a file name as an answer; also causes the FORM <SNAME *sname*>, where *sname* is the answer given, to be included in the plan. This sets the default directory for files referenced by the compiler; it also causes the temporary file (see below) to go to the *sname* directory.

'Use new compiler?' (Verbose and Short): specifies whether the 'new' compiler or the 'old' compiler should be used. Often, when there is only one compiler, this question will not be asked. If answered affirmatively, it causes the FORM

```
<OR <GASSIGNED? EXPERIMENTAL!-> <NEWCOMP!->>
```

to be included in the plan. This FORM will load a new compiler on top of the old if necessary.

'Debugging compiler?' (Verbose): causes DEBUG-COMPILE!- to be set to T, which causes the new compiler to generate extra information about what it's doing. This currently is asked only if the new-compiler question is answered affirmatively.

'Input from' (Verbose and Short): the file to be compiled. This appears in two places in the plan: as

```
<SETG COMBAT!- input-file>
```

and in the call to FCOMP described below.

'Output to' (Verbose): the file name to be used for the NBIN. The default is the input file name, with NBIN as the second file name instead of whatever it was for the input. This completes the call to FCOMP that ends every plan:

<FCOMP %.INCHAN *input-file output-file*>

This call is what actually invokes the compiler.

'Precompilation from' (Verbose): specifies a file containing a previously compiled version of the input file. Any FUNCTIONs which have corresponding RSUBRs in the precompilation, and which are not on the 'Redo' list, are not recompiled. It is appropriate to specify the temporary file as a precompilation if your previous compilation was interrupted by a system crash. Sets PRECOMPILED!-.

'Compare with' (Verbose): This question is asked only if a precompilation file is specified. If answered affirmatively (user types either SPACE ESC or a file name) MUDCOM (see section 8.1) will be run with *jcl* of the input file name, and the file name provided here (the default is as for precompilation), plus some extra stuff specified below. If 'FOO NBIN' is given here, then MUDCOM will look for the newest revision of FOO which was created before the NBIN. MUDCOM determines which FUNCTIONs in the file have changed and therefore need to be recompiled. It also determines whether the file is a PACKAGE, and answers the 'Package mode' question appropriately. It is therefore not usually necessary for the user to answer the 'Redo' and 'Package mode' questions directly.

'Check macros?' (Verbose): asked only if 'Compare with' is answered affirmatively. This adds '/M' to the *jcl* passed to MUDCOM, which causes it to check for MACROs and MANIFESTs which have changed: if a FUNCTION uses a MACRO or MANIFEST which has changed, the FUNCTION will be listed as changed. MUDCOM does not normally check for this.

'Extra JCL:' (Verbose): asked only if 'Compare with' is answered affirmatively. Whatever is supplied here will be passed to MUDCOM as *jcl*, before the files to compare. This can be used to load macro files; see section 8.1.

'Redo' (Verbose): asked only if a precompilation file was given. Takes a bunch of FUNCTION names, which will be recompiled. Note that the names supplied here will be appended to the list returned by MUDCOM, if any, and that duplications in the list are ignored. Sets REDO!-.

'Package mode' (Verbose): asked if a precompilation file was given and MUDCOM was not run (MUDCOM will set this if run). This should be the name of a PACKAGE, which is assumed to be the PACKAGE being compiled. FUNCTION names in the 'Redo' list will be looked up in the appropriate PACKAGE OBLISTs if this flag is set, thereby saving some typing of trailers. Sets PACKAGE-MODE!-.

'Temporary file to': The compiler writes intermediate results to the temporary file, which is normally

"*sname*;*fname1* >" (on ITS)
"<*sname*>;*fname*.TEMP" (on Tenex/TOPS-20)

You may change that by answering this question; there is rarely a good reason to do so. Sets TEMPNAME!-.

'Source file to': The compiler can be caused to write out the assembler input it generates by answering this question. Assembler output normally goes to

"*sname*;*fname*1 SOURCE" (on ITS)
"<*sname*>;*fname*.SOURCE" (on Tenex/TOPS-20)

which is the default for this question; another name may be provided if desired. Sets SOURCE!-.

'Special?': The compiler normally assumes that variables which aren't DECLed SPECIAL aren't SPECIAL. If this flag is T (defaults to FALSE), all variables will be assumed to be SPECIAL unless declared otherwise. Sets SPECIAL!-.

'Expand floads?': (Verbose) If true, FLOADs in the file being compiled will be expanded at load time. Sets EXPFLOAD!-.

'Expand splices?': If true, objects of type SPLICE (PRIMTYPE LIST) will be expanded and inserted into the group. Sets EXPSPLICE!-.

'Careful?': (Verbose) By default T, but if FALSE, the compiler will omit most of the bounds-checking code it normally generates for NTHs, PUTs, and so on. This obviously will make the compiled code run faster; it also makes debugging the compiled code nearly impossible. Sets CAREFUL!-.

'Reasonable?': By default T, but if FALSE, the compiler will generate reasonable code only if everything you call from the functions being compiled is loaded into the compiler. Sets REASONABLE!-.

'Glue?': By default T, but if FALSE, the compiler will not generate GLUE bits. There is *no* good reason to *ever* answer this. Sets GLUE!-.

'Macro compile?': By default FALSE, but if true, the compiler will compile MACROs. Sets MACRO-COMPILE!-.

'Macro flush?': By default FALSE, but if true, MACROs which appear in the file being compiled will not appear in the NBIN. Sets MACRO-FLUSH!-.

'Max space?': By default FALSE, but if true, the compiler flushes from core most of each RSUBR once it has been compiled; only the DECL is needed to help compile other functions. This can, for compilations which are very large, result in considerable improvements in speed. Sets MAX-SPACE!-.

'First things to do', 'Things to do' (Verbose), and 'Last things to do': It frequently is necessary to perform some actions before a compilation can be run: definitions files must be loaded, special environment setup might have to be performed, and so on. All three of these questions are designed to allow that: whatever you supply is put out after everything else in the plan but before the call to FCOMP. There are three questions, instead of one, to allow some things to be specified in a tailored compilation type, while others are provided at compile time, or possibly from another tailored type. The three questions do not depend on each other; they are asked in the order given here, and the answers appear in the plan in the same order.

5.2.3. Requesting Compilations

The first question asked by COMBAT is 'Type of compilation'. In addition to a number of special possibilities described later, there are two answers to this question (in addition to any provided by the user

through the tailoring facility) which request pre-defined tailored compilation types. These are 'Verbose' and 'Short'.

'Verbose' causes all the normal questions to be asked: 'New compiler?', 'Input file', 'Precompilation', switches, 'Things to do', and so on. 'Short', on the other hand, defaults the answers to all questions except 'New compiler?', 'Input file', and 'How to run'.

When requesting a compilation, one may type ↑Q at any time. This has the same immediate effect as an ESC, but in addition causes all questions between the one just answered and the 'Things to do' question to be defaulted. This is particularly useful in the 'Verbose' sequence of questions.

If 'Many' was given as 'How to run' for a previous compilation request, and the resulting plan has not yet been written out, subsequent plans will be appended to it. Using 'Many' will sometimes effect a major savings of time if several compilations wish to perform the same environmental setup; if they USE many of the same PACKAGEs, for example. When using 'Many' in combination with predefined compilation types, it is useful to remember that whatever is specified under 'Things to do' may end up being performed for each plan. You might modify your compilation types to reflect this, or alternatively, edit the plan file produced by COMBAT to remove redundant operations.

The *only* way to get rid of the 'Many' plan is to answer 'Many flush' to the 'Type' question. Typing ↑S or answering 'Abort' to the 'How to run' question will abort the current portion of the 'Many' compilation, but not the whole thing.

If 'Many' was mistakenly given as 'How to run', and you don't wish to destroy the plan you have generated, it is possible to (in essence) go back to the 'How to run' question by answering 'Many print' for the compilation type. In this case, you are *not* back in the plan-making loop; ↑R acts just like ↑S.

↑R, here, backs up to the last question asked. There are two qualifications. First, if ↑Q has been typed, then it backs up to the last question that would have been asked if ↑Q had not been typed. Second, the four questions 'Precompilation', 'Compare', 'Redo', and 'Package mode' are treated as a group: if the 'Package mode' question has not yet been answered, it is possible to back up normally; but once that question has been answered, backing up to it will go to the first member of the group, 'Precompilation'.

↑G allows one to obtain the answer to the current question from any user-defined compilation type. It requests a type name, and uses the answer or default supplied therein, printing the information so obtained. The ↑G must be typed as the first character of the answer for this to occur. This allows one to use parts of a

defined type without either using the type itself or altering it for the occasion. For 'Text' type input (such as 'Things to do'), the string is placed in the input buffer but not completed, so it may be edited before an ESC is typed. See also the 'Xerox type' command.

Note that there is a distinction made between 'Compare' and 'Redo'; the former causes a MUDCOM to be run, and the latter asks for the names of FUNCTIONs to be recompiled. It is possible to do both, in which case the two groups of FUNCTIONs are appended to form the 'Redo' list for the compilation. Note also that if a MUDCOM has been run, the 'Package mode' question will not be asked, since the answer is supplied by the MUDCOM. Either ↑R or ↑S may be used to kill a running MUDCOM.

One of the responses to the 'How to run' question is 'Abort', which returns directly to the 'Type of compilation' question without writing out a plan, starting up a PCOMP, or anything else. Its effect is exactly that of a ↑S. In particular, if you are making a long plan, only the portion just completed, *not* the entire compilation, will be aborted.

It is also possible at the 'How to run' question to supply an answer to any of the compilation questions (Input file, etc.). The 'Question' response asks for the name of a question, then asks that question. Any number of questions can be asked in this manner, one at a time. This is particularly useful for filling in the blanks left by a 'Short' type compilation, or by user-defined compilation types.

When a compilation request has been finished, COMBAT normally loops back to the "Type of compilation' question, but changes the default from 'Verbose' to 'None' (meaning 'Quit'), unless another compilation may reasonably be expected. Thus, one may leave by typing a single ESC.

It is possible to modify COMBAT's behavior such that it either kills itself after finishing the compilation plan, or loops back with 'Verbose' as the default for the "Type of compilation' question.

COMBAT first decides whether a long compilation plan is being made; if so, the default remains 'Verbose.' If not, it then examines the current compilation type: if 'Another compilation?' has been set to 'Yes', the question will be asked with default 'Verbose'; if it has been set to 'No', COMBAT will kill itself; if to 'Ask', further consideration is required.

If the user is in 'Multiple' mode (the 'Multiple' compilation type), the type of compilation will be asked with the 'Verbose' default. Otherwise, COMBAT examines the state of two tailorable switches, set by the 'Another compilation?' compilation type. If 'Another compilation?' has been set to 'No', COMBAT will die; if to 'Yes', the type question will be asked with default 'Verbose'; if to 'Ask', the type question will be asked

with default 'None'. Normally this is 'Ask'.

