A LISP INTERACTIVE PROGRAMMING ENVIRONMENT

by

Wayne F. Hall

B.Sc., University of British Columbia, 1972

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard.

The University of British Columbia
April 1974
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date April 19, 1974
Abstract

An implementation of a Lisp Library is described. This Library provides a simple and uniform scheme for programmer access to a number of Lisp support programs. A scheme by which programs are automatically loaded from this Library is presented, as well as methods of querying the system and unloading selected programs. A facility for aiding in the debugging of Lisp programs is also described.
Table of Contents

I Introduction ..................................................1

II Preliminary Remarks ..........................................3
  1 Design Considerations .....................................3
    1.A Related Work ........................................3
    1.B Design Criteria .....................................4
    1.C The Autoloader Mechanism .............................5
    1.D The Debug Package ..................................6
    1.E Prettyprinting ......................................7
      1.E.1 Syntactic Approach ...............................8
      1.E.2 Semantic Considerations ........................9

III The Lisp Library Manual ..................................11
  1 Introduction .................................................11
    1.A Notation ..............................................12
    1.B What to read first ...................................13
  2 The Lisp Library ..........................................14
    2.A How to Run ...........................................14
    2.B Library Structure ...................................16
      2.B.1 Index Format ....................................16
    2.C The Loader ...........................................18
      2.C.1 The Autoloader Routine ..........................20
    2.D Library Utility Programs ..............................21
      2.D.1 GENLIB Commands ................................24
  3 Lisp File I/O Functions ..................................30
    3.A The FNS list .........................................31
      3.A.1 Comments ..........................................32
Since its early development the Lisp language has undergone considerable change. It has grown from a mathematical language into a very powerful system for symbolic and logical manipulation of complex data structures. Lisp has been used almost exclusively as a research language in the field of Artificial Intelligence, although it is now beginning to be used by researchers in other fields.

Lisp is by nature an interactive language. Lisp programs are stored in the same way as data, thus allowing programmer access to all programs. The Lisp error handling routines are designed to permit "soft" recovery from all errors in a user's program. In order to best use the power and flexibility of the language one must interact with it at a conversational terminal.

It is important in the study of any programming language to investigate the use to which that language is to be put. The design and implementation decisions should be guided by the knowledge of how different features in the language will be used. For the type of computing done in Artificial Intelligence, almost all computing effort and time is spent debugging the programs. In many cases a program is discarded or is of no further interest once it is operational.

The basic demands of a programming system which is to perform well in this type of environment are that the system provide a powerful set of tools for editing, manipulating, debugging, and storing the programs which the programmer wishes
to use. Speed of execution, while still of considerable importance, takes a secondary role to the efficiency and ease of interaction with the language. The question "How can it be done?" becomes more important than "How much will it cost?".

The purpose of the work described in this thesis is the design and implementation of a collection of Lisp programs which will provide the Lisp programmer with a variety of aids for writing and debugging programs. One of the goals of this system is to provide an environment in which the Lisp programmer need not leave Lisp in order to edit or manipulate his programs. All the tools necessary to debug a user's program should be available within Lisp.

The mechanism by which this is achieved is the Lisp Library system. This Library, together with the Loader routines that it uses, provides the user with a simple yet powerful method of accessing a number of useful Lisp utility functions.
II Preliminary Remarks

1 DESIGN CONSIDERATIONS

1.1 Related Work

Of the many implementations of Lisp, Interlisp[4] provides the most complete set of programmer aids. Included in the Interlisp system is an Editor for editing Lisp programs within Lisp, a break package which allows program execution to be monitored or suspended, an extensive set of I/O features and a prettyprint program. Teitelman[3] has done considerable work both on the Interlisp system itself and on a "Programmer's Assistant". The "Programmer's Assistant" provides the user with an "assistant" which can remember what has occurred in previously typed lines, correct simple syntactic spelling mistakes, and undo the effects of previously executed programs.

Many of the programs described in this thesis have been designed after the concepts and programs in the Interlisp system. The Editor, which was written by P. Friedman[1], is patterned after the Interlisp editor. The Debug Package serves a similar function to the Interlisp break package, although the "step" features of the Debug Package are not present in the Interlisp system.

Design Considerations
The Lisp Library System described in Chapter III consists of a Loader which can load various utility programs into Lisp. The utility programs, such as the Editor, Debug Package, and File I/O routines, are collected into a single file called the Library.

In the design of the Library system, the following criteria were of primary importance:

1. The system should encompass all of the various subsystems available in Lisp. This includes the Editor, prettyprinter, I/O functions, and Debug Package as well as any commonly used programs or functions.

2. The user should be able to call any of the Library routines directly without explicitly having to load them. This means that the Library functions can be thought of as part of the Lisp interpreter by the programmer.

3. A user should be able to build his own Library of Lisp programs and have the Loader use them as well as, or instead of, the programs provided in the main Lisp Library.

4. Facilities should be provided which allow the user to query the Library system about its status. This
information could be used by the programmer to direct the Loader to release the space occupied by a Library function.

\[1.C\] The Autoloader Mechanism

One of the basic criteria for the Lisp Library is that a Library function should be available to the user simply by calling it. The user need not worry about when to load a function or what subfunctions are to be loaded. In essence this allows Library routines to be treated in the same manner as the basic functions of the Lisp interpreter. In fact, if a Library function were compiled or rewritten in assembly language, and made a resident part of the Lisp interpreter the user should not notice any differences in the way he uses that function, except for an increase in speed.

The mechanism by which this duality between interpreter and Library functions is achieved is the autoloader routine. This program is responsible for calling the Loader when a Library routine is called but not present in core memory. It is invoked by the error routines of the Lisp/MTS interpreter (see Appendix I for a brief description of Lisp/MTS). When a function is called which is undefined, the Lisp/MTS error routines suspend the execution of the user's program and call the Autoloader. The Design Considerations
Autoloader examines the undefined function's name, and searches the Library index, loads the Library function and restarts the execution of the function. If the function cannot be found in the Libraries, the error is treated as a normal Lisp error.

1.D The Debug Package

One of the most useful routines in the Lisp Library is the Debug Package. This program allows the user to observe the execution of his program step by step, with the ability to intervene in its execution. Included in the commands available to the user are commands to edit parts of the user's program, display and search the Lisp stack, and resume execution at various points in his program.

The basic design of the Debug Package is similar to the break package of Interlisp, a version of which has been written for Lisp/MTS by P. Friedman[1]. The MTS assembly language symbolic debugging system (SDS[2]), has influenced in some ways the design of the Debug Package. The STEP feature of the Debug Package is patterned after its counterpart in SDS.

The basic STEP mechanism is designed around a specialized interpreter function called STEP. By inserting into the
internal EVAL routine a counter, it is possible to cause a transfer to the Lisp error routines after a specified number of EVALs. By changing the error forms which the Lisp error routines call, it is then possible to transfer control to the Debug Package. The Debug Package commands give the user complete control over the execution of his program.

A more detailed description of the STEP function can be found in the Lisp/MTS Users Manual[5]. The Debug Package is described in greater detail in Chapter III.

# 1.E # Prettyprinting

One of the most valuable aids to writing, debugging, and understanding programs in any language is the visual structure of the program when it is printed on a page. If the programmer can indicate visually the logical structure of his program, he can more easily follow its logical structure. In most languages this visual structuring is done by the programmer when he first writes his program. This may cause him trouble when later corrections cause the logical structure to change, since he must then either restructure the program visually or throw away this aid.

Design Considerations
In Lisp, as in most other languages, it is possible to write a prettyprinter which will print a program in a visually structured way. In the Lisp Library system if the user rewrites or edits a program in Lisp he must write it back into a file. Due to the total unreadability of Lisp programs which are printed simply as huge lists, it is necessary that some form of prettyprinter be available.

1. E. 1  Syntactic Approach

Since all Lisp programs are simply large lists, it is relatively simple to write a Lisp program which will print out a program using the list structure to guide the indentation of lines. Since Lisp is almost totally devoid of syntax, the program is very easy to write.

In Lisp/MTS the character ' is recognized as standing for the QUOTE function. Thus, when the user types in

'\n(A B C)

the read routines convert this into

(QUOTE (A B C))

If a prettyprinter is to attempt to produce output which in some way mirrors the way the programmer would write it, these quote symbols must be re-inserted into the output. A similar problem exists with breakpoints from the Debug Package since these breakpoints make actual changes to the user's program. The Design Considerations
prettyprinter should not "see" these breakpoints when it writes out the program.

In order to efficiently implement these features in the prettyprinter the print macros of Lisp/MTS were modified in such a way that whenever the internal print routine is about to print (QUOTE (A B C)), a special print intercept function is called which prints '(A B C). A similar mechanism is used to handle breakpoints. This feature could also be used to extend the prettyprinter to handle other specialized functions (such as the Micro-Planner readmacro characters $?, $G, $T, etc.).

1.E.2 Semantic Considerations

Since Lisp has a very weak syntax, it is only reasonable that the semantics of the language be used to visually reflect the logical structure of a program. As an example, the COND function in Lisp consists logically of a number of alternate conditions and statements. A purely syntactic prettyprinter might print a COND statement as

(COND ((ATOM (CAR X)) (CDR X)) ((LISTP X) NIL) (T T))

when in fact its logical structure would be better displayed by writing it as
(COND ((ATOM (CAR X)) (CDR X))
     ((LISTP X) NIL)
     (T T))

Similarly, in Lisp/MTS the SETQ function allows any number of variable-value pairs of arguments instead of just one pair. In many instances a strictly syntactic prettyprinter would print these as

(SETQ A
     (CAR X)
     LST
     LST1
     LST2
     (CONS A B))

When in fact the semantic structure is more like

(SETQ A (CAR X)
     LST LST1
     LST2 (CONS A B))

Evidently, in order to produce a listing of a user's program which will be most useful for debugging and understanding the program, the prettyprinter must be aware of the meaning of parts of the program, as well as the list structure of the program. The prettyprinter described in Chapter III attempts to meet at least part of this requirement.
1 INTRODUCTION

The following manual describes a group of functions collectively called a Lisp Library. This Library contains a number of general purpose Lisp functions to aid a Lisp programmer with problems of writing, maintaining, and debugging his programs.

The Library scheme, described in the first section, is designed to allow a number of different routines to be conveniently accessed through a common interface. The Loader which loads these routines into Lisp allows the user to write and maintain his own Library and use it in conjunction with the main Lisp Library.

This manual assumes that the user is familiar with the LISP/MTS interpreter and User's Manual[5]. A brief description of some of the features of Lisp/MTS can be found in Appendix I. Listing of the Loader and Library programs can be found in Appendix II and Appendix III.
In the description of Lisp functions (of which there are many in this manual), it is necessary to convey certain key pieces of information about the function and how it is called. A complete description of what the arguments are and what they do is of prime importance. The value returned by some functions is also of importance. In the description of the arguments it is necessary to indicate which arguments are optional, what their default values are, and which ones get evaluated.