Note that 'Another compilation?' is like 'Toggle verbosity' in that it will have no effect unless user-defined compilation types exist.

5.2.4. 'How to Run' Options

There are four options available when answering the 'How to Run' question which determine where your plan file will be written and when the compilation it specifies will be run.

'Pcomp' places the plan file on the <SNAME> directory, and names it "PCOMP >". Additionally, COMBAT will start a PCOMP (or NPCOMP, as appropriate) process if it is exited after writing a PCOMP file. 'Pcomp' is the standard method for running a compilation in one's own process.

'COMBAT' writes the plan file to "COMBAT;PLAN >". The COMBAT demon successively compiles all such plans at night, informing the persons who submitted them of the result.

'Waste' is like 'COMBAT', except that the plan is written to "COMBAT;WASTE >". The 'waste' queue is only run after midnight, which is usually sufficient for those who are doing 'overnight' compilations. 'Waste' is the answer used by default for 'How to Run'.

'File' places the plan file on the <SNAME> directory, and names it "PLAN >". This means that it will not be run until you explicitly load it into a compiler process.

5.2.5. User Tailoring

It is often the case that a particular file is compiled quite often, or that some sequence of actions must be performed as the 'Things to do' before many compilations. COMBAT allows the user to define his own 'Compilation types', each of which specifies exactly those questions which should be asked and the answers for those which should not. For example, one could have a type named 'Esign', which says that the input file is always "SEND;ESIGN >" and in addition provides for the FLOADing of two files in 'Things to do'. Further, since most questions are defaulted, one might choose to answer only those questions which are interesting, such as 'Precompilation'. It is also possible to supply a default answer for a question which will be asked.

In addition, there are some questions which are not asked by the 'Verbose' compilation type, but which nevertheless are available to user-defined types. These are: 'Macro compile', 'Macro flush', 'Max space', 'Expand splices', 'Special mode', 'Glue', and others.

One can select any of one's own defined compilation types as an answer to the 'Type of compilation' question, just like 'Verbose' and 'Short'. Except that the questions asked may differ, user-defined types are

identical to the predefined types.

5.2.5.1. Tailor files

User-defined types are saved (and loaded) from the file "*sname*;%COMBT TAILOR". It is possible to load other tailor files, but the "%COMBT" file in *sname* is loaded during startup. Tailor files are quite similar to MDL GC-DUMPed files and thus cannot be edited other than with COMBAT.

5.2.5.2. Create type

This special compilation type requests a name for the type being made, then enters a loop with the prompt 'Question'. One may choose any of the available questions, and either supply an answer or (by default) request that the question be asked when a compilation of this type is being submitted. Note that only the 'How to run' and the following 'Type of compilation?' questions will be asked unless others are explicitly supplied; but one may supply answers to 'How to run' when creating a type.

In this mode, ↑R will return to the 'Question' loop if one is about to supply an answer; otherwise, it returns to the 'Type of compilation' loop, aborting the type creation.

↑G behaves exactly as it does in the normal loop. To indicate that one is finished, one should answer 'Finis' to the 'Question' prompt. It is possible to supply several different versions of the answer to a particular question: the last one given will be used. One may wish to default a particular question, after specifying that it was to be asked or after supplying some different default. This may be done by answering 'Delete question' to the 'Question' prompt, whereupon one will be asked for a particular question to ignore. This question will then be completely ignored. Note that *all* interesting questions are initially in this state.

There is also a 'Set question default' 'Question'. This requests a question name, then asks the user to supply an answer. The question will be asked, with the default supplied. Thus default settings of switches can be changed, and one can supply a file name for the precompilation while still being asked whether precompilation is desired. Unfortunately, user-supplied defaults for 'Text'-type questions are used if ESC is answered; to get rid of the default, type SPACE ESC. Note that this is exactly the inverse of the convention for defaulting file names.

When 'Finis' has been typed, a new copy of one's tailor file is written out. This may, in combination with 'Load tailor' and 'Replace tailor', have undesirable side effects.

5.2.5.3. Print type

This requests the name of one of the types currently loaded, and prints out for it all questions which either will be asked when a compilation is being submitted or which have user-supplied defaults. If a particular question has been globally 'turned off' (such as the 'New compiler?' question, when there is no new compiler), an asterisk will be printed on the appropriate line to indicate that the information there is currently not used.

5.2.5.4. Delete type

This requests the name of one of the currently-loaded types, and deletes it. A new copy of the tailor file is written out, so all trace of the type will vanish when this command is used.

5.2.5.5. Alter type

This requests a type name, then becomes identical to 'Create type', except that some questions already have answers. Again, 'Finis' must be typed to leave the loop and cause the modifications to be filed; typing ↑R or ↑S will leave the loop, but the modifications will be forgotten.

5.2.5.6. Load tailor, Replace tailor

Both of these request a file name, defaulting to the last one used for either a 'Load tailor' or 'Replace tailor' command. Initially this is "*sname*;%COMBT TAILOR". 'Load tailor' appends the types defined in the specified file to those already loaded, while 'Replace tailor' first throws away those already loaded. The types defined in this way are not distinguished from those loaded from one's own COMBAT tailor file; in particular, using 'Toggle verbosity' or any of 'Create', 'Alter', and 'Delete type' will cause all the types currently loaded to be written out to the COMBAT tailor file. If, therefore, one has done a 'Replace tailor', one can easily lose all of one's own types in this manner. I.e., it is very easy to destroy yourself.

5.2.5.7. Xerox tailor

This requests the name of an existing user-defined type, and a new type name. The new type becomes an exact copy of the previously-existing type. This is particularly useful when one has several different types which do almost the same thing.

5.3. The Compiler (Internals)

The compiler's job is to take a MDL FUNCTION or group of FUNCTIONs and produce an operationally equivalent machine-language subroutine (RSUBR) using whatever information can be extracted from the source code and whatever additional information the user wishes to supply. The efficiency of the output code produced is directly proportional to the amount of information supplied by the programmer and inversely

proportional to the generality of the source program.

The information supplied by the programmer is usually in the form of optional data-type declarations (DECLs) and the use of programmer-defined data types (NEWTYPEs) that have built-in declarations. Unlike many programming languages, however, declarations are never required. The compiler will compile programs with no declarations at all, but the resulting output will not run as fast as with well-declared code.

The current compiler can achieve speed-up factors of anywhere from about 4 to 100. The factor of 4 represents the speed-up for a very general program with very poor declarations. On the other hand, the factor of 100 represents a program with a very narrow range of application that has very good (that is, restrictive) declarations. Typical programs can expect to achieve factors of 20-40.

5.3.1. How it Works

The compiler as it currently exists is really two distinct programs. GETORDER is basically an interface between files of MDL functions and the compiler. It is a relatively small program that reads in the file, sets up the various compiler switches, calls the compiler one or more times and writes out the final file of RSUBRs.

COMPILE itself is basically a compiler with three major and three minor passes. Pass 1 builds a model of the program, pass 2 analyzes each node of the tree and does data type analysis, pass 2.5 (minor) allocates stack space for variables and temporaries, pass 3 generates output code and two minor passes do final stack allocation and peep-hole optimization.

5.3.1.1. COMPILE and COMPILE-GROUP

There are two distinct modes of compilation available. They are simple and multiple. Simple compilation occurs when COMPILE is called with one FUNCTION. It simply compiles that FUNCTION and returns. Multiple compilation occurs when COMPILE is called with a list of FUNCTIONs. It compiles each FUNCTION into a separate RSUBR. It differs from multiple calls to COMPILE in that it sometimes partially compiles a FUNCTION out of order to determine its calling sequence and do argument type-checking. This behavior is necessary when compiling mutually recursive FUNCTIONs.

In all modes of compilation, COMPILE-FUNCTION is called to actually compile the individual FUNCTIONs. It calls the various compiler passes.

5.3.2. Modeling Pass

The first pass of the compiler takes the input FUNCTION and builds an expanded model of it. In the process of doing this, it produces a symbol table entry for every local variable bound and/or declared in the FUNCTION, any of its PROGs/REPEATs or MAPF/MAPR FUNCTIONs. It also produces the RSUBR DECL for the final output. Pass 1 also tries to decide if an internal entry (that is, an entry which can be called efficiently (see section 6.1)) can be used with this FUNCTION. If an internal entry turns out to be possible, Pass 1 generates an appropriate calling sequence for internal calls to use.

The model built by Pass 1 looks like the original FUNCTION with all of the nodes in the FUNCTION's structure replaced with objects of type NODE (a new type defined for the compiler). A node in the model may have anywhere from 5 to 30 elements. The 5 element node is for simple quoted objects like fixed-point numbers, ATOMs etc. The 30 element nodes are for major elements of the program such as the node for the FUNCTION itself and nodes for PROGs and REPEATs. The majority of the nodes are general SUBR nodes, which have 10 elements.

The Pass 1 structure is built in the following way. The top level program in Pass 1 generates a node for the entire FUNCTION. This node gets the following information put into it

1. A code specifying that this is a FUNCTION node.
2. The data type that this FUNCTION is declared to return (or ANY).
3. A LIST that will eventually contain the nodes comprising the body of the FUNCTION.
4. A UVECTOR of internal names for internal calls to this FUNCTION.
5. A symbol table for the variables declared and/or bound in this FUNCTION.
6. A list of entries in the symbol table specifying how the arguments are to be set up (whether they are optional, QUOTEd, TUPLE etc.).
7. The final RSUBR DECLs.
8. A specification of how to pass arguments to this FUNCTION when it is compiled (whether the arguments should be in registers or on the stack).
9. The number of required arguments and the total number of possible arguments.

In addition to the above information, slots exist in the node for additional information to be supplied by later compiler passes.

After the main node for the FUNCTION is built, the sub-nodes for the FORMs comprising the body of the FUNCTION are built. This is done by first dispatching to special Pass 1 code for the first element of the FORM. If no special code exists for this first element, a dispatch is made on the TYPE of the first element of the FORM (that is, ATOM, FIX, FUNCTION etc.). If no special code exists for either the first element or its TYPE, a general FORM node is built. In the case of an ATOM as the first element of the FORM, the normal lookup rules are invoked on the ATOM and it is dispatched again based on its value. ATOMs with no values either cause compilation warnings or are assumed to be RSUBRs (depending on compiler switch REASONABLE).

All FSUBRs (COND, AND, OR, FUNCTION, PROG, REPEAT, UNWIND, etc.) have special Pass 1 code and produce very specific nodes. Most SUBRs don't dispatch to specific code during this pass. The exceptions are things like MAPF, ILIST, GET etc., which have somewhat non-standard treatment of their arguments. (Actually, MAPF and MAPR don't treat their arguments non-standardly, but they are treated specially in Pass 1 so that the inner FUNCTION may be open compiled.)

As mentioned previously, all nodes have at least 5 elements. These are as follows:

1. A node type code.
2. A pointer to the parent node (if one exists).
3. A specification of the data type the node will generate.
4. A list of sub-nodes referred to as *kids*.
5. A name for the node, which may have different meanings for different nodes.

In addition, nodes other than nodes for QUOTEd objects have additional elements that are filled in during later passes of the compiler.

After Pass 1 all additional passes work on the model built during Pass 1. The original FUNCTION is no longer even considered.

5.3.3. Analysis Pass

During Pass 1, very little information is determined regarding the resulting data types of various nodes. Indeed, with the exception of nodes produced by quoted objects, structured objects which will produce code to build copies of themselves, and FUNCTIONs, PROGs and REPEATs with declared values, no type information is produced. Even in the cases where type information is produced during Pass 1, it is usually not as detailed as other passes would like. The Analysis Pass has the job of refining the result type of each

individual node based on various criteria

1. The declared types of the variables used in the program including GDECLs and MANIFESTs.

2. The known type transformations produced by various SUBRs. (For example, it is known that LENGTH always produces a FIX result.)

3. Some analysis of the context of the nodes within the program. (For example in the following code:

```
<COND (<AND <TYPE? .X LIST> <NOT <EMPTY? .X>>>
       <1 .X>)>
```

regardless of how X is declared, it is obviously a LIST when the EMPTY? is run, and it obviously is not empty when the <1 .X> is run.)

The Analysis Pass performs a standard depth-first left-to-right tree walk on the Pass 1 model. The main dispatch function during this pass is called ANA. It does an initial dispatch based on the node type of each node. Since most nodes are still considered 'SUBR nodes', most of the dispatches end up at the SUBR call analyzer. The SUBR call analyzer has two types of further dispatch available. First it looks in a table for SUBRs that are capable of being completely open-coded; if it finds an entry in the table, the analyzer for that SUBR is invoked. If this SUBR is incapable of being open-coded, ANA checks another table to see if this SUBR has an internal entry available. If it does, the node is changed from a SUBR node to an internal SUBR node. If both dispatches fail, another table is checked to see if the object type returned by this SUBR is known, and if it is the result is put into the SUBR node.

Most of the work done by the Analysis Pass happens when the first dispatch occurs and special SUBR analyzers are invoked. Generally speaking, these analyzers check to see if they know enough about their arguments to transform their nodes to an open-code specification. For example, an invocation of the SUBR REST only transforms to an open-code node if both the PRIMTYPE of the first argument is known at compile time and there are no SEGMENTs in the call to REST. If a special SUBR analyzer decides that it can't open-compile in this case, it either leaves the node as a SUBR node or transforms it to an internal SUBR node.

5.3.4. The Type Analysis Model

In addition to the model of the FUNCTION built in Pass 1, the Analysis Pass adds additional information to the model concerning the current states of local variables. As the analyzer plunges down into the tree, it tries to keep track of the current DECL of each variable. Specifically, there is a slot in each symbol table entry called CURRENT-TYPE. The analyzer updates that slot based on its current knowledge. A call to SET causes the CURRENT-TYPE slot to be changed to the analyzed type of SET's second argument. When multiple

control paths meet, the CURRENT-TYPE slots of a variable are OR'd together at the joining point.

Conditional control structure nodes for COND, AND and OR also maintain two lists of transient information. These are called TRUTH and UNTRUTH. They specify what information will be valid if the true or false branches are taken respectively. For instance, a COND clause compilation can assume that any TRUTH information generated in the predicate of the COND will be valid for the rest of the clause.

Some of the analyzers for the more widely used predicates have special code in them to add information to the current TRUTH and UNTRUTH values. These predicates include TYPE?, EMPTY?, LENGTH? and NOT.

Looping control structures pose additional problems for the type analysis model. The approach taken by the type analyzer is to build a copy of the current types of all variables before analyzing the loop structure. This copy of the local type information constitutes the assumptions currently in effect. After the loop analysis is compete, the assumptions are checked against the current state of the variables. If any of the assumptions have been violated, the assumptions are updated and the loop is re-analyzed.

5.3.5. Life-and-Death Analysis

The Analysis Pass also performs a life-and-death analysis on the local variables. This is done by assuming that the variable's value is dead at each LVAL node for that variable. If another LVAL node for this variable is discovered that is reachable from this one before any intervening SET nodes for this variable, the original node is updated to be alive. This life-and-death information is used during the Code Generation Pass.

5.3.6. The Variable Allocation Pass

The Variable Allocation Pass (VAP) is a relatively simple one. Its purpose is to allocate stack space for all of the variables bound in the FUNCTION, its PROGs and REPEATs and its MAPF/MAPR FUNCTIONs. There are various switches that control the manner in which this allocation is performed.

The most important switch specifies whether or not this FUNCTION needs a FRAME or not. The VAP always starts out assuming it does not need to build a FRAME. This assumption will be changed if it is discovered that externally accessible named ACTIVATIONs exist in the FUNCTION or any of its inner blocks (PROGs or REPEATs or FUNCTIONs) or if at any time it is discovered that the address of a variable cannot be specified as a fixed offset from the top of the stack. Whenever this assumption is changed, the VAP starts over again with the new assumption in affect.

Another switch that controls the behavior of the VAP specifies whether or not the stack slots for inner

blocks will be pre-allocated because the stack will be in a 'fuzzy' state when these blocks are running. The stack is said to be in a 'fuzzy' state when the number of slots currently being used cannot be determined at compile time. This usually occurs when a TUPLE is being constructed for a MAPF. For instance, in

```
<DEFINE F (X Z)
        <MAPF ,VECTOR <FUNCTION (Y) <==? .Y .Z>> .X>>
```

the elements of the VECTOR will be between the top of the stack and the location of variable Z. Even if F has a FRAME, the location of Y will not be known relative to the FRAME pointer at compile time. Therefore, the initialization code for F will pre-allocate the stack space for Y.

During the VAP, each symbol table entry gets its address field set based on where that variable will be on the stack. Also nodes for PROGs, REPEATs and MAPF/MAPR FUNCTIONs that have bound variables get additional information inserted in themselves. This information includes where the SPECIAL variables start and where the UNSPECIAL variables start.

5.3.7. The Code Generation Pass

The Code Generation Pass (CGP) is probably the most complicated of all the passes. Fortunately, the Analysis Pass has already refined the model so that the CGP can dispatch immediately to the special-purpose code generators. Besides building a list of assembly-language instructions as output, the CGP keeps track of the current state of the stack, the contents of the registers, the current state of variables (whether they are in registers or on the stack or both) and the contents of the temporaries.

The general dispatch routine during the CGP is called GEN. It takes two arguments: A NODE and a specification of where to leave the result. The second argument can be any of the following:

1. The ATOM FLUSHED, meaning that the code will be executed for effect rather than value.
2. The ATOM DONT-CARE, meaning that the caller of GEN is leaving the decision up to the specific generator as to where to leave the result.
3. An object of type DATUM which specifies a place for the type and value of the result to be left.

Type DATUM is of PRIMTYPE LIST and contains two elements, one for the type and the other for the value. The elements of a DATUM may take on a variety of values in different circumstances. These include:

1. A TYPE name. This can only occur in the type slot and it means that the type of the object is known at compile time and this is it. It indicates that the code generator need not put the type-code anywhere.
2. The ATOM DONT-CARE. This means that the caller doesn't care where the result for this field is

left.

3. The ATOM ANY-AC. This tells the generator to leave the result in any available AC.

4. An object of type AC. This tells the generator to force the result into a specific AC.

5. An object of type ADDRESS:C or ADDRESS:PAIR. Both of these specify addresses on the stack or in the interpreter.

6. An object of type OFFPTR. An OFFPTR has three fields: a DATUM, an offset (a FIX), and a PRIMTYPE. An OFFPTR tells the generator to leave the result in the word pointed to by the inner DATUM and offset by the offset.

If an element of a DATUM is ANY-AC or DONT-CARE, the generator is required to update the DATUM to reflect the actual location of the result. If the element is a TYPE, the generator may change it to an AC which means that it happened to end up with the TYPE in that AC.

The generators always return a DATUM specifying where the result was actually left, unless the caller wanted the result FLUSHED. There is one special DATUM that can be returned. It is the GVAL of the ATOM NO-DATUM and it means that the specified node will not return a value (that is, it is a RETURN or an AGAIN or something).

There are six objects of type AC in the compiler, corresponding to ACs 0, A, B, C, D and E. AC 0 is special since it can't be used as a pointer, and it always contains very transient information. It is never used to fill in an ANY-AC slot in a DATUM. The other five ACs are in the pool of available ACs. Objects of type AC have about ten different slots associated with them. They are used for finding available ACs and generating output code that uses them. The slots used in AC allocation are as follows:

1. ACLINK. If this is FALSE, the AC contains no temporary value for the current computation. Otherwise, it is a list of active DATUMs that contain it.

2. ACAGE. This is only used when the ACLINK is non-FALSE. It is updated to a higher number at each use of the AC and is used in an LRU algorithm when an active AC must be flushed.

3. ACRESIDUE. If this AC is currently equivalent to some local variables, this slot contains a list of the symbol-table entries for these variables. The symbol-table entries themselves have a slot called INACS that points back to the ACs that contain its type and/or value. They also contain a slot called STORED that specifies whether the only copy of the variable is in the ACs or it is also in memory.

4. ACPROT. This slot is a boolean saying whether this AC is protected or not. If the AC is protected, it can't be allocated for any reason. Protection is only invoked for very stretches of code.

5. ACPREF. This slot says that this AC deserves slightly preferential treatment. It means, all other things equal, don't choose this AC.

The AC allocation algorithm consists primarily of trying to find the best possible candidate when an AC is needed. The routine GETREG is used to find an available AC. First it rejects all ACs that are protected (if they all are protected, the compiler generates an internal error since this should never happen). If there are one or more ACs with their ACLINKs FALSE, GETREG will choose from among them. It will prefer ACs with no ACRESIDUE, that are numerically adjacent to another free AC (because some PDP-10 instructions destroy the next AC), and which do not have their ACPREFs on. If the AC chosen has an ACRESIDUE, code is generated if necessary to store any of the variables that are only in ACs.

If no AC exists with an ACLINK that is FALSE, GETREG finds the AC with the smallest ACAGE. Code is generated to store the contents of the AC in a temporary so that it is available. The DATUMs that were in the ACLINK are updated to indicate that they are now pointing to temporaries as opposed to ACs. Thus it is possible that a generator could need sub-results in ACs, and after causing one to be generated in an AC, find that while generating the second one the first slipped back into a temporary. The generator would then have to generate code to reload an AC from the temporary.