The following scheme is used to describe all the functions and commands in this manual:

1. Optional arguments are enclosed in square brackets as in [FILE]. The default values, if any, are indicated in the description of the argument.

2. Quoted arguments, that is ones which are not evaluated, are indicated by underscoring, as in FN1 or [OUTFILE].

3. Where one or more occurrences of the same argument is possible the sequence is indicated as A1 A2 ... An.

Examples:

(DISKIN FILE1 FILE2 ... FILEn) indicates one or more FILES may be specified for DISKIN; none of the arguments will be evaluated.

(EDIT F) evaluates the single argument F.

(XREF F [OUTFILE]) does not evaluate either argument, and the second one is optional.
Since this manual is quite large, it is probable that few people will want to read it completely the first time. For someone who would like to use a particular function or group of functions, it is useful to indicate some order of priorities as to what should be read first.

The first section to read, and you probably already have done so, is the Introduction, especially the sections on Notation and "What to read first". The only other part absolutely necessary is the beginning of the next section entitled "How to Run". With this minimum of information, it is possible to call and execute many functions in the Library.

The routines in the section on File I/O are probably the most commonly used functions. This section includes the prettyprinter, DISKIN, DISKOUT, and UPDATE. The description of the FNS list is important, since many other functions in the Library use these lists.

The Debug Package is in a more or less self contained section. The Debug Package will probably take some getting used to, so be prepared to spend some time to get familiar with it. (Try it, you'll like it).
This section describes the structure of Lisp Libraries and how they can be used and generated.

2. How to Run

The Library routines are made available by loading into Lisp the LOADER routines. If we assume that this file is under the ID PSIL, the following MTS command will run Lisp with the Library:

$RUN PSIL:LISP SCARDS=PSIL:LOADER

When in Lisp, any Library routine may be used simply by calling it. If it has not yet been read in, as is the case prior to its first invocation, it will be loaded from the Library. A message is printed to inform the user that the function has been loaded, and execution continues as if it had always been in core.

The following example shows a simple run using the Library.
Example:

```lisp
# RUN PSIL: LISP SCARDS=PSIL:LOADER
APR 2/74: REVISED INTERPRETER
> (LOADER
* (DISKIN TESTFYLE)
> LOADING: DISKIN
> 4 FORMS READ FROM: TESTFYLE
>
* (SETQ ?MAXLEN 50)
> 50
* (PP F91)
> LOADING: PP
> (LAMBDA (Z)
>   (IF (GREATERP X 100)
>       (SUB X 10)
>       (F91 (F91 (ADD X 11)))))
>
* (EDITF F91)
> LOADING: EDITF
: EDITING F91
: (LAMBDA (Z) (IF & & &))
* INSERT X FOR Z 1
: (LAMBDA (X) (IF & & &))
  OK
> NIL
* (UPDATE TESTFYLE)
> LOADING: UPDATE
> FILE: TESTFYLE UPDATED.
* (STOP)
```
This section deals with the actual structure of Lisp Libraries, and how and where the various bits of information necessary for loading a function are stored.

The Library file itself consists of an INDEX which is a Lisp S-expression, and the Library functions themselves. The file is an MTS line file. The INDEX is normally stored in the negative line numbers while the function definitions are stored in the positive lines. The INDEX describes for each function, where it is and what other subfunctions must be loaded with it.

2.8.1 Index Format

The Index is a list of the form

(INDEX S1 S2 ... Sn)

where each Si is the loader information for a single Library function. The Si are of the form

(FN L1 L2 ... Ln)

FN is the name of the Library function being indexed, and the Li specify where in the Library file the necessary subfunctions are to be found. Each Li is either the MTS line number where the functions are to be found, or the name of some other Library function, either in this same Library or in some other Library.
The use of the names of other Library functions reduces the number of Li elements in the index entries. It also allows a function in one Library to have a subfunction in another Library.

Example:

(INDEX
   (DISKIN 1 101 201)
   (PP 301 401 501 601 701 801)
   (UPDATE 901 PP)
   · · ·
)

In this example, to load the routine DISKIN, the functions at lines 1, 101, and 201 must be loaded. To load UPDATE, the functions at lines 901, 301, 401, 501, 601, 701, and 801 must be loaded. The function GENLIB, described later in this document, can be used to generate this Index semi-automatically.
The Lisp Library manuals are the heart of the Lisp Library system. The Loader is able to search through a library index, read in the Lisp functions appropriate to the function being loaded, and restart the execution of the user's program. The Loader will handle any number of Library files. The function LOADFN can be used to explicitly call the Loader or the default automatic loading feature can be used to cause the Loader to be invoked whenever a Library function is called which has not been loaded.

For each Library that has been referenced, the Loader saves all the information it needs in order to access the Library and load functions from it. This information is stored on the property list of the Library name. The following list of property tags are used by the Loader. The tags are all of the form ".tag." in order to avoid likely conflicts with user defined property tags.

".BUFFER." - the buffer (actually a Lisp IOARG) which is opened on the Library file. Saving the buffer avoids the necessity of reopening the file each time the Loader is called.

".INDEX." - the Library index, as described above. Note that the index is not read in until the first reference to the Library.
"LOADMAP." - a list of all functions and subfunctions which have been loaded from this Library. This list is used by the Loader to avoid rereading functions which have been read in previously. The list is of the form:

((L1 FN1 . C1) (L2 FN2 . C2) ... (Ln FNn . Cn))

where Li is the MTS line number from which this function or expression has been read, FNi is either the Expr function name, or NIL if a non-Expr was read, and Ci is the number of times the function has been loaded (normally 1).

"TOPCALLS." - a list of all functions which the Loader has been called on to load. These are only the top level routines, not the subfunctions of these routines.

The above information is stored the first time the Loader is called on to load a function from the particular Library. Successive calls update the existing lists. The UNLOADFN routine will delete entries from these lists.

The global variable #LIBRARYLIST# is a list of all the Libraries from which the Loader can load functions. This is initially the list (PSIL:LIBRARY) which is the Library file for the routines described in this document. The user may add his own Library to this list. The Libraries are searched in the order in which they appear in this list. If two Libraries both have a function with the same name, the Library closer to the front of this list will be the one from which the function will be loaded.

The Loader will also allow functions in one Library to reference functions from another Library. This avoids the necessity of making a copy of one Library routine in order to
use it from another Library.

2.C.1 The Autoloader Routine

The Loader contains a special routine which interfaces with the Lisp error handling routines. This routine enables the Loader to automatically load a Library function when it is called. The same routine also handles the automatic invocation of the Debug Package (see Section 4). The Autoloader is invoked whenever an undefined function error is detected by Lisp. It searches the index of each of the Libraries listed in #LIBRARYLIST#. If the function is found, it is loaded and execution resumes; if it is not found, the error is processed as a regular Lisp error. When the Autoloader loads a function, a message is printed on the terminal indicating to the user that the function has been loaded.

The Autoloader is controlled by the following global switch:

?AUTOLOAD - default is T. If this switch is NIL the Autoloader will not be invoked when a Library function is to be read in, hence all Library routines will have to be explicitly loaded via the LOADFN routine. If this switch is non-NIL, the Autoloader will attempt to load the function (see below).
In order to provide a more flexible Library system, the following functions have been included in the Lisp Library. They are functions which explicitly manipulate Libraries, and Library functions.

The function LOADINFO can be called to provide the user with information about which Library routines have already been loaded from the Libraries. UNLOADFN is used to remove definitions of Library routines that are no longer required. In some instances it may be desirable to extract from a Library a function together with all its subfunctions, in order to add it to another Library or to write it onto a separate file. The function EXTRACTFN is meant to do this. Finally, GENLIB can be used to build a new Library or update an existing one.

* (LOADFN F1 F2 ... Fn)

Purpose: to explicitly call the Loader routines.
Fi - the name of the Library function to be loaded.

The functions specified are loaded from the Libraries specified by the list #LIBRARYLIST#. If a function cannot be found in any of the Libraries, an error message is printed. The value returned by LOADFN is T.
Example:

(LOADFN PP DEBUG UPDATE) loads the functions PP, DEBUG, and UPDATE from the Library.

* (EXTRACTFN F)

Purpose: used to extract a Library function from a Library.
F - the name of the Library routine to be removed.

EXTRACTFN loads a library routine much as the Loader would, but in the process of loading, a list is built of all functions and forms which it loads. This list has the FNS format described in the section on File I/O functions. The list is stored on the property list of the function name under the tag "FNS." A function which has been extracted in this way can be written out to a file using DISKOUT, as if it had been read in by DISKIN.

The value returned by EXTRACTFN is the FNS list it generates.

Example:

(EXTRACTFN PP) extracts the prettyprint routines from the Lisp Library.

(DISKOUT -A PP) writes it out to file -A.
• (LOADINFO [FULL])

Purpose: to print out information about what Library functions are currently loaded.

FULL - this is an optional argument which will cause a more extensive list of Library functions to be printed. Its value is not used, only its presence or absence is of any consequence.

LOADINFO prints for each Library, the top level routines which have been loaded from the Library. If the argument FULL is present, a list of all the subfunctions, their file line numbers, and load counts are also printed. These lists are the "TOPCALLS." and "LOADMAP." lists described previously in the section on the Loader.

LOADINFO can be used in conjunction with UNLOADFN to avoid keeping routines around which are no longer needed.

• (GENLIB FILE LIB)

Purpose: used to generate a new Library file.

FILE - the name of an MTS line file which is to be the new Library file. This file should be created by the user. GENLIB does not empty the file before writing into it.

LIB - the name of an existing Library. All the functions in this Library will be the initial entries in the new Library. If LIB is NIL, no initial Library is used.

GENLIB extracts all the functions in the old Library specified by LIB, then allows the user to enter any of a number of commands specifying which additions or deletions are to be made to this list. After all changes have been made, the command OK will cause GENLIB to write out the new Library file.
The functions are written using the DISKOUT function.

GENLIB operates by first building a list, called FNLIST, of FNS lists built from the index of the old Library. This may be modified, and new sublists may be added by using the commands below. When the command OK is entered, the FNLIST is used to generate the new Library index which in turn is written out to the new file. The new index starts at MTS line number -1000 of the new file.

GENLIB returns the value DONE if the Library was successfully written, and FAIL if it was not. If the routine FAILED, an error message will be printed explaining what happened.

In order to allow a new Library to use functions from other Libraries without copying them, the command EXTERNAL is included. This command indicates to GENLIB what functions are external, and are to be entered in the new index as external references (see INDEX description earlier in this section).

2. D. 1 GENLIB Commands

The following set of commands are recognized by GENLIB. In addition to these commands, anything which is not a valid command name is EVALed, and the result printed. For commands which allow an arbitrary number of arguments, the end of a line
is used as a terminator. Any error in the processing of one of the arguments of these commands will cause the rest of the line to be ignored. The prefix character "$" is used by GENLIB to indicate that it is waiting for a command to be typed in.