The CGP invokes various special-case optimizations by passing information up and down the tree as code is generated. The generators for conditional branching FSUBRs like OR, AND and COND employ a predicate generator whenever possible. This generator is like GEN except that it takes three additional arguments: a label to branch to, a flag saying whether to branch on truth or falseness, and a flag saying whether this predicate is being NOTed. The general predicate generator then looks at the predicate node to see if it can take the additional arguments for predicate generation. If it can, the general predicate generator just passes all the arguments down; otherwise it calls GEN and generates the additional testing and branching code itself. Currently AND, OR, COND, ==?, N==?, G?, G=?, L?, L=?, 0?, 1?, TYPE?, NOT, ASSIGNED?, MEMQ, LENGTH? and EMPTY? have special predicate code associated with their generators. Others may be added as the need develops.

Other optimizations are invoked by simply recognizing common patterns of MDL code. For instance, the compiler recognizes <SET X <+ .X 1>> as a PDP-10 AOS instruction and it generates very efficient code for <REST .X <- <LENGTH .X> 1>> by recognizing the pattern of code.

The compiler always takes advantage of as much knowledge as it has about the types generated by particular nodes to generate good code. This is especially the case when it is handling the code for NTH, REST and PUT in structures. It uses type information concerning the length of the structure and the amount being

RESTed for the NTH, REST or PUT, to figure out whether or not to generate bounds checks in the compiled code. It also uses information about the current type of the slot being read or written to decide whether not to read or write the type word. Obviously, a lot of this type information was the same information obtained during the Analysis Pass of the compilation.

Some code generation routines are capable of changing the order of generation of the sub-nodes. This is done to try to get the node requiring the most ACs compiled first so that it won't interfere with any AC requirements of the current node. This obviously requires that the commuted nodes have no interacting side effects.

6. Making It Run Faster

Once you have a working program, you will probably want it to run fast. The most obvious way of doing this is to compile it. MDL provides other ways to speed up code, chiefly by eliminating mediated subroutine calls, and by reducing the size of garbage-collected space.

Mediated subroutine calls (or 'MCALLs') are the standard method of function calling in MDL. They provide a great deal of information and control during program development and debugging, but the overhead of an MCALL is superfluous in debugged production programs. Consequently, several methods exist for removing this overhead.

A subtle impediment to increased speed in a production program is the amount of time devoted to garbage collection. As this is proportional to the size of the garbage collected space, it is advantageous to make that space as small as possible. One way to do this is to purify as many of the static data structures in the MDL as possible.

One by-product of the procedures mentioned above is that much of the resulting code and structure becomes pure and therefore shareable between many MDL processes.

6.1. GLUE

A facility exists to allow separately compiled and assembled RSUBRs to be 'glued' together. This makes calls between RSUBRs in the group much faster, as MCALLs are replaced by PUSHJs. The many instructions of an MCALL are replaced by the single PUSHJ, but the mediation provided by MCALL is lost: No FRAME is produced. GLUEing is accomplished by the concatentation of the code and reference VECTORs of the RSUBRs being GLUEd, which gives them a common 'frame of reference.'

Additionally, GLUE is interfaced with the compiler such that:

1. The RSUBRs can be run unGLUEd for convenient tracing and debugging. After debugging, they can be GLUEd together and run much faster.

2. An individual FUNCTION can be recompiled without the overhead of recompiling everything GLUEd to its RSUBR. After the recompilation, the entire set can be reGLUEd.

6.1.1. How to Glue

"GLUE" is a PACKAGE and it may be obtained by doing

```
<USE "GLUE">
```

The call to glue a group of RSUBRs and/or RSUBR-ENTRYs is:

```
<GROUP-GLUE group-name:atom
            substitute:boolean
            script:channel
            package:string-or-list
            survivors:list
            victims:list>
```

where:

group-name is an ATOM as returned by GROUP-LOAD, and it is the only required argument.

substitute is a flag; if it is true, the current RSUBRs and RSUBR-ENTRYs will be fixed so that they may still run in the current MDL. This is expensive but necessary if PRINTTYPEs or interrupt handlers are among the RSUBRs in the group. If the flag is FALSE or not supplied, the group must be GROUP-DUMPed and reloaded before use.

script if supplied and a CHANNEL is used by GROUP-GLUE to print out its progress through its task. Otherwise, GROUP-GLUE works silently.

package, if provided and non-FALSE, implies PACKAGE mode will be used. This argument should be a STRING specifying the PACKAGE that is being glued. In PACKAGE mode only the ENTRYs of that PACKAGE will be preserved and all RSUBR-ENTRYs associated with internal functions will be removed. This option can also be used by setting the ATOM PKG to the name of the PACKAGE. *Package* may also be a LIST of PACKAGE names, in which case the ENTRYs of all the PACKAGEs listed will be preserved.

survivors if provided indicates that SURVIVOR mode will be used. This argument should be a list of those RSUBR-ENTRYs to be preserved. All other RSUBR-ENTRYs will be flushed. This option overrides PACKAGE mode. This option can also be used by setting the ATOM SURV to the LIST of RSUBR-ENTRYs being preserved.

victims allows 'survivors' to be specified by default; that is, it is a LIST of those functions which should *not* survive after GLUE has run. This is sometimes more convenient to specify than explicit survivors.

There are two advantages to removing unneeded RSUBR-ENTRYs. The group is made smaller by the absence of the RSUBR-ENTRYs. Also the code for the group is reduced, as the code for handling MCALLs to those RSUBR-ENTRYs is removed. In general only the ENTRYs need to be kept for a PACKAGE. This can be done by specifying the PACKAGE using PACKAGE mode. SURVIVOR mode should be used if the user wishes to explicitly state which RSUBR-ENTRYs are to be kept.

6.1.2. GLUE as a Program

In addition to the "GLUE" PACKAGE, there is a program in which GLUE and PDUMP (see section 6.3) are preloaded. It will prompt for each of the usual arguments to GROUP-GLUE, permitting the user to conveniently GLUE (and PDUMP) several PACKAGEs in one session.

6.2. Glue Bits

GLUE is able to perform its transformations on compiled or assembled code with the aid of a data structure produced during assembly. This structure is called the 'GLUE Bits'. It is an association placed on the RSUBR by this FORM:

```
<AND <ASSIGNED? GLUE>
     .GLUE
     <PUT rsubr GLUE glue-bits:uvector>>
```

Thus if .GLUE is non-FALSE the association will be available to programs wishing to use it.

Internally, the GLUE bits consist of two bits for each word of code in the CODE element of the RSUBR, followed by words specifying calling information. For each INTERNAL-ENTRY in the code, there is a word giving the number of arguments it takes and the offset of the INTERNAL-ENTRY in the CODE UVECTOR.

The two bits for individual instructions are interpreted with the index field of the instruction as follows:

Bits 0 implies the instruction is uninteresting;

Index field (M) and bits 1 implies the instruction is a reference to the code itself (a jump, perhaps);

Index field (R) and bits 1 implies a reference to an impure slot of the RVECTOR (the compiler does not generate such references);

Index field (R) and bits 2 implies the instruction is an MCALL;

Index field (R) and bits 3 implies the instruction is a reference to a pure slot of the RVECTOR.

See section 7 for more details on the format of MDL Assembly code.

6.3. PDUMP

MDL provides a mechanism for sharing compiled programs among several MDL processes, and for dynamically moving the compiled code in and out of the virtual address space as space is needed in the interpreter. This mechanism is described in detail in section 4.2. This section describes how to convert a compiled program into a sharable version, known as an FBIN (Fast-BINary) version of the program.

First load the group-purifier,

```
<USE "PDUMP">
```

Next, GROUP-LOAD your group (or groups).

```
<GROUP-LOAD binary-file:string>
```

which returns the group-name of the group. This (and any other groups to be dumped together) is then passed to the pure-dumper:

```
<PDUMP group-name1:atom group-name2:atom ... >
```

This creates several files, only one of which you need be concerned with:

sname;*group-name1* FBIN

If given more than one group-name, PDUMP will create one FBIN file for each group, but only a single FIXUP and a single SAV file containing the fixups and code for all of the groups named. The FIXUP and SAV files are put on the "MUDTMP" directory and eventually are inserted in the pure code library, as described in section 4.2.

Alternative methods of PDUMPing are to specify that as an option in to the program GLUE (see section 6.1.2), or to use its preloaded PDUMP directly after exiting its READER with ↑S.

A warning about combining GLUE and PDUMP: if you attempt to PDUMP several groups that have been GLUEd together, you will lose. This is because the references to the 'group-RSUBR' will fall on the wrong OBLISTs.

PDUMP also produces a structure analogous to the GLUE bits (see section 6.2) produced by the compiler, but containing only information about the RVECTOR of the RSUBR, for the use of PURIFY (see section 6.5).

6.4. SUBRFY

SUBRification is a way of getting rid of many of the MCALLs which could not be practically removed using GLUE. If a FUNCTION is called by many separate groups, it is difficult to GLUE it to all the groups or to GLUE all the groups together.

What is really needed is to be able to allow something to be called with PUSHJ from separate groups without forcing it to be part of those groups. This is indeed the case with PUSHJ entries to MDL SUBRs (in the interpreter). A user can make his RSUBRs look like SUBRs in this respect.

SUBRFY takes a group, which must be in NBIN format. It purifies the RSUBRs and RSUBR-ENTRYs in the group and changes them so that they can be called with PUSHJ. It also produces a file, known as the 'preload' file, which can be used by the compiler to generate PUSHJs to the functions in the SUBRified group.

SUBRFY should be loaded before loading the group to be processed. The reason for this is that it guarantees that GLUE bits stay around. To load SUBRFY

```
<USE "SUBRFY">
```

You should then GROUP-LOAD the group. Your group should be GLUEd already, since SUBRFY does not GLUE the group together.

SUBRFY can then be called in the following manner:

```
<SUBRFY group:atom
        file-name:string
        output:channel>
```

where

group is the name of the group.

file-name is the name of the file in which SUBRFY should put the information for the compiler. This defaults to the name of the input file with second name "PRELOD".

output is an optional argument which specifies a CHANNEL on which to print information about SUBRFY's progress. The default is not to print anything.

The file produced by SUBRFY should be FLOADed for compilations where functions in the SUBRified group are called. This can be done by FLOADing it in the 'Things to do' part of a COMBAT plan.

Like purification, SUBRification changes the MDL. The only way to preserve the SUBRified group is to SAVE the MDL. Before SAVEing the MDL the "SUBRFY" PACKAGE should be removed. This can be done by doing a

```
<KILL-SUBRFY>
```

followed by a

```
<GC 0 T>.
```

SUBRFYing a group implies that the group is not going to change at all frequently, if ever. A new SUBRFYed SAVE file may be created at any time, and elements of the group may be recompiled. However, if the calling sequences of any of the functions in that group change, you invalidate any functions compiled using the 'preload' file for that group. In short, think twice before tying yourself down with SUBRFY.

6.5. Purification

A facility exists to permit the purification of MDL objects. Purified objects can be shared between MDL processes and also are not examined by the garbage collector. What follows is a description of how this facility can be used.

The purification facility in MDL is most useful in the creation of subsystems. Non-purified RVECTORs of RSUBRs and tables used by subsystems are kept in garbage collected space. This means that these objects, which will never become garbage, are examined at each garbage collection, slowing down the garbage collection process. Also, if two people are using the same subsystem, they cannot share the tables and RSUBRs kept in garbage collected space. By using purification these two problems can be alleviated.

To purify most objects the user can call the PURIFY SUBR. The object will be purified, and all references to that object in the MDL core image will be changed to point to the new pure object. This simple method cannot be used in the case of RSUBRs. Purification of RSUBRs is a several step process beginning with compilation.

6.5.1. Purifying RSUBRs

Once your FBIN or NBIN is ready you can actually do purification. To do this first

```
<USE "PURITY">
```

This PACKAGE contains the routines needed to purify RSUBRs. Then GROUP-LOAD the files you wish to have purified. Once this is done type

```
<GROUP-PURIFY group:atom output:channel>
```

This will purify and link all RSUBRs and RSUBR-ENTRYs in the *group* and will also attempt to purify any RSUBRs or RSUBR-ENTRYs called by the *group*. Giving the optional *channel* will cause GROUP-PURIFY to print information concerning the progress of the purification.

GROUP-PURIFY will only purify RSUBRs and RSUBR-ENTRYs. In order to purify tables, etc. use the PURIFY SUBR directly. Since purification is an extremely expensive operation, it is recommended that you collect together the things you wish to purify into a LIST, VECTOR, etc. and purify that structured object.

Once purification has occurred, several things may be done to recover wasted garbage collected space. The user can get rid of the "PURITY" PACKAGE by doing a

```
<KILL:PURITY>
```

The user can also remove much of the overhead of keeping a group around by UNASSIGNing the group-name. Removals of this type should be follwed by an explicit call to the garbage collector invoking the 'hairy' GC feature, as much of the storage to be regained is pointed to by associations. This can be done by

```
<GC 0 T>
```

In order to save a file with purified MDL objects you must SAVE. Restoring a SAVEd file with purified MDL objects will cause those objects to share with any other MDL RESTOREd from the same SAVE file.

6.5.2. Purifying an Environment

Many subsystems maintain a list containing pointers to all the static data structures built by that subsystem: dispatch tables, data bases, and so on. The list can be given to PURIFY to move all its components into the pure area. However, there are other structures in garbage collected space that may be purified; e.g., the RVECTORs of RSUBRs, RSUBR DECLs, and so on.

The "CLEAN" PACKAGE examines these structures, looking for those which may be purified. It may also be used for informational purposes. To get it

```
<USE "CLEAN">
```

"CLEAN" has one major ENTRY, CLEANUP, which examines every ATOM of every OBLIST in the MDL. It may perform a variety of functions, but it is most often used to make DECLs share storage and to accumulate a LIST of purifiable structures. All of its arguments are optional.

```
<CLEANUP print?:boolean
         reset?:boolean
         decl?:boolean
         gdecl?:boolean
         pure?:boolean
         check?:boolean
         avoid:list-of-oblists>
```

print? is by default FALSE. If non-FALSE, information about each ATOM examined will be printed as CLEANUP runs. This is a *lot* of information.

reset? is by default T. If non-FALSE, the LISTs of objects previously collected will be reset before CLEANUP runs.

decl? is by default T. If non-FALSE, each DECL element will be made to exist exactly once in the entire core image. E. g., there will be only one copy of the DECL <LIST [REST FIX]> in the core image.

gdecl? is by default T. It is similar to *decl?*, but refers to GDECLs.

pure? tells whether to make a LIST of all the purifiable objects in the core image. It is by default T.

check? tells whether to make LISTs of all the TYPEs, RSUBRs, RSUBR-ENTRYs, etc. in the core image. It is by default T.

avoid is a LIST of OBLISTs not to look in; it is by default the OBLISTs associated with "CLEAN" and "PURITY".

CLEANUP returns (if *pure?* is non-FALSE) a structure (also stored as the GVAL of PURELST) which may be given to PURIFY.

The results of running CLEANUP may be examined by

```
<PRINT-CLEANUP>
```

As the object in running CLEANUP is to shrink the size of one's MDL and its garbage-collected space, it is useful to be able to remove CLEAN after it has done its work.

```
<FLUSH-CLEANUP>
```

removes everything associated with the PACKAGE from the MDL.

6.5.3. Purification Summary

In a simple case, one can purify a 'subsystem' of one group maximally by

```
<USE "PURITY" "CLEAN">
<GROUP-LOAD "foo">
<CLEANUP>
<GROUP-PURIFY foo>
<KILL:PURITY>
<FLUSH-CLEANUP>
<GC 0 T>
<SAVE "foo">
```

6.6. TEMPLATEs

The PRIMTYPE TEMPLATE cuts down on the need for storage by allowing the user to specify exactly what he wants a structured object to contain, similar to 'structures' in PL/I or C.

To use this feature one must create a new TYPE of PRIMTYPE TEMPLATE. This can be accomplished by using the RSUBR TEMPLATE. The procedure for doing so is:

```
<USE "TEMPLATE">
```

```
<TEMPLATE name:atom ... specs ... >
```

where *name* is the name of the new TYPE and *specs* are specifications for each element of the TEMPLATE. This returns the TYPE name of the TEMPLATE and creates a creator of TEMPLATEs of TYPE *name*, called *name* itself, which can be applied to arguments to create objects of that TYPE of TEMPLATE.

The specification for the elements can be of several forms. It can be one of

a TYPE: *type:atom*

a 2-element LIST: (*type:atom length:fix*)

a 3-element LIST: (*type:atom length:fix count:fix*)

Below are some examples along with explanations:

`LIST`

is an 18 bit `LIST` pointer.

`(FIX 18)`

is a halfword `FIX` (can be both positive and negative and is checked for overflow).

`(FLOAT 18)`

is an 18 bit `FLOAT` (which is the left halfword of a 'normal' `FLOAT` and therefore somewhat restricts the precision).

`(FIX` *n*`)`

(where *n* is less than 18) is a positive `FIX` of length *n* bits (is not checked for overflow).

`BOOLEAN`

is not a MDL `TYPE`, but a one bit `FALSE` or non-`FALSE` depending on whether the bit is 0 or 1.

`(UVECTOR 18` *n*`)`

is an 18 bit `UVECTOR` pointer. The `UVECTOR` is of length *n*. The same can be done for `VECTOR`s.

`(STRING 36` *n*`)`

is a 36 bit string byte pointer. The `STRING` is of length *n*.

`ANY`

is not a MDL `TYPE`, rather anything can go here. This is relatively inefficient to use in `TEMPLATE`s as it takes up 2 words.

In order to provide more flexibility in using `TEMPLATE`s, two other fields are allowed, an *optional* field and a *rest* field. The *optional* field allows the user to create `TEMPLATE TYPE`s which will have the same basic structure but which can have optional elements determined when the actual `TEMPLATE` is created. The *rest* field, like the *optional* field, allows elements to be optional but specifies a pattern for any elements that are added on. It is analogous to `REST` in `DECL`s. Separation of fields is accomplished by the use of the strings `"REST"` and `"OPTIONAL"`. For example:

```
<TEMPLATE FOO FIX "OPTIONAL" LIST BOOLEAN "REST" FLOAT>
```

This creates a `TYPE FOO` of `PRIMTYPE TEMPLATE` which always has a `FIX` as the first element, can have a `LIST` as a second element, can have a one bit `T` or `#FALSE ()` as the third element and can have any number of `FLOAT`s from the fourth element on.

6.6.1. Use of TEMPLATEs

`TEMPLATE TYPE`s may be thought of as primitive `TYPE`s, in that they each have a unique storage representation. On the other hand, the `TYPEPRIM` of any `TEMPLATE TYPE` is `TEMPLATE`. A primitive `TEMPLATE` (which cannot truly exist in the language) would look like

{ element-1 element-2 ... element-n }

Real TEMPLATE TYPEs are represented as NEWTYPEs of this primitive TEMPLATE TYPE.

#type-name { ... elements... }

This method is similar to the usual method in MDL for representing any new TYPE, in that a RESTed TEMPLATE will be printed 'CHTYPEd to its PRIMTYPE.' Note that a TEMPLATE so printed cannot be read by READ: a 'primitive TEMPLATE' cannot exist. It is best to avoid printing RESTed TEMPLATEs.

Below are some examples of the use of TEMPLATEs.

```
<TEMPLATE BAR
          FIX
          "OPTIONAL" BOOLEAN
          "REST" (FIX 18) (FLOAT 18)>$
BAR

<BAR 1>$
#BAR {1}

<BAR 1 T>$
#BAR {1 T}

<BAR 1 <> 1 1.0>$
#BAR {1 #FALSE () 1 1.0}

<SET A <BAR 1 <> 1 1.9 2>>$
#BAR {1 #FALSE () 1 1.8984375 2}

<PUT .A 1 6>$
#BAR {6 #FALSE () 1 1.8984375 2}

<PUT .A 4 1.999>$
#BAR {6 #FALSE () 1 1.9960937 2}

<TEMPLATE BAR (STRING 36 4) "REST" ANY>$
#FALSE ("ALREADY A TEMPLATE")

<TEMPLATE BAR1 (STRING 36 4) "REST" ANY>$
BAR1
```

```
<SET A <BAR1 "HELP" 2 () <>>>$
#BAR1 {"HELP" 2 () #FALSE ()}

<PUT .A 1 "GOOD">$
#BAR1 {"GOOD" 2 () #FALSE ()}

<PUT .A 1 "GOOD-BYE">$

*ERROR*
TEMPLATE-TYPE-VIOLATION
PUT
LISTENING-AT-LEVEL 2 PROCESS 1
```

6.6.2. Assembly of TEMPLATEs

Once a set of TEMPLATE TYPEs is created, as for the TYPE definitions of a subsystem, it saves time to store away the 'compiled' TEMPLATE generators and not recreate them each time the definitions are to be used.

The "TEMHAK" PACKAGE modifies files which define TEMPLATE TYPEs to contain the TEMPLATE descriptions and RSUBRs rather than the calls to TEMPLATE. It is only useful, of course, when the TEMPLATEs are defined in a file which will not normally be edited, since the new files are in 'NBIN' format. To load this PACKAGE,

```
<USE "TEMHAK">
```

The PACKAGE has two entries.

<TEMPLATE-DUMP *group-name:atom*>

takes the group and modifies it such that <USE "TEMPLATE"> becomes <USE "TEMHLP">, and all *top-level* invocations of TEMPLATE are replaced by calls to BUILD-TEMPLATE (for the TEMPLATE descriptions), SETGs of the TEMPLATE-generating RSUBRs, and the GLUE bits for the RSUBRs.

<FILE-TEMPLATE *input:string output:string*>

takes an *input* file and performs the same service, GROUP-DUMPing the result to the optional *output* file (by default the same file with second name "NBIN"). This is useful for files which contain nothing but TYPE definitions, a common practice in large subsystems.