- **ADD NAME FNS**
  
  -- adds a new function to the library. The name of the function will be NAME. FNS specifies what functions and subfunctions are to be part of this entry. If FNS is an atom, the FNS list stored on the property list of that atom under the tag "FNS" will be used, otherwise FNS must be a list of the same format as those created by DISKIN. See the FNS description in Section 3.

- **ADDTO NAME F1 F2 ... Fn**
  
  -- add new subfunctions to existing entries in FNLIST. The entry in FNLIST for the function NAME is modified by adding to it the function names Fi. The Fi must be valid elements of a FNS list, that is, either function names or dotted pairs of atoms and property tags, or a list whose CDR is a form to be added to the library.

Example:

If THISFN is a function in the library, the command

ADDTO THISFN THATFN THEOTHERFN

will add the functions THATFN and THEOTHERFN to the list of routines which must be loaded by the Loader when THISFN is loaded.

- **CLEAR**

  -- the EXTERNAL list, generated by the EXTERNAL command, is reset to NIL.
- **DELETE NAME1 NAME2 ... NAMEn**  
  Abbreviation: DEL

  -- deletes a Library function from the new Library. This command has the opposite effect to the ADD command. NAMEi must be the name of one of the entries in the FNLIST. The entry in FNLIST whose CAR is NAMEi will be deleted from the list, thus removing this function from the new Library. This command can be used in conjunction with the LIB argument to GENLIB to remove unwanted functions from an existing Library.

- **DELFROM NAME F1 F2 ... Fn**

  -- each of the subfunctions Fi are removed from the index entry of NAME in the FNLIST. This command has the opposite effect to the ADDTO command.

  Example:

  If THISFN is a Library function which has the subfunctions THATFN and THEOTHERFN, the command

  DELFROM THISFN THATFN THEOTHERFN

  will remove THATFN and THEOTHERFN from the index entry for THISFN.

- **EXTERNAL F1 F2 ... Fn**

  -- adds the functions Fi to the list of external Library functions.

  The list EXTERNAL is a list of those functions which are to be entered in the new index as being external references. (See the INDEX description) Any subfunction name which is in this list will be treated by GENLIB as a function in another Library.

- **LIST**

  -- produces a listing of the current FNLIST and EXTERNAL lists. This allows the user to see what the current functions in the new Library are. The first element of each list printed is the name of the new Library routine. The rest of the elements are the functions and forms which must be loaded by the Loader when it loaded this function. Note that in most cases the first and second names in the list are the same since the function itself must be loaded.
- OK

-- generate and write out the new Library. This command will cause the GENLIB routine to write out the new Library and index, and then stop.

- STOP

-- exit the GENLIB routine. This stops execution without generating a new Library.

Example:

The following example shows how GENLIB can be used to construct a Library.

* (GENLIB MYLIB NIL)
> LOADING: GENLIB
> ADD PRETTY (PP)
> (DISKIN TESTFYLE)
> LOADING: DISKIN
> 4 FORMS READ FROM: TESTFYLE
> ADD IF (IF)
> ADD F91 (F91)
> (EXTRACTFN CONSES)
> (CONSES)
> ADD CONSES CONSES
> LIST
> *INDEX*
> (PRETTY PP)
> (IF IF)
> (F91 F91)
> (CONSES CONSES)
> *EXTERNAL FNS*
> NIL
>
At this point we have 4 functions in the Library, PRETTY (which is a call to the function PP), IF, F91, and CONSES. Since PP is to be an external call, we must use the EXTERNAL command to indicate this.

* additive F91 IF
* EXTERNAL PP
* LIST
>
After listing the index again, we type `OK` which causes the Library to be written out to the file `MYLIB`. If we list lines -1000 to 0 of `MYLIB` we can see the index generated:

```
* (MTS "$LIST MYLIB (-1000,0)"")
> -1000 ; LIBRARY INDEX
> -999 (INDEX
> -998 (CONSES 3)
> -997 (F91 103 IF)
> -996 (IF 203)
> -995 (PRETTY PP)
> -994 ) ; END OF INDEX
# END OF FILE
> NIL
```

```
* (UNLOADFN F1 F2 ... Fn)
Purpose: used to unload the Library routines Fi.
Fi - the name of a Library routine which has been loaded.

UNLOADFN is used to remove from core the definition of Library routines previously loaded, but which are no longer required. Lisp will not actually reduce the amount of core it is using (unfortunately) but the space will be garbage collected.
```
Example:

* (LOADFN DISKIN UPDATE)
> T
* (LOADINFO)
> LOADING: LOADINFO
> LIBRARY: FRED:LIBRARY
> FNS LOADED: DISKIN PRINTMSSG PP UPDATE DISKOUT LOADINFO

* (UNLOADFN DISKIN UPDATE)
> LOADING: UNLOADFN
> (UPDATE DISKOUT DISKIN PRINTMSSG PP)

Notice that DISKIN uses PRINTMSSG as a subfunction, and UPDATE uses PP and DISKOUT as subfunctions. This is why these functions were unloaded as well as UPDATE and DISKIN.

* (LOADINFO FULL)
> LIBRARY: FRED:LIBRARY
> FNS LOADED: LOADINFO UNLOADFN
> SUBFNS:
> (9003 LOADINFO . 1) (11103 UNLOADFN . 1)
> (11201 UNLOADFUN . 1) (11301 UNLOADFUN1 . 1)
> (11401 UNLOADFUN# . 1) NIL
3 LISP FILE I/O FUNCTIONS

These I/O functions are written to provide the user with a flexible and convenient method of printing, reading and storing files of Lisp functions. These functions are used as follows:

1. Read in an MTS file containing a number of Lisp functions, using the function DISKIN.

2. Debug these programs, modifying them with the editor or by redefining or adding new functions.

3. Write the functions back to an MTS file using the function UPDATE or DISKOUT.

As well as being able to write functions into files in a Lisp readable form, it is desirable that the file be person readable. Thus, a structural print function, called a "prettyprinter", is used to write out the files. The prettyprinter uses various indentation rules based on the list structure of the function to produce an easily read listing. In addition, the prettyprinter will ensure that atoms which must be read into Lisp with enclosing quotes, such as "...", are printed in this form.

The prettyprinter is also available as a separate function which can be called from Lisp to print any Lisp functions or forms while the user is debugging or executing his programs.
In order to conveniently write out all functions read from a file, a scheme has been developed whereby the function DISKIN will "remember" what has been read in from a file, and UPDATE or DISKOUT can use that information to write the structures out again. DISKIN builds a list, called the FNS list, which contains all the information necessary to write out the file. The list is stored on the property list of the atomic file name from which the read was done, under the property tag ".FNS.". The structure of the FNS list is as follows:

(S1 S2 S3 ... Sn)

Each Si corresponds to one form or structure which was read from the file, and consequently must be written out. Each Si can be one of three forms:

ATOM - if Si is an atom, then the structure referred to is the Expr definition stored on the property list of the atom; in other words, this is the name of a function to be written.

(ATOM . TAG) - the structure referred to is the S-expression stored under the property TAG on the atom ATOM. The form read or written will be (DEFPROP ATOM TAG ... )

(ATOM . FORM) - where FORM is a non-atomic Lisp structure. The FORM itself will be written out as-is. The ATOM is ignored.

DISKIN, when reading a file, will generate a FNS list as described above and store it on the property list of the file name under the tag ".FNS.". If an S-expression read in begins
with (DEFUN ... , DISKIN generates an Si of the form ATOM above; if the S-expression begins with (DEFPROP ... , DISKIN generates an Si of the form (ATOM . TAG) above; otherwise the S-expression itself is saved as the CDR of the dotted pair (ATOM . FORM). The ATOM used as the CAR of this dotted pair is a GENSYM atom generated by DISKIN.

3.A.1 Comments

In addition to remembering which functions and forms were read in, DISKIN will also save any comments which appear before each top level function or form in the file. The comments must be on lines by themselves and cannot be inside function definitions, or they will be ignored. The comments are stored on the property list of the ATOM part of the Si described above, under the tag 
".COMMENTS.".
(DISKIN FILE1 FILE2 ... FILEn)

Purpose: reads programs in from the files specified.

FILEi - an atom, the name of the MTS file to read.

This function reads in an MTS line file. As well as reading and EVALing all forms, DISKIN generates the FNS list necessary to enable UPDATE or DISKOUT to write the file out again (see FNS description above). DISKIN will also save any top level comments it reads as described in the Comments section above.

A message is printed for each FILEi indicating how many forms were read from the file. The value returned by DISKIN is a blank.

Example:

* (DISKIN TESTFYLE FRED:PA)
> LOADING: DISKIN
> 4 FORMS READ FROM: TESTFYLE
> 5 FORMS READ FROM: FRED:PA
>

***Note: If 2 or more forms appear on the same line of a file, only the first one will be read by DISKIN. This is because DISKIN scans for comments between top level S-expressions. Files written by UPDATE or DISKOUT will never have more that one
form per line.

- (DISKOUT FILE FLIST1 FLIST2 ... FLISTn)

Purpose: writes the function definitions or forms specified by the FLISTs out to the file FILE.

FILE - an atom, the name of the MTS file which is to be written into. DISKOUT will not empty the file before writing to it.

FLISTi - either an atom or a FNS list. If FLISTi is an atom, the FNS list stored under the property "FNS." on the property list of the atom, is retrieved. If FLISTi is a list, it must have the format of a FNS list (see description of the FNS list in Section 3.A).

The FLISTi specify which functions and forms are to be written out to the file FILE. The forms are written out using the prettyprinter. Any comments associated with the forms are written out immediately before the form.

The value returned by DISKOUT is the atom DONE if the DISKOUT was successful, and FAIL if it was not (an error message will be printed if the output failed).

Example:

(DISKIN FYLE) read in the functions in FYLE

(DISKOUT NEWFYLE FYLE) write them out to NEWFYLE

(DISKOUT -A (LOADFN LOADFUN (QUOTE . MACRO))) writes out three forms to the file -A. The function definitions of LOADFN and LOADFUN are printed and the form (DEFPROP QUOTE MACRO value)

Lisp File I/O Functions
(UPDATE FILE [FLIST1 FLIST2 ... FLISTn])

Purpose: to update the contents of file FILE.

FILE - an atom, the name of the MTS file to be updated.

FLISTi - either an atom or a FNS list. This argument is the same as the FLIST argument to DISKOUT.

UPDATE writes out a file in exactly the same way as DISKOUT. The first argument, FILE, must have been read by DISKIN. All the functions and forms read from FILE are re-written to FILE, as well as any functions or forms specified by the FLIST arguments. Notice that this function differs from DISKOUT, where FILE is only the MTS file name. UPDATE treats FILE as both the file name and an FLISTi parameter.

Since UPDATE is re-writing an existing file, it is possible to lose the information in the file, due to Lisp errors, file overflows, MTS system failures etc. To prevent this kind of disaster, UPDATE makes a copy (actually an MTS RENAME) of the current contents of FILE before emptying it and writing the new information. The new file will have the name =FILE. The example below best explains the series of MTS commands which are used by the UPDATE function.

Example:

(DISKIN MYFILE) read in the MTS file MYFILE

... 

(UPDATE MYFILE) the functions read from MYFILE by DISKIN are first written to the scratch file -MYFILE. The following MTS commands are then executed:

$DESTROY =MYFILE OK
$RENAME MYFILE =MYFILE OK
$\textsc{rename} \ - \ \textsc{myfile} \ \textsc{myfile} \\
$\textsc{truncate} \ \textsc{myfile} \\
$\textsc{permit} \ \textsc{myfile} \ \textsc{like} \ \textsc{=myfile}$

The result of the above commands is to put the new version of the file \textsc{myfile} in the file \textsc{myfile}, and the previous version of the file in \textsc{=myfile}.

***Note: the file name \textsc{file} should not be of the form \textsc{id:name}, since \textsc{update} cannot rename files under another \textsc{id}.

$\textsc{diskmaxlen} \ - \ \textsc{both diskout} \ \textsc{and update} \ \textsc{allow} \ \textsc{the} \ \textsc{user} \ \textsc{to} \ \textsc{set} \ \textsc{the} \ \textsc{maximum} \ \textsc{output} \ \textsc{line} \ \textsc{length}. \ \textsc{By} \ \textsc{default}, \ \textsc{this} \ \textsc{length} \ \textsc{is} \ 105 \ \textsc{characters}. \ \textsc{?diskmaxlen} \ \textsc{may} \ \textsc{be} \ \textsc{set} \ \textsc{to} \ \textsc{any} \ \textsc{value} \ \textsc{less} \ \textsc{than} \ 255. \ \textsc{Since} \ \textsc{this} \ \textsc{length} \ \textsc{does} \ \textsc{not} \ \textsc{include} \ \textsc{the} \ \textsc{closing} \ \textsc{parenthesis} \ \textsc{which} \ \textsc{may} \ \textsc{be} \ \textsc{printed} \ \textsc{at} \ \textsc{the} \ \textsc{end} \ \textsc{of} \ \textsc{a} \ \textsc{line}, \ \textsc{it} \ \textsc{should} \ \textsc{be} \ \textsc{set} \ \textsc{to} \ 5 \ \textsc{or} \ 10 \ \textsc{less} \ \textsc{than} \ \textsc{the} \ \textsc{desired} \ \textsc{maximum} \ \textsc{length}.$

- (PP F [TAG])

Purpose: to prettyprint a Lisp structure.

\textsc{F} - the name of the thing to be prettyprinted. If an atom, the form under the tag \textsc{TAG} (default \textsc{EXPR}) is prettyprinted. If \textsc{F} is a Lisp form, the form is \textsc{eval}ed and the value prettyprinted.

\textsc{TAG} - the property list tag under which, on the atom \textsc{F}, the form to be prettyprinted is found. \textsc{TAG} defaults to \textsc{EXPR}.

The function \textsc{PP} calls the prettyprinter to print a Lisp structure directly on the user's terminal. All occurrences of the form (QUOTE form) are printed as 'form and all forms which have Breakpoints on them are printed as @form.
MAXLEN - This is the maximum length that output lines are to be printed. The default is 80 but may be reset to any value. This value is used only by PP, not by DISKOUT or UPDATE.

Example:

(PP THISFUNCTION) prettyprints the function THISFUNCTION.

(PP PICKUP THEOREM) prettyprints the form on the property list of the atom PICKUP under the tag THEOREM.

(PP (CADR (GETL 'FN ' (EXPR THEOREM)))) prettyprints either the EXPR or THEOREM definition of FN.

(PP (OR X)) prettyprints the value of X.

3.C The Prettyprint Algorithm

The primary purpose of a prettyprinter is to produce an easily read listing of a Lisp s-expression (usually a function definition). The structure of the s-expression is reflected in the way it is printed, thus parenthesization errors can be more easily detected. A secondary prerequisite, necessary for functions such as UPDATE and DISKOUT, is that the output be Lisp readable. This involves printing atoms which must be enclosed in quotes, such as "A B C", with the quotes.

The prettyprint algorithm used in this program invokes some simple overall rules governing indentation, plus some specific exceptions which make Lisp programs easier to read. The basic algorithm for prettyprinting a form is best described as

Lisp File I/O Functions
follows:

1. If the form will fit on the current output line, starting at the current indentation column, print it.

2. If the form is an atom, print it anyways.

3. (we now have a list to print) Print a left parenthesis "(".

4. Prettyprint the first element (CAR) of the structure.

5. If the first element is an atom, prettyprint the second element of the list on the same line.

6. Indent the left margin and prettyprint each of the rest of the elements of the list, one per line.

7. Print the right parenthesis ")".

Various modification to the above algorithm have been made to produce a more readable output.

1. All s-expressions of the form (QUOTE form) are printed as 'form.

2. Breakpoints are printed as @form by the function PP, and ignored completely by UPDATE and DISKOUT.

3. The local variables list of a PROG is printed as many elements per line as possible. This eliminates the annoying problem of the PROG function with too many variables to print on one line printing its variable list one per line (as it would under the rules described above).

4. The SETQ and SET functions are written so as to pair their arguments instead of listing them one per line.

5. The COND function always prints its cases on separate lines.
Related Functions

A number of other functions in this library make use of the FNS list generated by DISKIN. The function EXTRACTFPN, generates a FNS list for the functions it extracts from a library. Thus

(EXTRACTFPN PP)
(DISKOUT -FYLE PP)

will print a copy of the prettyprinter on the file -FYLE. The function SORTFILE sorts the FNS list alphabetically. XREFFILE will crossreference all the functions in a file, using the FNS list. GENLIB uses the FNS list to build a new library. The editor command PP calls the prettyprint function to print a part of the edited form.
Consider the following MTS file:

```lisp
1 ; THE 91 FUNCTION
2 (DEFUN F91 (X)
3   (IF (GREATERP X 100) (SUB X 10)
4      (F91 (F91 (ADD X 11))))
5
6 ; A LISP VERSION OF THE IF STATEMENT
7 ; (IF A B C) IS EQUIVALENT TO
8 ; IF A THEN B ELSE C
9 (DEFUN IF NEXPR (ABC)
10    (COND ((EVAL A) (EVAL B))
11      ((EVAL C)))
12
13 (DEFPROP NON SENSE (WORDS WORDS AND MORE WORDS))
14
15 (SETQ Q (LIST 'MAPC 'MAP 'MAPCAR 'MAPCAN 'MAPCON 'MAPOB))
```

We call DISKIN to read in this file.

* (DISKIN TESTFILE)
  > LOADING: DISKIN
  > 4 FORMS READ FROM: TESTFILE

* (SETQ ?MAXLEN 50)
  > 50

* (PP IF)
  > LOADING: PP
  > (NLAMBDA (ABC)
  >    (COND ((EVAL A) (EVAL B))
  >        ((EVAL C)))

* (PP NON SENSE)
  > (WORDS WORDS AND MORE WORDS)

* (PP (OR Q))
  > (MAPC MAP MAPCAR MAPCAN MAPLIST MAPCON MAPOB)

* (DISKOUT -A (IF (NON . SENSE)))
  > LOADING: DISKOUT
  > DONE

* (UPDATE TESTFILE)
  > LOADING: UPDATE
  > FILE: TESTFILE UPDATED.

Lisp File I/O Functions
4 THE DEBUG PACKAGE

The Debug Package is a set of routines which allow the Lisp programmer to monitor the execution of his program, as well as to intervene in the normal flow of its execution. A special "step" feature allows the user the option of executing the program incrementally one form at a time. Commands given to the Debug Package allow easy access to information kept on the Lisp stack, allow control to be transferred to any point on the stack, and enable the setting of "breakpoints" in a program. These Breakpoints return control to the Debug Package to give the user access to all the commands in the system.

An optional error processing routine will trap any Lisp errors which occur in the normal execution of a program. When an error occurs, the Debug Package is automatically invoked, and the user has available to him all of its commands.

The Debug Package consists of three top level library routines: DEBUG, BREAKF, and UNBREAKF. Loading any one of these functions, either explicitly via the LOADFN function, or implicitly by calling it, will cause all of the Debug Package to be loaded. As with other library functions, the UNLOADFN program will release the freespace occupied by these routines so that the garbage collector can reclaim it.
4. A  Invoking the Debug Package

The Debug Package can be invoked in one of three ways:

1. Via a Breakpoint set by BREAKF
2. By calling the function DEBUG on a form
3. When a Lisp error occurs

The form which caused the break to occur, called the breakform is passed to DEBUGMMAIN, the main routine in the Debug Package. If a break was set on a particular function, as in (BREAKF FOO), the Breakform is the first form in the body of the broken function. On a Lisp error, the Breakform is the form being evaluated when the error occurred.

Once the Debug Package has been invoked, all calls are treated in the same manner. The Debug Package allows the user to type in commands directing it to display any form or value stored on the Lisp stack, edit any form on the Lisp stack, set or remove Breakpoints, return from any form on the stack, or continue processing. Typing STOP or NIL will return to the top level of Lisp just as NIL would do in a normal Lisp error break.

4. A. 1  Debug Package Handling of Errors

The Loader routines enable the Debug Package to trap all Lisp errors. This option is initially off, and will be enabled
whenever the Debug Package is loaded.

?DEBUG - default NIL. When ?DEBUG is non-NIL, any Lisp error not trapped by the Autoloader, will cause immediate transfer to the Debug Package. This switch is set to T when the Debug Package is loaded.

## 4.B Breakpoints

A Breakpoint is a function inserted into the user's program which will cause, under certain conditions, the execution of the program to be suspended or broken. When a Breakpoint is encountered, control is passed to the Debug Package which will then give the user control over how execution is to proceed.

Breakpoints come in two varieties: Breakpoints on functions, and Breakpoints on forms within a function. Both types can optionally have a Lisp predicate associated with them which will determine each time they are encountered whether the break is to be acknowledged or ignored. A Breakpoint on a function will cause all calls to that function to be broken, subject to the predicate. A Breakpoint on a form within a function allows closer control of exactly where a break is to occur. Both types of Breakpoints are set by the BREAKF function or the BREAK command, and can be removed by the UNBREAKF
function or the UNBREAK command.