If the TEMPLATE TYPEs are defined in a file which will be edited frequently, a different set of routines is used after creating the TEMPLATE TYPEs:

<DUMP-TEMPLATES *descriptions:string*>

places the TEMPLATE descriptions (*not* the RSUBRs) in the specified *descriptions* file. It does so for all TEMPLATE TYPEs currently defined.

<DUMP-RSUBRS *rsubrs:string* *template-type:atom* ... >

will perform the same service for the TEMPLATE-generating RSUBRs of the TYPEs given as the second and later arguments to DUMP-RSUBRS.

There will now be two files, one containing the TEMPLATE descriptions and the other the RSUBRs. These may now be used to create the TEMPLATE TYPEs without USEing "TEMPLATE". To do so:

<USE "TEMHLP">

This defines the RSUBRs needed to take the TEMPLATE descriptions and make them useful to MDL.

<FLOAD *descriptions:string*>

the file of descriptions (the file created with DUMP-TEMPLATES): this *must* be loaded before the RSUBRs file. Then load the RSUBRs file (the file created by DUMP-RSUBRS):

<FLOAD *rsubrs:string*>

For maximum convenience, it may be necessary to put a FORM in files that create TEMPLATEs: if the TEMPLATE files described here exist, FLOAD them; otherwise, <USE "TEMPLATE"> and create the TEMPLATEs from scratch. It is of course possible to manually merge the two TEMPLATE definition files (preferably by using GROUP-LOAD and GROUP-DUMP), so long as the TEMPLATE descriptions precede the TEMPLATE RSUBRs.

TEMPLATE RSUBRs are created with GLUE bits, so it is possible to glue them into groups and to purify them.

7. The Assembler

It is occasionally necessary to write MDL routines in assembly language, usually to interface with a feature of the operating system not available in the interpreter. The MDL assembler (which is also used by the MDL compiler) provides this ability.

7.1. The Assembler

The MDL assembler provides the MDL user with a means of writing RSUBRs directly in machine language. The assembler is also used as the object language of the compiler. This section is a description of the assembler, its use, and some of its pseudo-operations.

7.1.1. General Organization

The MDL assembler is written in MDL to produce code that runs in the MDL environment. It takes arguments in the following form

```
<FILE-ASSEMBLE input-file:string
               output-file:string
               quick:boolean>
```

The arguments are an *input-file* containing MDL assembly code (possibly for several RSUBRs), an optional *output-file* in which to put the binary output (by default the same file as *input* but with second file name "NBIN"), and an optional third argument which tells whether to use NBIN format output, and which under normal circumstances should always be T. There are four other optional arguments which are the same as the second through fifth arguments of ASSEMBLE.

```
<ASSEMBLE body
          locals
          messages
          list
          symbols>
```

(All the arguments are optional with the exception of *body*.)

body may be a CHANNEL, in which case all instructions in the file associated with the CHANNEL are assembled, or it may be a structured object, in which case all instructions in the object are assembled.

locals specifies the OBLIST to use for local symbol lookup when the *body* is a CHANNEL. The default is <1 .OBLIST> when the assembler is called.

messages is a CHANNEL to receive error messages, etc. It defaults to .MESSAGE-CHANNEL.

list is a CHANNEL to receive an assembly listing. If *list* is not supplied, no listing is generated. If *list* is a non-FALSE non-CHANNEL, and *messages* is a CHANNEL, then the *messages* CHANNEL will receive the address of each label. If *list* is a FALSE, then no listing is produced. The default is .LINE-CHANNEL

(Initially LINE-CHANNEL is FALSE.)

symbols indicates if true that a DDT symbol table of all the labels for use with "RDB" (see section 7.2) will be generated. The default is .MAKE-SYM-TABLE (Initially MAKE-SYM-TABLE is FALSE.)

7.1.2. The Assembler as a Program

The assembler also exists as program called ASSEM, which encapsulates FILE-ASSEMBLE.

7.1.3. Format of Assembler's Source

The MDL assembler's equivalent of a line of code is a FORM. It assembles FORMs into instructions in much the same way that a typical assembler treats lines of source code. ATOMs at the top level (i.e. not in FORMs) are treated as labels. The FORMs are assembled based on the TYPE of the GVAL of the first ATOM in the FORM. The GVALs of ATOMs whose PNAMEs are the PDP-10 instructions are of TYPE OPCODE (PRIMTYPE WORD; the 'value word' has the 36 bit value of the instruction. For example, in

```
<MOVE A* 1 (B)>
```

the value of MOVE (in the OP OBLIST) is #OPCODE *200000000000*. This FORM is assembled directly into an instruction.

If the GVAL of the first ATOM in a FORM is something applicable (SUBR, FUNCTION, RSUBR etc.) the FORM is EVALed and the resulting SPLICE of FORMs is assembled. This is how macros and pseudo-ops are implemented. Notice that a pseudo-op or macro may produce no code by returning an empty SPLICE.

7.1.4. Instruction Assembly

Having determined that a FORM is going to assemble into an instruction, the assembler basically adds up the values of all the items in the FORM. In the case of items of TYPE OPCODE, a full 36 bit add is performed. Items of TYPE ADDRESS refer to labels in the program. Since the code is all location insensitive and will move around during garbage collection, references to labels must be indexed by accumulator M, the base register. Therefore, label symbols include an M in the left half and must also be added in with a full-word add. Items of PRIMTYPE WORD other than OPCODEs and ADDRESSes are ANDBed with *777777* before being added, and the carry from right half to left half is suppressed. When ATOMs are found in FORMs that are being assembled into instructions, special lookup rules are in effect. If the ATOM has a global value, that value is used. If the ATOM does not have a global value but has a local value, it is used. If the ATOM has neither a local or global value, it is assumed to be a local symbol for this assembly. In this case the symbol value is used if it has already been defined, otherwise it is added to a list of as yet undefined symbols.

Objects other than ATOMs or PRIMTYPE WORDs cause the assembler to take special action.

- LISTs are used to indicate swapping left and right halves. For example

  ```
  <MOVE (1)>
  ```

  would put the 1 in the index field of the MOVE instruction (similar to MIDAS).

- A VECTOR indicates a constant. The VECTOR may contain any number of FORMs to be assembled at the end of the program. For example:

  ```
  <PUSH TP* [<1 (1)>]>
  ```

  pushes a constant containing 1 in the right and left halves.

- A FORM is simply EVALed and the value returned is used.

7.1.5. Initial Symbols

The OBLIST structure in effect during assembly is

(*op mdl* DEFAULT *local root*)

The OBLIST *op* is named OP and contains the PDP-10 opcodes, the MDL accumulator definitions (in both accumulator and address fields), and the pseudo-ops. The OBLIST *mdl* is named MUDDLE and contains values of many labels in the interpreter. This enables programs to do things like <JRST FINIS>, the standard way to exit from an RSUBR. When an instruction is assembled using a symbol from the MUDDLE OBLIST, a fixup is also generated so that, if the symbol gets a different value in a new MDL, the code can be fixed up when it is loaded. *Local* is the user's local symbol OBLIST and *root* is the ROOT OBLIST.

As stated earlier, every accumulator has two symbols associated with it, one for the address field and one for the accumulator field. This is because there is no syntax to specify which field is intended. The address symbol is simply the accumulator's name, and the accumulator symbol is the name with an asterisk (*) appended to it; e.g., A versus A*.

7.1.6. Macro Writing

Whenever an element or subelement of an instruction is a FORM and the first element of the FORM has an APPLICABLE GVAL, the FORM is evaluated and the result (unless it is a SPLICE) is re-evaluated as if it were in place of the FORM. This feature constitutes the assembler's macro facility.

For compatibility between 'top-level' macros, which generate whole instructions, and macros which generate parts of an instruction, top-level macros may wish to return several instructions. To indicate that what is returned is several instructions, it is necessary to return an object of type SPLICE (PRIMTYPE LIST). The elements of the SPLICE are treated as individual instructions. An empty SPLICE may be returned from

a macro which is part of an instruction, and the effect is as if a 0 were returned. This is the only SPLICE which may be returned from a macro which is a part of an instruction.

7.1.7. Pseudo Operations

The next part of this document will describe pseudo-ops available in the MDL assembler. There is no difference between a pseudo-op and macro in the assembler except that the pseudo-operations are supplied by the system.

<TITLE *name:string*>

This is about the only required pseudo-op. It must be the first instruction to be assembled. It takes one argument, the name of the RSUBR being assembled. If additional TITLEs are found in a file being assembled, they are assumed to both end the previous RSUBR and begin the next. The assembler prints each TITLE on the messages CHANNEL as it is encountered.

<SUB-ENTRY *entry:atom decl*>

This pseudo-op is used to define additional RSUBR-ENTRYs for the RSUBR being assembled. The *entry* argument is the name of the RSUBR-ENTRY and the optional *decl* argument is a DECL for the entry.

<INTERNAL-ENTRY *entry:atom args:fix*>

is used to create an INTERNAL-ENTRY for a GLUEable RSUBR. Its arguments are the name of the INTERNAL-ENTRY and the number of arguments that will have been pushed on the stack for it when it is called. See also section 7.1.9 for details on writing GLUEable RSUBRs.

<DECLARE ("VALUE" *decl decl decl* ....)>

is used to supply declarations for the RSUBR named in the TITLE. It must occur before any code-generating instructions. DECLARE takes a LIST as its one argument. The format of the LIST is as described in [3]. The string "VALUE" is optional; if supplied it causes the first *decl* to declare the TYPE of the value of the RSUBR. Each additional *decl* is associated with one argument. Special STRINGs may also appear in the LIST with the following meanings:

"QUOTE" The next argument is QUOTEd (not EVALed).

"OPTIONAL" The rest of the arguments are optional (the RSUBR must supply any defaults for these).

"CALL" If this appears, it must be directly after the "VALUE" *decl*. It says there is one argument and it is the FORM generating the call (see "CALL" for FUNCTIONs in [3]).

"ARGS" This must be the last STRING. It says treat the rest of the arguments in the FORM as a LIST and pass it as the argument (see "ARGS" for FUNCTIONs in [3]).

"TUPLE" EVAL the rest of the arguments and pass them.

<END>

indicates the end of an RSUBR or group of RSUBRs. Only the text between TITLE and END pseudo-ops will be processed by the assembler. This makes it possible to intermix assembler source code and normal MDL source code in the same file (although assembly must be done before compilation in such cases).

<TYPE-CODE *type:atom*>

allows references to the internal TYPE codes for both system and user defined TYPEs. It takes one argument, the MDL TYPE name. For example:

```
<MOVSI A* <TYPE-CODE FIX>>
```

puts the TYPE code for FIX into the left half of accumulator A.

<TYPE-WORD *type:atom any ...* >

generates a reference to a word containing the TYPE code for *type* in the left half and possibly other junk in the right half. The first argument is the TYPE name and the rest of the arguments are optional but if supplied are added into the right half. If the TYPE is an initial TYPE and no right half is generated, a reference to the '$T*type*' location in the interpreter is generated. For example,

```
<PUSH TP* <TYPE-WORD FIX>>
<PUSH TP* [0]>
```

would push a FIX 0 on the stack.

<GETYP *a, type:atom*>

has the same form as a PDP-10 instruction. It gets the TYPE code for *type* into the right half of its accumulator from its address. This is done by generating an appropriate LDB (load byte) instruction.

<MQUOTE *object:any*>

allows the RSUBR to reference garbage collected space. It adds its argument to the RVECTOR (if it isn't already there) and evaluates to an address of the form *offset*(R), pointing to the value word for *object*.

<PQUOTE *object:any*>

is identical to **<<MQUOTE** *object:any*> **-1**> i.e. it points to the type-word, not the value-word. This is a more consistent way to look at things.

<IQUOTE *object:any label:atom*>

is like PQUOTE except that this will add a new element to the reference VECTOR each time called. The optional *label* if given defines the ATOM to be a label referring to that element. This is the only way to refer to that element again.

<PSEUDO *arg:any*>

evaluates its argument for its side effects and assembles no code.

<SIXBIT *string*>

makes SIXBIT of the legal characters of *string*.

<SQUOZE *string sqbits:word*>

makes SQUOZE of the legal characters of *string* and sticks the low-order four bits of the optional *sqbits* in the high-order four bits of the value. See the MIDAS Manual [4] for an explanation of the SQUOZE code.

<BYTE *boundary:fix byte-size:fix location*>

Example: <BYTE 1 35 (C) 1> is like <(*014300*) (C) 1>.

<ARG *argnum:fix*>

is like <(AB) <* 2 <- .*argnum* 1>>>. ARG should not be used in GLUEable code.

<STACK *sym1:atom sym2:atom sym3:atom* ... >

makes *sym1* a symbol for <(TB) 0>, *sym2* a symbol for <(TB) 2>, *sym3* a symbol for <(TB) 4>, etc. STACK should not be used in GLUEable code.

<DPUSH *ac args*>
<DPOP *ac args*>
<DMOVE *ac args*>
<DMOVEM *ac args*>

are the double-word PDP-10 instructions. For example,

<DPUSH *ac args*>

expands into

#SPLICE (<PUSH *ac args*> <PUSH *ac args* 1>)

<UNDEF? *symbol:atom*>

evaluates to true only if the *symbol* has previously in the code been used as a symbol, but has not been defined.

<IF-NEEDED *symbol:atom instructions* ... >

If <UNDEF? *symbol*> evaluates to true, then all the *instructions* are inserted at the current location, otherwise they are not.

<*INSERT *file-spec:string*>

takes a file and reads instructions from it and inserts the instructions read at the current place.

7.1.8. The Type RSUBR

An RSUBR is a MDL object of PRIMTYPE VECTOR. The first element of an RSUBR is always of TYPE CODE (or PCODE). CODE is of PRIMTYPE UVECTOR, consisting of words or instructions. The second element of an RSUBR is an ATOM which is the RSUBR's name. If the RSUBR has declarations they are the third element. The rest of the RSUBR contains MDL objects which must be referenced by the code

An RSUBR-ENTRY is a VECTOR of two or three items. The first item is either an RSUBR or an ATOM whose GVAL is an RSUBR, the second is an ATOM which is the entry's name and the third is a DECL for the entry. The difference between an RSUBR and an RSUBR-ENTRY is that an RSUBR always starts running at the beginning of the code when it is called while an RSUBR-ENTRY usually starts running somewhere in the middle of the code.

7.1.9. Writing Gluable RSUBRs

Certain conventions must be followed when writing hand coded RSUBRs in order to get the most benefit from GLUEing. If the RSUBR (or RSUBR-ENTRY) has "TUPLE" in its DECL, it is already in the best shape possible. In all other cases, the code after the TITLE or SUB-ENTRY pseudo-operation should simply push the arguments onto the TP stack and PUSHJ P* to one of the internal entries based on the number of items on the stack. After the PUSHJ it should do a <JRST FINIS>. An internal entry is set up by using the INTERNAL-ENTRY pseudo-op which takes two arguments: an *atom* and a *fix*. The *atom* acts as if it were a label on the next instruction and may be used as a label. The *fix* specifies how many items (type-value pairs) are on the stack at this internal entry. In the simple case where there are no optional arguments, only one internal entry exists and its number argument is exactly the required number of arguments. If optional arguments exist, some kind of dispatch will have to be done.

In the rest of the body of the RSUBR, no references to AB or TB (through the ARG or STACK pseudo-ops or directly) can be made, because after GLUEing their contents may be meaningless. All references to the TP stack must be indexed by TP. The usual precautions concerning the possible movement of code if an INTGO or MCALL is done also apply (i.e. the use of <SUBM M* (P)> at the beginning and <JRST MPOPJ> at the end of the code are essentially mandatory).

7.2. Debugging Binary Code

Binary code produced by the MDL assembler or the MDL compiler may be debugged with DDT, like any other binary code. However, an interface between that code and the DDT environment must exist. That interface is the "RDB" PACKAGE. It is obtained by

```
<USE "RDB">
```

The symbol table optionally produced by the assembler can be passed to DDT and at the same time the RSUBR frozen (moved out of normal garbage-collected space) by:

<RFREEZE *name-of-rsubr:atom*>

Note that *name-of-rsubr* may also refer to an RSUBR-ENTRY.

<RBREAK *name-of-rsubr:atom*>

is similar, but in addition causes DDT to put a breakpoint at the first instruction of the RSUBR.

If there is no symbol table, RFREEZE and RBREAK merely freeze the RSUBR and pass up symbols for the RSUBR name and any sub-entries.

In all cases the symbols passed up are made up of the legal SQUOZE characters (letters, digits, !\$, !\%, !\.) of the name, up to six characters. For example the ATOM FOO-*BLECH becomes the symbol FOOBLE.

<ADR *object:any*>

returns the address of *object* as a FIX. For example, <ADR *rsubr*> would return the location of the *rsubr* in core.

<RUNBREAK *name-of-rsubr:atom*>

clears the breakpoint(s) at the beginning of the RSUBR and of any of its sub-entries.

7.3. Unassembling Binary Code

Converting compiled or assembled binary code back into something resembling the original assembler source code is an operation that is performed primarily in one situation: tracking down a MDL compiler bug. It is, however, almost invaluable in that situation. The PACKAGE containing the unassembler is "UNASSM". The main entry is

<UNASSEMBLE *code:rsubr-or-group*
output:channel-or-string
glue?:boolean>

code is the object being unassembled. It is either an RSUBR (not an RSUBR-ENTRY, note), or an ATOM whose LVAL is a group (as created by GROUP-LOAD).

output is where to put the output; if it is a STRING, then the output is put in a file with that name. If *output* is a CHANNEL, then output is done on that CHANNEL. The file is "*code* UNASSM" by default.

glue? (by default T) tells whether there are glue bits for the code loaded. If there are none, this argument should be given as a FALSE.

The output produced by UNASSEMBLE is like the MDL compiler's assembler input, with the addition of comments which give code and stack offsets for stack slots referenced. This information is useful in tracing exactly what is going on in the code, but it is not always accurate, since the compiler's stack model is sometimes too complex for the unassembler to understand.

MDL compiler bug reports are expected to contain MDL source and UNASSEMBLEd compiled code if possible.

8. Informational Aids

This chapter discusses a few programs, most written in assembly language rather than MDL, which are nonetheless of use to MDL programmers. Most are informational aids of one sort or another. They include:

MUDCOM, a program for comparing versions of a MDL program. It is used by COMBAT (see section 5.2) to aid in the preparation of compiler plan files. It has several useful aliases.

MAT, the MDL 'atsign' program, produces listings, indexes and cross-reference files for MDL programs. @, a similar program which is not MDL-specific, will perform approximately the same tasks.

MUDINQ is an interface to the ITS IPC device and is therefore a means of interacting with any MDL that has the IPC device enabled. It has an alias, STATUS, which is particularly useful for determining the progress of compilations.

8.1. File Comparison and Checking with MUDCOM

MUDCOM is an assembly language program (not written in MDL), which nonetheless understands the syntax of MDL programs. It is used for comparing two versions of the same program, and also (under the name MUDCHK) for checking the syntax of MDL source files more rapidly than they can be loaded into a MDL. MUDCOM is not interactive; all instructions must be passed on the *jcl* line.

MUDCOM understands the following MDL structures at top level:

FUNCTIONs	<DEFINE FOO .....>
MACROs	<DEFMAC BAR .....>
GVALs	<SETG MUMBLE ....>
LVALs	<SET MUMBLE .....>
MANIFEST	
PACKAGE	
ENTRY	
ENDPACKAGE	
MSETG	<MSETG FOO 1> is <SETG FOO 1> <MANIFEST FOO>

The *jcl* for MUDCOM in the simplest case is *filename1*,*filename2*. MUDCOM will compare the two files and print out information concerning those structures it understands which have been removed, changed, or inserted.

MUDCOM has a number of switches which can be set. They are given as */switch*, where *switch* is the name of the switch. Currently the following switches are useful:

T prints totals at the end of the comparison.

L prints all FUNCTIONs and GVALs in the file.

C checks the file given for syntax (only one file name at a time).

M checks the files for changed MACROs and MANIFESTs In this mode, MUDCOM will make a second pass through the first file given in the *jcl*, looking for all occurrences of calls to changed MACROs and MANIFESTs. MUDCOM will consider FUNCTIONs making such calls as having been 'changed' and will tell which MACRO or MANIFEST caused the 'change'.

The following other *jcl* is understood by MUDCOM:

(*atom* . . .) appearing before the file names in the *jcl* will cause MUDCOM to think that those FUNCTIONs have been changed and will print them as such.

"*filename*" appearing anywhere in the *jcl* causes commands to be read from that file until the end-of-file is reached.

{*filename* . . . } is used to specify files to search in calls to MUDFND (see below).

Aliases of MUDCOM:

1. MUDCHK. MUDCHK *filename* checks a file for MDL syntax errors. This is the same as

   MUDCOM /C *filename*

2. MUDLST. MUDLST *filename* lists all FUNCTIONs and GVALs found in the file. This is the same as

   MUDCOM /L *filename*

3. MUDFND. :MUDFND *atom* . . . {*file file*} searches *files* for FUNCTIONs/GVALs called *atom*s. It can be used for finding a FUNCTION in a haystack. This is the same as

   MUDCOM (*atom atom*) {*file file*}

   Since typing this can be tedious, it is easier to use the "filename" convention and have a disk file containing the files to be searched (surrounded by {}s). Thus,

   MUDFND FOO BAR BLETCH "MARC;ZORK FILES"

   will look for the typical FUNCTION names in the files specified in MARC;ZORK FILES.

8.2. The MDL Listing Program MAT

MAT is a program for producing listings of MDL programs on the Xerox Graphics Printer (XGP) or a lineprinter. (MAT is short for 'MDL Atsign', after the general listing program named @).