A function is broken by physically changing the body of the function. The form being broken is replaced by the form:

(BREAKPOINT FORM NAME PREDICATE)

where FORM is the form being broken; NAME is the name of the break (used by the Debug Package), and PREDICATE is the predicate which determines whether the break is to occur. PREDICATE defaults to T. If a specific function is broken, FORM is the first form within the body of the Lambda expression defining the function. This means that a break of a function name will break after its arguments have been processed and the Lambda variables have been bound, while a break on a specific form in a function occurs before the form has been evaluated.

The Prettyprinter will print the symbol @ in front of any broken form in the function being printed. The functions UPDATE and DISKOUT will ignore any Breakpoints and print the original function. This means that functions do not have to be unbroken (using UNBREAKF or the command UNBREAK) before they are written on a file using UPDATE or DISKOUT.

?BREAKSW - this is a switch which when set to NIL causes all Breakpoints to be ignored. The switch is set to T whenever BREAKF is called, and set to NIL when the function UNBREAKF is called with no arguments, as in (UNBREAKF). In particular, calling (BREAKF) will turn on the processing of Breakpoints and calling (UNBREAKF) will turn off the

The Debug Package
processing of Breakpoints. It is important to observe that (UNBREAK) does not remove Breakpoints, it only stops them from being processed.

?BROKEN - is a global variable which is set to the list of all current Breakpoints. BREAKF adds its arguments to this list, and UNBREAKF removes its arguments from the list.

Example:

consider the following functions:

> (PP SUM)
> (LAMBDA (A B) (ADD (FACT A) (FACT B)))
> (PP FACT)
> (LAMBDA (X)
> (COND ((LESSP X 2) 1)
> ((TIMES X (FACT (SUB1 X))))))
> (DEBUG (SUM 3 4))
> LOADING: DEBUG
0=>(SUM 3 4)
@ BREAK FACT
0=> BREAK @ FACT
@

We now have set a Breakpoint on the function FACT. Execution continues by using the CONTINUE command:

@ @ CONTINUE
0=> AT: FACT
@ PRINT
0=>(COND ((LESSP X 2) 1) ((TIMES X (FACT (SUB1 X)))))
@ CONTINUE
0=> AT: FACT
@ PRINT
0=>(COND ((LESSP X 2) 1) ((TIMES X (FACT (SUB1 X)))))
@ CONTINUE
0=> AT: FACT
@ BKF
0=>COND
0=>FACT
0=>TIMES
0=>COND
0=>FACT
0=>TIMES
0=>COND
0=>FACT

The Debug Package
At this point we have re-entered FACT 3 times recursively, breaking on the first form in FACT (the COND) each time. We now restart the execution of SUM by use of the GO command:

```
GO SUM
0=>(SUM 3 4)
UNBR FACT
BREAK (FACT IN SUM)
0=> BREAK # (FACT IN SUM)
```

Having cleared the old break of FACT and set a Breakpoint on the calls to FACT within the function SUM, we first STEP to the first form in SUM, then continue execution:

```
STEP
0=>(ADD # (FACT A) # (FACT B))
CONTINUE
0=> AT: (FACT IN SUM)
PRINT
0=> (FACT A)
CONTINUE
0=> AT: (FACT IN SUM)
PRINT
0=> (FACT B)
CONTINUE
> 30
```

Notice that this time we only stopped twice, once for each call of FACT from within SUM, but not for the recursive calls to FACT.

4.B.1 Pseudo-breakpoints

In order to give the user the ability to "step" through a program the Debug Package sets up pseudo-breakpoints in the user's programs. A pseudo-breakpoint behaves like a Breakpoint set by the BREAKF function, but does not really appear in the user's
program. It is set by using the Lisp STEP function, which causes the Lisp interpreter to signal an error after a specified number of internal calls to EVAL. The error is trapped by the Debug Package which in turn allows the user to continue typing commands. The Debug Package will not set a pseudo-breakpoint on the functions QUOTE, LAMBDA, FLAMBDA, NLAMBDA, DUMP, or STEP. Thus, in stepping through a program, the Debug Package will not stop at any of these functions, but will pass them and stop at the next function.

### 4.C The Stackpointer

The commands GO, FIND, and EDIT all allow a series of locator parameters which are used to set an internal stackpointer to a particular form on the Lisp stack. The locators are atomic function names. Each locator is processed in order and results in moving the stackpointer down the stack (towards top level Lisp) to point to the next occurrence of a call to the function specified by the locator. Consider the situation in which the stack has on it the forms:

```lisp
(ADD ...)
(COND (...
(F ...)
(ADD ...)
```

The Debug Package
(COND (\(\ldots\)
  (F \(\ldots\))

The locators COND ADD would set the Stackpointer to the fourth form from the top of the list. From this point, the locator F would set the Stackpointer to the bottom form in this list.

Once the Stackpointer has been set, the actions specified by the commands EDIT, GO, and VALUE are executed as if the user were at that point in the execution of the program. The command TOP is used to reset the Stackpointer to the top of the Lisp stack, that is, pointing to the Breakform.

\section{4.D \ The Functions}

\* (BREAKF F_1 F_2 \ldots F_n) \*

Purpose: this function allows the user to set Breakpoints in his program. The function is a Pexpr which will take an indefinite number of arguments.

Fi - the specification of where a Breakpoint is to be set. If Fi is an atom, it must be the name of a user defined function which is to be broken. The break is unconditional; each call to this function will cause a break to occur. Because it is necessary to change the actual Lambda expression of the function definition, only Exprs may be broken, not Subrs.
If $Fi$ is a list of the form $(Fn_1 \text{ IN } Fn_2)$, all calls to the function $Fn_1$ which occur in the body of the function $Fn_2$ will be broken. As with the previous case, $Fn_2$ must not be a Subr. The function $Fn_1$ may be either an Expr or a Subr.

Breakpoints may have associated with them a predicate which will be evaluated when control reaches the Breakpoint. If the predicate evaluates to NIL the Breakpoint is ignored and execution proceeds normally. If the predicate evaluates to a non-NIL value, the break is taken.

If a predicate is to be specified at a Breakpoint, $Fi$ must be a list of the form $(\text{WHERE PREDICATE})$ where the argument WHERE is either a function name or a list of the form $(Fn_1 \text{ IN } Fn_2)$, as described above, and PREDICATE is the Lisp form which will be evaluated as the predicate test. Thus, the four possible forms of the argument $Fi$ are:

- FOO
- $(\text{FOO1 IN FOO2})$
- $(\text{FOO (EQ A B)})$
- $((\text{FOO1 IN FOO2}) \text{ (ATOM X)})$
• (UNBREAKF F1 F2 ... Fn)

Purpose: to remove Breakpoints set by the function BREAKF.

Fi - the specification of what is to be unbroken. This argument has the same form as the two BREAKF argument forms which do not specify a predicate. If Fi is an atom, the function of that name has its break removed; if Fi is a list of the form (Fn1 IN Fn2), all Breakpoints of the function Fn1 within the function Fn2 are removed.

The special case of a call to UNBREAKF with no arguments will turn off all Breakpoints temporarily, until the function BREAKF is called again (See BREAKF above).

• (DEBUG FORM)

Purpose: to call the Debug Package explicitly on the form FORM.

FORM - an arbitrary Lisp form.

Example:

(DEBUG (F 27)) begins a call to the function F and immediately calls the Debug Package with (F 27) as the first Breakform.

Examples:

Consider the function F defined as follows:

*  
*  (PP F)
>  (LAMBDA (X)
>    (COND ((GREATERP X 100) (SUB X 10))
>           (T (F (F (ADD X 11)))))
>
We first set a Breakpoint on the function F:

*  
*  (BREAKF F)
0=> BREAK @ F
>  DONE
*  (F 27)
0=> AT: F
@ CONTINUE
0=> AT: F
@ UNBR F

The Debug Package
Notice that this type of Breakpoint causes a break on each recursive call to F. The final value returned is 91.

Now we set Breakpoints on specific forms with the function F:

```
* (BREAKF (ADD IN F) (SUB IN F))
0=> BREAK @ (ADD IN F)
0=> BREAK @ (SUB IN F)
> DONE
* (F 34)
0=> AT: (ADD IN F)
@ CONTINUE
0=> AT: (ADD IN F)
@ CONTINUE
0=> AT: (ADD IN F)
@ UNBR (ADD IN F)
@ CONTINUE
0=> AT: (SUB IN F)
@ STOP
> **STOP**
*
```

In the next case, we turn off all breaks by calling UNBREAKF with no arguments.

```
* (UNBREAKF)
> DONE
* (F 45)
> 91
*
```

If we call BREAKF again, the breaks will be turned on again.

```
* (BREAKF)
> DONE
* (F 56)
0=> AT: (SUB IN F)
@ STOP
> **STOP**
* (UNBREAKF (SUB IN F))
> DONE
*
```

We next set a conditional break on the call to COND within the function F.

```
* (BREAKF ((COND IN F) (EQ X 91)))
0=> BREAK @ (COND IN F)
```
> DONE
* (F 54)
  O=> AT: (COND IN F)
  AT: X
  O=> X = 91
  AT: CONTINUE
  O=> AT: (COND IN F)
  AT: X
  O=> X = 91
  AT: BKF
  O=>COND
  O=>F
  O=>COND
  O=>F
  O=>F
  O=>COND
  O=>F
  O=>F
  O=>F
  AT: CONTINUE
  O=> AT: (COND IN F)
  AT: CONTINUE
  O=> AT: (COND IN F)
  AT: CONTINUE
  O=> AT: (COND IN F)
  > 91

Notice that in this last instance that although the COND has been executed many times (as is indicated by the backtrace printed by BKF), the break occurred only when the value of X was 91.
The following list of commands are broken loosely into three groups; the commands which control or alter the flow of execution of the user's program, those which find and look at forms on the Lisp stack, and those which manipulate Breakpoints. Some commands, such as GO, could belong logically to more than one of these groups. The division is somewhat arbitrary.

4.E.1 Input Format

In order to distinguish user typed input lines from lines printed by the Debug Package, and to aid the user in determining when the Debug Package is expecting input, prefix characters are used on all input and output in the Debug Package. The single symbol "S" precedes all lines of input and is typed by the system whenever the Debug Package expects a command to be typed in. The symbols "n=>" where n is an integer precedes each line of output. The number n indicates the break level that the Debug Package is currently operating in. This level number is normally 0, but if an error occurs in the evaluation of a form from within the Debug Package, the level number is increased by one.

The Debug Package will allow multiple commands on a line.
Where ambiguity may occur between the optional parameters of one command and the next command, the atom : may be used as a separator. The Debug Package treats each end-of-line as if there were a :, hence commands may not span more than one line.

If a Lisp form or an atom which is not a command name is entered, the Debug Package will print the result of EVALing the form, or print the value of the variable.

- HELP

-- gives a short (1 line) description of each command available in the Debug Package. The description is not intended to be in any way complete - it should only serve as a reminder of what a command does or what it expects as an argument.

4. E.2 Program Control Commands

- CONTINUE Abbreviation: CO C

-- continue execution of the user's program, starting with the Breakform. If the Breakform has already been EVALed by the command EVAL, it will not be re-evaluated. No pseudo-breakpoint is set, only an error or Breakpoint will cause the re-invocation of the Debug Package.

The Debug Package
• **EVAL**

  Abbreviation: **E**

  -- evaluate the Breakform and print its value. The value is retained so that the CONTINUE and NEXT commands will not re-evaluate the form. If an error occurs in EVALing the Breakform, the Debug Package is re-invoked at a higher level number. The UP command describes how to return from this new break level.

• **GO [LOC1 LOC2 ... LOCn]**

  -- go to the place on the Lisp stack determined by the locators LOC1 LOC2 ... LOCn. The LOCi are stack locators as described in the section above on the Stackpointer.

  After resetting the Stackpointer, the command GO transfers to the form pointed to by the Stackpointer. A pseudo-breakpoint is set at the new form causing the Debug Package to immediately break on this form. The effect of the GO command is to restart the execution of the user's program at some previous function call. This can be useful when variable bindings may be incorrect due to some error in the program. Restarting the PROG or Lambda expression which bound them will cause these variables to be rebound.

• **NEXT**

  Abbreviation: **NX N**

  -- the current Breakform is evaluated if it has not already been EVALed by the EVAL command, and control transfers to the next form in the user's program. This enables the user to avoid stepping through all the function calls in the Breakform but still causing the Debug Package to stop at the next form after the Breakform. This can be particularly useful in functions which contain a series of forms or statements such as PROGs, LAMBDA,s, CONDs, SELECTQs, etc.

• **RETURN FORM**

  Abbreviation: **RET R**

  -- the form FORM is evaluated and returned as the value of the Breakform. A pseudo-breakpoint is set for the next form in the user's program. As with EVAL, if an error occurs in the evaluation of FORM, the Debug Package is re-entered one break level higher. The UP or ! command can be used to return to the previous break level.

  RETURN provides a convenient way of returning the "correct" value from a function which is not working correctly.

The Debug Package
• **STEP [N]**
  
  Abbreviation: ST S
  
  -- execution of the Breakform continues, with a pseudo-breakpoint set for the Nth form to be executed. The default value of N is 1. Thus STEP 1 will break after one form has been evaluated, STEP 3 after three forms have been evaluated, etc. Note that unlike NEXT and CONTINUE, STEP begins evaluation of the Breakform regardless of whether it has been explicitly EVALed by the EVAL command.

• **STOP**
  
  Abbreviation: NIL ||
  
  -- stops execution of the program and returns to the top level of Lisp. Thus NIL typed after an error break has the same effect as it does when the Lisp error breaks are in effect.

• **UP**
  
  Abbreviation: |
  
  -- used when an error or break has occurred in the evaluation of a form from within the Debug Package. In this case the output prefix will be n=> with n non-zero (the normal prefix is 0=>). This command will return to the previous level of the Debug Package. The return is not to the command which caused the error, but back to the command processor. The Stackpointer will be reset to the Breakform and the Debug Package will start processing commands again.

• **USE FORM**
  
  -- used to correct undefined function or undefined atom errors. Often these errors are caused by misspelling an atom or function name. The USE command can be used to correct these errors without the necessity of calling the Editor. The value of FORM is substituted for the erring atom or function name. The change is permanent; it is done by a RPLACA on the actual form in the program. After making the change, the corrected form is printed out.
4.3 Stack Commands

These commands manipulate and display information stored on the Lisp stack. The print commands ? and P are included in this group since they print the Breakform, which is the top form on the stack.

- ?
  -- print the current Breakform. The complete form is printed including any calls to BREAKPOINT (see PRINT below).

- BACK [N]
  Abbreviation: BKE BK
  -- print a backtrace of N forms on the stack, starting at the point specified by the Stackpointer, which is normally set to the Breakform. The default value for N is 10. If the global switch ?TERSE is non-NIL (the default), only one line is printed for each form. Only functions in the user's program appear in the backtrace; if any functions which are part of the Debug Package are on the stack they are indicated by a series of three dots.

  If N is the atom FULL, a full backtrace will be printed. This is effectively equivalent to an infinite value for N.

- BKF [N]
  -- print a backtrace of the names of N forms on the stack. This command works the same as BKE above, but prints only the function names, not the whole form.

- EDIT [LOC1 LOC2 ... LOCn]
  Abbreviation: ED
  -- calls the Lisp editor on the form on the stack indicated by the Stackpointer. The locators LOCi are first used to set the Stackpointer. In the simplest case, if the first command entered after a break is EDIT, the form broken is sent to the editor. The editor command OK will return control to the Debug Package.
• FIND [LOC1 LOC2 ... LOCn] Abbreviation: F

-- this function is used to set the Stackpointer. The form pointed to by the Stackpointer is printed out.

• PRINT

  Abbreviation: P

-- prints out the form pointed to by the Stackpointer. The form is printed in a terse (one line) mode and has all quoted functions printed as 'FORM and all Breakpoints printed as @FORM, where FORM is the broken form. This same print routine is used by BACK and FIND to print forms on the stack.

• TOP

-- resets the Stackpointer back to the top of the stack (the Breakform).

• VALUE FORM

  Abbreviation: V

-- prints out the evaluation of FORM at the point on the stack indicated by the Stackpointer. This command allows the user to display the value of variables bound on the stack at some previous level of a recursion. FORM may contain function calls to simple list searching or predicate Subrs. The execution of FORM is simulated by the VALUE command since Lisp only has the facility to find values of atoms bound on the stack, not forms. The VALUE command determines the values of all atoms in the FORM, then applies the functions of the FORM to these values.

Possible functions which can be used in the FORM are: CAR, CDR etc., NTH, ATOM, LISTP etc.

4.E.4 Breakpoint Commands

The following commands have similar effects to the functions BREAKF and UNBREAKF, and can be used in conjunction with these functions. The command ?BREAK evaluates the global...
list $\text{?BROKEN}$ which is a list of all Breakpoints currently set.

- $\text{?BREAK [F}_1 \ F_2 \ ... \ F_n]\quad$ Abbreviation: $\text{?BR}$
  
  -- prints a list of the broken functions. If no arguments are given, a complete list of all current Breakpoints is printed. If $F_i$ is a function name, all breaks of the form $(F \text{N IN } F_i)$ are printed, as well as indicating whether the function $F_i$ is broken.

- $\text{BREAK [F}_1 \ F_2 \ ... \ F_n]\quad$ Abbreviation: $\text{BR}$
  
  -- sets Breakpoints at the locations specified by the $F_i$ in the same way as the function $\text{BREAKF}$ does. As in the function $\text{BREAKF}$, if $F_i$ is an atom, the Expr $F_i$ is broken, if $F_i$ is of the form $(F_{n1} \text{ IN } F_{n2})$, all calls to $F_{n1}$ from within $F_{n2}$ are broken. If $F_i$ is of the form $(\text{WHERE PREDICATE})$, the Breakpoint at location WHERE (which must be of the form $F_i$ or $(F_{n1} \text{ IN } F_{n2})$ as above) is made conditional on the form PREDICATE being non-NIL.

  As with $\text{BREAKF}$, typing the command $\text{BREAK}$ with no arguments, will reset the global switch $\text{?BREAKSW}$. (See the section on Breakpoints for a more detailed description of how Breakpoints are implemented.)

- $\text{UNBREAK [F}_1 \ F_2 \ ... \ F_n]\quad$ Abbreviation: $\text{UNBR}$
  
  -- removes the Breakpoints set by the $\text{BREAK}$ command or the $\text{BREAKF}$ function. The arguments $F_i$ are of the same form as in the function $\text{UNBREAKF}$; if $F_i$ is a function name, that function is unbroken, if $F_i$ is of the form $(F_{n1} \text{ IN } F_{n2})$ all Breakpoints of the function $F_{n1}$ in the body of the function $F_{n1}$ are removed.

  As with $\text{UNBREAKF}$, typing the command $\text{UNBREAK}$ with no arguments will set the switch $\text{?BREAKSW}$ to NIL, thus disabling all breaks temporarily. (See the $\text{BREAKF}$ and $\text{UNBREAKF}$ functions, and the description of Breakpoints)
4.E.5 Restrictions

1. The Debug Package uses the STEP function to set pseudo-breakpoints, hence the user should avoid using this function in his own programs.

2. The Debug Package will step over any calls to the functions STEP, LAMBDA, FLAMBDA, NLAMBDA, DUMP, BREAK, and QUOTE.

3. Attention and program interrupts are not trapped by the Debug Package. They are handled in the usual way by the Lisp error handling routines.


5. Using the STEP, NEXT, and RETURN commands it is possible to set a pseudo-breakpoint at the end of the program which will not be in the user's program. In this case the next form typed to Lisp after the Debug Package has been exited may re-invoke the Debug Package. This can be very confusing. Typing the command CONTINUE will allow the program to continue.

Example:

The following example illustrates how the Debug Package error handling works:

```lisp
* (PP PRLEN)
> (LAMBDA (X)
> (COND ((ATOM X) (PLEN X))
> ((CDR S)
> (ADD 1
> (PRLEN (CAR X))
> (PRLEN (CDR X)))))
> ((ADD 2 (PRLEN (CAR X)))))
> *
> (LOADFN DEBUG)
> T
> *
> (PRLEN * (THIS IS (A LIST)))
> ***16 UNDEFINED ATOM
+ S
+ 0=>(CDR S)
@
```

We have defined a function PRLEN, and loaded the Debug Package to monitor the execution of this function. When the error is detected we are immediately transferred to the
Debug Package with the Breakform set to the form which caused the error.

@ VALUE X
0=> X = (THIS IS A LIST)
@ VALUE S
0=> S = *UNDEF*
@ USE X
0=> (CDR X)

After checking the values of X and S, we use the USE command to correct the error.

@ STEP
0=>(ADD 1 (PRLEN (CAR X)) (PRLEN (CDR X)))
@ STEP
0=>(PRLEN (CAR X))
@ NEXT
0=>(PRLEN (CDR X))
@ X
0=> X = (THIS IS A LIST)
@ EVAL
0=> 13
@ CONTINUE
> 18

The STEP and NEXT functions are used to proceed with the execution of the program. The command EVAL evaluates the Breakform (PRLEN (CDR X)) which yields the value 13, then CONTINUE continues execution which finally results in the value 18 for the function.
The functions described in the section are a group of utility routines which provide the user with information about the structure of his programs. The function FNMAP allows the user to find out what functions are currently defined, and FNSIZE gives the size of Lisp functions. Finally, the functions XREF and XREFFILE provide a structural analysis of a program or group of programs.

* (FNMAP [FILE] [TAGS])

Purpose: this function produces a list of all atoms on the OBLIST which have a particular property tag.