Besides a listing of the program itself, MAT includes a symbol table -- a list of defined objects (arguments to DEFINE, SETG, etc.) and optionally a cross-reference listing -- a list of every place in the program each ATOM is used. MAT can also a produce a record file, so that the next time MAT is run on the same program, only pages that have changed will be printed.

MAT is invoked with a *jcl* line in the following format:

MAT *lrec=output<u>i</u>nput-files ... /switches ...*

More specifically, it takes any number of *input* files (separated on the *jcl* line by commas) and produces a listing of them in the *output* file, with options specified by the *switches* (each preceded by a /, and optionally a record file *lrec* (see section 8.2.4).

The *output* file name defaults on ITS to *xuname*; *input* @ or @XGP depending on whether the X switch is used, and on Tenex/TOPS-20 to *input*.MAT or *input*.XGP in the connected directory.

8.2.1. MAT Switches

The specific sorts of options available in MAT are controlled by a variety of switches which determine such things as whether to produce a cross-reference listing, whether to use the XGP as the output device, and so on. The following switches are implemented:

/C

causes a cross-reference listing to be produced. This is a table showing each reference to each ATOM (other than SUBRs, FSUBRs, and locals) in the *input* files.

/D[*file-name*]

specifies *file-name* as the file containing the user's definitions. Definitions are discussed in detail below.

/F[*text-font*,*header-font*,*comment-font*]

specifies the XGP fonts to use in the *output* file. They are respectively the font to use for the program itself, the font for subtitles and other headers, and the font for MDL COMMENTs and top-level STRINGs. The default directory is FONTS and the default second file name is KST. The default font is 20FG. /F also causes a /X to be performed.

/I[*file-name*]

specifies a file which contains the names of input files. This is in lieu of typing them all in each time MAT is run, useful for large subsystems incorporating many files. The input files listed should be separated by commas or carriage-returns.

/N

causes output of only the symbol tables and cross-reference listing (if specified). No heading or title pages are produced.

/P

On ITS, VALRETs a :PROCED to DDT and continues. Useful for long MAT runs.

/Q[*message*]

prints *message* at the bottom of each page. The default is a copyright message.

/R

creates a record file (this is automatic if '*lrec=*' is used). See below for details about record files.

/S

outputs each file in a multiple file listing separately.

/T[*name1 name2*]

specifies names to use on the title page (in lieu of the file names of first *input* file).

/U

prints a separate symbol table for each type of defined item in the *input* file(s) (e.g. FUNCTION, GVAL, etc.).

/X

declares that output is to be for the XGP. This changes the default *output* file second name to @XGP. If /F is used, /X is done automatically.

8.2.2. Subtitles

Subtitles can be used by including STRINGs in an input file which begin with the word SUBTITLE. The remainder of the STRING will be used as part of the header of each output page until another subtitle is found. The STRING need not be a COMMENT. Subtitles may have a maximum of 79 characters.

Any file containing subtitles will have a table of contents at the beginning of the listing.

8.2.3. MAT Definition

The facility exists in MAT to cause user specified actions to occur at the time a specific ATOM is about to be cross-referenced. The most important use of this is for functions which define things which the user would like MAT to recognize, for example, a function one of whose side-effects is to SETG one of its arguments.

When MAT encounters an invocation of the function FOO, where FOO has been defined to MAT, it runs code generated by the user's MAT definition for FOO, which causes various actions to be performed.

MAT definitions are always located in a disk file which is specified by the /D switch. Each definition must be of the form:

[*name arg1 arg2 arg3 . . .*]

where *name* is the name of the item which is being defined and the *args* are action specifications as described below.

The syntax of a MAT definition is somewhat complex. Basically, there are two types of actions which can take place: 'setting' an ATOM to be equivalent to a specified type (i.e., FUNCTION, MACRO, etc.) or 'cross-referencing' the ATOM (i.e., making it appear in the cross-reference listing).

The actual definition for an ATOM is a string of MAT action specifications, one for each argument in a call to that ATOM. For example, defining FOO to be

```
[FOO SETG SKIP SETG]
```

implies at least three arguments to FOO, the first and third of which should be treated as if they were SETGed. Thus, if

```
<FOO FROB 1 MUMBLE>
```

were encountered in an input file, it would be treated as though

```
<SETG FROB any>
<SETG MUMBLE any>
```

had been encountered. The symbol table would then point to the line on which the application of FOO appeared as the location of the definitions of FROB and MUMBLE.

The following tokens are meaningful action specifications:

CREF means to cross-reference this ATOM.

SKIP means to do nothing with this argument (a place holder).

REST means that the rest of the action specifications may be repeated for the rest of the arguments.

name (where *name* is the name of a MDL SUBR which causes some action to be routinely performed) means to act as though the ATOM had had that SUBR applied to it. For example, SETG will cause MAT to treat the item as if a SETG had been performed on it. Similarly, MANIFEST will cause MAT to believe it MANIFESTed.

ALSO means to do another thing to this ATOM. Thus, [SETG ALSO MANIFEST] specifies that the argument should be treated as though it were both SETGed and MANIFESTed.

=*xy* where *xy* are two characters, causes a user defined symbol type to be created. In the cross-reference, this will appear as *xy* in front of the name of the ATOM.

Any of the preceding tokens may have !-*oblist* added. This means that instead of the ATOM being set to the specified type, *atom*!-*oblist* will be set. Thus, for example,

```
REST SETG!-FLAGS
```

might specify a function which takes a LIST of ATOMs and performs

```
<SETG <INSERT atom <GET FLAGS OBLIST>> any>
```

on each of them.

[SPEC *xy name*] specifies *name* to be the expansion of *xy* for purposes of the symbol table. *Name* cannot have spaces in it.

Since not all items to be recognized within a function call are at top level, there is a facility for telling MAT to recognize structures. This is done by inserting the correct bracket (which MAT will encounter) around the part of the action specification referring to a structure. For example, a definition for GDECL (which *is* handled internally, however) might be

```
REST (REST GDECL) SKIP
```

which specifies that the arguments are alternately a LIST of things to GDECL and an argument which is unimportant.

A special case of bracketing is when the location of the structure is not known. In this case, *bracket*! means 'find the next object that starts with this bracket'. An example later demonstrates this.

What follows are some examples from a real definition file.

```
[NEWSTRUC NEWTYPE SKIP REST SETG SKIP]
```

NEWSTRUC takes an ATOM which becomes the name of a NEWTYPE, the DECL for that TYPE (which is not interesting to MAT) and an arbitrary number of pairs of ATOMs (names of offsets in the structure) and their DECLs (again, not interesting).

```
[FLAGWORD REST SETG]
```

FLAGWORD takes an arbitrary number of ATOMs and SETGs them something.

```
[SPEC PG Pure-Gval]
[SPEC OB Object]
[SPEC AC Action]
[SPEC VB Verb]
[SPEC OS Object-Synonym]
[SPEC AD Adjective]
```

These define the long descriptions for the newly defined symbol types created in the examples.

```
[PSETG =PG]
```

PSETG takes an ATOM and a value and SETGs the ATOM (also putting it in a LIST of ATOMs to purify).

```
[GET-OBJ "CREF"]
```

GET-OBJ takes a STRING PNAME of an object and returns the object. This definition allows "*object*" to be cross-referenced here. Note that CREF is in quotes because the element being dealt with is a STRING.

```
[OBJECT ["=OB" REST "=OS"] [REST "=AD"]]
```

OBJECT creates objects which are referenced by GET-OBJ. OBJECT first takes a VECTOR of STRINGs, the first of which is the true object specifier (OB) and the rest of which are synonyms (OS). The second argument is a VECTOR of STRINGs, which are PNAMEs of adjectives referring to the object (AD).

```
[ADD-ACTION "=AC!-ACTIONS" SKIP REST [[!"=VB!-WORDS" SKIP]]]
```

ADD-ACTION creates 'verbs'. The name of the verb is the first argument, which is a STRING. ADD-ACTION SETGs *string*!-ACTIONS to an item of type ACTION (AC). The second argument is not interesting. The rest of the arguments are VECTORs, somewhere in which is a VECTOR of a STRING and an uninteresting object. ADD-ACTION SETGs this latter STRING (the PNAME of an ATOM in the WORDS OBLIST) to something of type verb (VB). This is about as complicated as a MAT type specification is likely to get.

```
[1ADD-ACTION "=AC!-ACTIONS ALSO =VB!-WORDS"]
```

1ADD-ACTION takes as its first argument a STRING which is SETGed both in the ACTIONS OBLIST and in the WORDS OBLIST, to an ACTION (AC) and a verb (VB), respectively.

8.2.4. MAT Record Files

Listing Record (or LREC) files, akin to @ LREC files, can be produced in MAT by including *file*= in the *jcl* line. Use of an LREC file has the advantage that future invocations of MAT using it need only output the changed pages of the listing. The LREC file produced will be placed in *file* and contains all relevant *jcl* information, so that future calls to MAT for comparison listings need only have *file*= in the *jcl* line. Additional *jcl* may then be appended. There is, however, no way to turn off flags once set up. Therefore, if a cross-reference file is to be used only occasionally, leaving the cross-reference (/C) flag off for the initial listing and appending it at other times is preferable.

An alternate way of creating a Listing Record file is to use /R which is equivalent to

input-file-first-file-name LREC=

in the *jcl*. Obviously, /R is not sufficient for future comparisons.

8.3. The MDL-IPC Device Interface MUDINQ

MUDINQ is a small program that formulates, sends, and receives messages to and from MDLs over the ITS IPC ('Inter-process Communication') device. The user specifies a target MDL process by its *uname* and *jname*, either on the *jcl* line or to MUDINQ directly. He then inputs the message to be sent to that MDL. The message sent is enclosed in an invisible protective shield (an ERROR handler and so forth) to prevent it from interfering in the operation of the target. The message is PARSEd and EVALed by the target, and the result put in a file which is printed by MUDINQ when it appears.

The most common use of this program is to answer the question 'What could my compilation (or whatever) be doing after all this time?' The answer may be obtained by MUDINQing a <FR&> or <FRAMES> at it.

Inquiring after the state of a compilation is such a common use of MUDINQ that there is an alias of it, STATUS, which MUDINQs a <STATUS> (see section 5.1.1) at a compiler process and waits for a response.

Finally, an alias of MUDINQ called WHOM lists those MDL jobs listening on the IPC device.

For more details on the operation of the MDL IPC interface, see [3].

References

[1]

Edward H. Black.
Using MDL's Calico User Interface.
Technical Report SYS.11.21, MIT LCS Programming Technology Division, 1976.

[2]

Richard M. Stallman.
EMACS.
Technical Report 519, MIT AI Laboratory, August, 1979.

[3]

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

[4]

Peter Samson.
MIDAS.
Technical Report 90, MIT AI Laboratory, October, 1965.

[5]

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

Index