FILE - the MTS file or device on which the output is printed. The default is *SINK*, the user's terminal.

TAGS - a list of property tags that FNMAP will look for. The default is (EXPR). Note that if this argument is specified, the FILE argument must also be specified.

FNMAP searches the OBLIST testing each atom for the properties specified. A table is produced of the atoms and the property tag that was found.

Example:

(FNMAP) gives a table of all Exprs currently defined.

(FNMAP *SINK* (SUBR FSUBR NSUBR)) gives a table of all Subr type functions.

(FNMAP -A (GRAMMAR THEOREM)) prints a table of all atoms with either the tag GRAMMAR or THEOREM onto the MTS file -A.
• (FNSIZE F1 F2 ... Fn)
Purpose: to determine the size of the functions Fi.
Fi - an atom, the name of the function to be analyzed.

FNSIZE counts the number of cons cells in the definition of each function Fi and prints a table of the function names, their sizes, and the total size of all the functions.

Example:

(FNSIZE FNSIZE) gives the size of the function FNSIZE.

(FNSIZE DISKIN DISKIN1 DISKINCOM) prints a table of the sizes of subfunctions of the DISKIN routine.

5. A n The Cross Reference Program

The cross referencing program, XREF, is used to analyze the structure of Lisp programs. Calling (XREF FN) where FN is a Lisp Expr, will produce a structural analysis of FN. The analysis includes a list of all other Exprs called by FN, all local and global variables used by FN, optionally all Subrs called by FN, and a list of all other Exprs analyzed by XREF which call FN. As well as analyzing FN, XREF also produces the same information about all functions called by FN, and all functions they call etc.

A number of global switches are used by the XREF routines
to allow the user some control over what functions are or are not to be analyzed by the cross referencer. These switches are described at the end of this section.

- `(XREF F [OUTFILE])`

Purpose: this function produces a cross referencing of the functions specified by F.

F - either a single name of a function which is to be cross referenced, or a list of functions to be cross referenced.

OUTFILE - the name of the MTS file or device where the output is to be printed. The default OUTFILE is *SINK*, the user's terminal.

XREF prints, for each function in the list F, a tree indicating which functions call which other functions. The tree is printed horizontally on the page, starting at the top left and growing towards the bottom right of the page. If the argument F is a list of functions, a separate tree is printed for each function in F. Each function is expanded only once in the tree, at the time it is first printed on the page.

After the tree is printed, each of the functions are listed, alphabetically, along with the following information:

1. Its parameter list.
2. Its type (EXPR, FEXPR, NEXPR).
3. Any comments which may have been associated with the function by the DISKIN routine.
4. A list of other EXPRs called by this function.
5. A list of Exprs which XREF has encountered in this analysis which call this function.
6. A list of local variables. These are all PROG and Lambda variables in the body of the function.

7. A list of all global variables used by this function.

8. Optionally, a list of Subrs called by this function.

A number of switches can be set to control some of the decisions that XREF makes. These switches allow the user to specify the cross referencing of Subrs, Library functions, and inhibit cross referencing certain Exprs which would normally be looked at. All switches default to their most common setting.

XREFEXPRS - default T. If set to a list of function names, only these functions will be analyzed by XREF. If set to T, the default, all Exprs called by a function being analyzed by XREF will themselves be analyzed. Library Exprs are treated separately. If XREFEXPRS is set to NIL, only functions explicitly mentioned in the P argument of XREF will be analyzed.

XREFLIB - default NIL. If set to a list of Library function names, these functions will also be analyzed by XREF. If set to T, all Library functions will be XREFed. If set to NIL, the default, Library routines will be listed as being Exprs called by a function, but the routines themselves will not be XREFed.

XREFNOT - default NIL. If any of the variables XREFEXPRS, XREFLIB, or XREFSUBRS is set to T, this list can be used to override the XREFing of selected functions. Any function name in this list will not be XREFed, despite the setting of the other variables.
XREFSUBRS - default NIL. If set to a list of Subr names, any occurrences of these Subrs in one of the functions being XREFed will result in that Subr being mentioned in the table of information printed for the function. If set to T, all Subrs in the function being XREFed will be listed. If XREFSUBRS is NIL, no Subrs will be XREFed.

Example:

In this example we call XREF on the function LOADFN of the loader routines. The list XREFSUBRS is set to the list of functions which we want the XREF routine to tell us about for each function it encounters.

* (SETQ XREFSUBRS '(PROG LAMBDA RETURN GO))
> (PROG LAMBDA RETURN GO)
* (XREF LOADFN)

CROSS REFERENCE FOR: LOADFN
MAR 28, 1974 20:20:16

LOADFN LOADFUN LOADSETUP LOADINDEX READFROM
LOADFUN# READFROM
LOADFUN

=================================

LOADFN (L) PXEXPR
CALLS: LOADFUN
CALLS SUBRS: 
CALLED BY: 
LOCAL: L
GLOBAL: 

LOADFUN (FN) EXPR
CALLS: LOADSETUP LOADFUN# LOADFUN
CALLS SUBRS: PROG RETURN LAMBDA
CALLED BY: LOADFUN LOADFN
LOCAL: FN TOPCALLS LINE LIST
GLOBAL: ERROUT #LIBRARYLIST#

LOADFUN# (LINE) EXPR
CALLS: READFROM
CALLS SUBRS: 
CALLED BY: LOADFUN
LOCAL: LINE
GLOBAL: ENTRY LOADMAP BUFFER FORM NAME

LOADINDEX (LIB) EXPR
CALLS: READFROM
CALLS SUBRS: PROG RETURN
CALLED BY: LOADSETUP

Function Analysis Routines
The tree structure indicates that LOADFN calls only the function LOADFUN. LOADFUN calls LOADSETUP, LOADFUN#, and itself. LOADSETUP calls LOADINDEX which in turn calls READFROM. LOADFUN# calls only READFROM.

* (XREFFILE FILE [OUTFILE])

Purpose: to cross reference all functions in the file FILE.

FILE - the file whose functions are to be XREFed. If FILE has not been read by DISKIN, it is read in first.

OUTFILE - default is *SINK*. This is the file to which the output from the cross referencer is to be printed.

XREFFILE takes the FNS list of functions generated by DISKIN, removes all non-Exprs, and passes this list to the function XREF. The OUTFILE argument of XREFFILE is the same as the one for XREF.

This function provides a convenient way of cross referencing a large Lisp program, especially if the user is not familiar with the programs in the file. All switches and options of XREF apply to XREFFILE.
Example:

(XREFFILE BATC:COMPILER2) will produce a (long) cross reference of all functions in the Lisp compiler.
This section includes a number of Library routines which do not conveniently fit under any of the major classifications in this document. Many of them are used as subfunctions of other Library routines and are listed here because they may be of use or interest to others.

\[ (*\text{PUTOB } A_1 A_2 \ldots A_n) \]

Purpose: to add atoms to the OBLIST.

\( A_i \) - an atom, a copy of which will be put on the Lisp OBLIST.

This function adds copies of its arguments to the OBLIST. Any READs which occur after the execution of this function will find the new atoms instead of the previous ones. The original atoms are left on the OBLIST, complete with their values and property lists. Any functions which were read in prior to the call to *PUTOB will still use the old atoms.

This function is used in conjunction with *REMOB to effectively isolate atoms which might otherwise cause conflicts in functions which EVAL some of their parameters or PROG variables. *PUTOB and *REMOB are actually part of the Loader routines; they are not part of the Library.
- (**REMOB A1 A2 ... An)

Purpose: to remove atoms from the OBLIST.

Ai - an atom to be removed.

**REMOB is the NSUBR version of the Lisp function REMOB. This means that it is the same as REMOB but does not EVAL its arguments. It is used in conjunction with *PUTOB to isolate Lambda and PROG variables.

Example:

```
* (*PUTOB A B C)
> NIL
* (DEFUN IF (A B C)
  > (COND ((EVAL A) (EVAL B))
  > ((EVAL C))))
> IF
* (*REMOB A B C)
> A
* (SETQ B 2)
> 2
* (IF NIL (CONS 1 2) B)
> 2
```

If the *PUTOB and *REMOB were not present, the IF call would return the value (CONS 1 2) instead of 2. This is due to the fact that B is bound within the function IF to IF's second argument. When B is EVALed, it is its local value that is returned, not its global value. *PUTOB and *REMOB solve this problem by making the atom B in the function IF a different atom from the atom B used elsewhere in Lisp.

- (CASE N S1 S2 ... Sn)

Purpose: a Lisp CASE statement.

N - a Lisp expression which evaluates to an integer number.

Si - any Lisp expression.

The value of CASE is the value of S(N), where N is the first argument. If N is less than 1, the value is Sn, the last
expression. If \( N \) is greater than the number of \( S_i \), the value is NIL.

- \((\text{CONSES } S)\)
  
  Purpose: counts the number of cons cells in the structure \( S \).
  
  \( S \) - any arbitrary Lisp expression.

  This function is used by \( \text{FNSIZE} \) (described elsewhere) to give the number of cons cells in a Lisp function. If \( S \) contains a circular list, \( \text{CONSES} \) will never return.

- \((\text{DREVERSE } L)\)
  
  Purpose: to reverse the list \( L \).
  
  \( L \) - any Lisp list.

  \( \text{DREVERSE} \) behaves exactly like \( \text{REVERSE} \) except that the original list is modified instead of making a copy.

- \((\text{EDIT F})\)
  
  Purpose: calls the editor on the form \( F \).
  
  \( F \) - any Lisp expression. The value of this expression is passed to the editor.

  This function is the editor written by Paul Friedman, which is documented in the file \( \text{PSIL:MINIED.FMT} \). This Library contains a copy of this editor with small changes to interface it to the prettyprinter described in the document.

Miscellaneous Library Routines
• (EDITF F1 F2 ... Fn)

Purpose: calls the editor on the functions Fi.
Fi - the name of a Lisp Expr to be edited.

EDITF is the alternate entry point to the Lisp editor. See EDITE above.

• (ELEM L N)

Purpose: returns the Nth element of list L.
L - should evaluate to a Lisp list.
N - must evaluate to an integer number.

ELEM return the Nth element of the list L. If N is less than 1, the last element of L is returned, if N is greater than the length of the list L, NIL is returned.

• (INFO)

Purpose: prints current storage, time, and cost information.

This function prints out the elapsed and CPU time for the current Lisp run, the current stack and freespace allocations, and the MTS VMSIZE and total cost information.

Example:

* (INFO)
> ELAPSED TIME SINCE START OF RUN = 8 MINUTES 26 SECONDS
> CPU TIME = 1 SECONDS.
> CURRENT FREESPACE ALLOCATION = 6. PAGES.
> CURRENT STACK ALLOCATION = 1. PAGES.
# VMSIZE = 35 PAGES
# COST = $2.93
>

Miscellaneous Library Routines
• (MT "cmd")

Purpose: to execute the MTS command specified.

The MT function converts the command specified by the arguments to MT into proper MTS command format. It adds commas and removes excess blanks from the command.

Example:

(MT L MYFILE (10 20))
= $L MYFILE(10,20)

(MT PERMIT (A B) LIKE FOO)
=$PERMIT (A,B) LIKE FOO

• (PRINTL S1 S2 ... Sn)

Purpose: used to print messages.

Si - either an atom or a list of S-expressions.

If Si is an atom, it is printed (with a PRIN1), if it is a list, each element of the list is EVALed and printed. The atom : will cause a carriage return and 5 space indent. The value of the function PRINTL is NIL.

Example:

* (SETQ X 3 Y 5)
> 5
* (PRINTL X= (X) AND Y= (Y))
> X= 3 AND Y= 5
> NIL
* (PRINTL : X= (X) : Y= (Y) : X+Y= ((ADD X Y)))
> X= 3
> Y= 5
> X+Y= 8
> NIL
*
* (PRINTMSSG BUFFER S1 S2 ... Sn)

Purpose: to print a message to a particular buffer.

BUFFER - the name of a buffer into which the message is to be printed. The buffer must have previously been opened.

Si - either an atom, or a list of S-expressions.

This function is the same as PRINTL above, but allows the specification of a buffer into which the message is printed. PRINTL uses the buffer LISPOUT.

Example:

* (SETQ X 22)
> 22
* (PRINTMSSG LISPOUT THE VALUE OF X IS (X))
> THE VALUE OF X IS 22
> NIL
* (SETQ X *THEUNDEFINEDATOM)
> THEUNDEFINEDATOM
* (PRINTMSSG LISPOUT *ERROR* : (X) IS UNDEFINED)
> *ERROR*
> THEUNDEFINEDATOM IS UNDEFINED
> NIL
*

* (SETELEM L N V)

Purpose: to set the Nth element of L to V.

L - any Lisp list.

N - evaluates to an integer.

V - evaluates to the new value for the Nth element of L.

This function does not make a copy of argument L, the change is a destructive one.
Example:

```
* (SETQ X ' (A B C))
> (A B C)
* (SETELEM X 2 ' (X Y Z))
> ((X Y Z) C)
* X
> (A (X Y Z) C)
*
```

* (SORTFILE FILE)

Purpose: sorts the functions in a file alphabetically.

FILE - the name of the MTS line file to be sorted.

If FILE has not been read in by (DISKIN FILE), it is read. The FNS list on the property list of the file name is passed to the function SORTLIST. The sorted list is put back on the property list of the file name. If UPDATE or DISKOUT is then called to write out the file, it will be printed in alphabetical order.

Example:

```
(DISKIN MYFILE) ; read the file
(SORTFILE MYFILE) ; sort it
(UPDATE MYFILE) ; write it out again
```
(SORTLIST LST F)

Purpose: to sort the list LST alphabetically.

LST - evaluates to the list to be sorted.

F - must evaluate to a Lisp function of one argument.

The function F is applied to each element of the list LST to retrieve the sort key. The value of the function SORTLIST is the new (sorted) list.

Example:

* (SETQ X * ((A Y) (D X) (B V) (C W)))
  > ((A Y) (D X) (B V) (C W))
* (SORTLIST X 'CAR)
  > ((A Y) (B V) (C W) (D X))
* (SORTLIST X 'CADR)
  > ((B V) (C W) (D X) (A Y))
* (SETQ Y * (A E C D F B))
  > (A E C D F B)
* (SORTLIST Y 'QUOTE)
  > (A B C D E F)
*
Although all of the programs described in this thesis are currently in use, there exist many rough spots in the implementation.

The entire system suffers from the lack of a Lisp compiler for Lisp/MTS. Compiling the programs would not only improve the speed and space requirements of the programs, but would reduce the conflicts between user and Library function names and variables. It is hoped that a compiler will be available for Lisp/MTS in the near future.

The prettyprinter appears to perform well over a large class of user programs. The special heuristics used to improve the visual structuring of particular Lisp functions work well, but might be extended to a number of other functions. The basic duality of programs and data which exists in Lisp will still cause the prettyprinter to produce "incorrect" structuring. The prettyprinter cannot know whether a particular list structure is a program segment or a data structure. If it assumes that it is a program, data may be visually structured incorrectly. If the list is assumed to be data, programs will not be properly prettyprinted. This problem is unresolvable; the best that one can hope for is a program which "almost always" works correctly.

There exist many problems with the Debug Package both in implementation and in user interaction. Because of the order in which Lisp processes function calls, it is possible to correct an error with the Debug Package only to find that the
interpreter does not "see" the correction. Nevertheless, if execution is restarted from some higher level, the program will work correctly. In most cases it is obvious when execution must resume at some prior point in the evaluation of the program. There are however, situations in which execution should be restartable at a given point, but is not. This problem is due to the order in which the interpreter processes Lisp function calls. The problem cannot be corrected without major changes in the internal structure of the Lisp interpreter.

Clearly, the Debug Package is designed for use by sophisticated Lisp programmers. A novice will be more confused than helped by this facility. If the system is to be used for teaching Lisp, it would be wise to augment it with a simplified "Mini-Debug Package".

The command language of the Debug Package, while quite comprehensive, requires some extensions. Some sequences of commands would be better if merged into a single command. A macro facility would offer the user the ability to adapt the Debug Package to his particular style of debugging by allowing him to write his own "best" set of commands.

A useful extension to the Library system would be the inclusion of "undoable" functions. These could be used in conjunction with the Debug Package to allow the user to "back up" the execution of his programs to an arbitrary previous state. In particular, the "undoable" functions would undo
changes to variable bindings, property lists, and list structures that may have occurred during the execution of a program. This would allow re-execution under the same initial conditions.

The Lisp Library provides a basis upon which future enhancements to the Lisp environment may be built. The system admits the addition of both new functions and new Libraries. With this system it is hoped that a more flexible Lisp programming environment will be available to Lisp programmers.


1 INTRODUCTION

The Lisp/MTS interpreter was originally implemented at the University of Michigan by Wilcox[5]. As UBC uses the same operating system as Michigan (MTS), the interpreter was easily transferred to UBC. At UBC the interpreter has been altered by P. Friedman and myself. Local changes include the addition of many new function, changes to the names of some functions, and alterations to the printmacro and stack operations.

In its basic form, Lisp/MTS is very similar to the MACLISP system of MIT. All forms are typed in as regular Lisp forms to top level Lisp; there is no evalquote feature. Most of the standard functions of Lisp are available.
This section will attempt to indicate some of the features of Lisp/MTS which differ from other Lisp systems. Many of these differences are due, at least in part, to the way that programs running under MTS can communicate with the operating system and its files.

2. A Stack Operations

Lisp/MTS provides various functions which can access information stored by the Lisp interpreter on its pushdown stack. The function UNEVAL allows the accessing of EVAL forms on the stack. A call to (UNEVAL LOC) will return the form stored at location LOC on the stack. LOC can be an integer n which indicates the n-th form from the top of the stack (a negative integer counts from the bottom of the stack), or the name of a function which is to be searched for. If an optional second argument is present, it specifies what is to be done at that location on the stack. A second argument of T causes execution to resume at that point on the stack. Any other value must be a legitimate Lisp form which will be evaluated instead of the form at the specified location.
The function `RETURN` has been extended to allow an optional second argument of the same form as the LOC argument of `UNEVAL`. This allows values to be returned from any function, not just `PROGS`.

The `DISPLAY` function allows the accessing of variable bindings at specified locations on the stack. This is used in the Debug Package to implement the `VALUE` command.

2. B Error Handling

Lisp/MTS assigns to each possible error (of which there are about 40), an error form which is a Lisp expression to be evaluated when the error occurs. The stack is left intact while the error is being processed, thus allowing the user complete access to it via the stack functions. By default, the error form is the system `DUMP` function which prints backtrace information and error messages.

The Debug Package makes use of error forms by changing them so that they call the Debug Package whenever an error occurs. Since the Debug Package can access the stack via `UNEVAL` and `DISPLAY`, it has complete control over the further execution of the user's program.
2.C Status Function

The Lisp/MTS user accesses most of the system switches and functions via a common STATUS function. Each STATUS subfunction has a unique number through which it can be accessed. There are various I/O related STATUS switches available to set end-of-file functions, line file modifiers and line numbers, file and buffer prefixes etc. Time and core usage information is also available. The current error number and error form used by the Autoloader and Debug Package are accessed by using the STATUS function.

2.D Lisp/MTS I/O Structure

The Lisp/MTS system provides a very flexible and powerful scheme for reading and writing MTS files and communicating with the user's terminal. It is because of this flexibility that many of the functions and programs in the Lisp Library exist.
2. D. 1  MTS File Structure

This section is not intended in any way to be a complete or even technically accurate description of MTS files. The purpose of this description is to indicate how MTS files are used from the Lisp/MTS system.

In MTS most user's programs and data are stored in line files which are disk files composed of lines of from 1 to 256 characters. Each line has associated with it a line number via which it can be accessed. The system I/O routines allow random access to any lines in the file for the purpose of either reading or writing.

The conversational terminals in MTS are treated in the same way as a file; writing on the pseudo-file name *SINK* writes to the user's terminal, reading from *SOURCE* reads the user's typed input.

2. D. 2  Lisp/MTS Buffers

Lisp/MTS accesses the MTS files and the user's terminal via a file-buffer pair called an IOARQ. A buffer is a special atom whose print name is variable in length from 1 to 255 characters. All writing and reading is done into or from these buffers. If no file is attached to the buffer, the user's terminal is used. The prettyprinter described in Chapter III uses a buffer to
determine the length of a particular Lisp structure. In this case, no actual I/O occurs; the contents of the buffer is never printed out.

The IOARGS are accessed and manipulated via a number of STATUS subfunctions as well as by the I/O functions READ, PRINT, PRIN1, etc. All I/O functions allow an optional buffer argument which specifies for that particular call of the function, which file-buffer pair to use. This allows the use of many different input or output streams at once.

2.D.3 I/O Intercepts and Macros

An intercept function in Lisp/MTS is a Lisp form which will be evaluated when a certain condition occurs. The error forms are an example of one type of intercept; when an error occurs, the associated error form is invoked. There also exist in Lisp/MTS a number of intercept forms which are associated with I/O operations. An end-of-file on a file will cause the invocation of an intercept. The attempt to read past the end of a line or print past the end of a line can cause an intercept. These can, of course, be turned off. The end-of-line intercepts are used by the command processors of the Debug Package, Editor, and GENLIB routines.

Another type of intercept function in Lisp/MTS is the

Language Extensions and Differences
printmacro facility. The Lisp user can specify that a certain Lisp atom is to be recognized by the print routines as a printmacro atom. Whenever the printer attempts to print a list whose first element (CAR) is a printmacro atom, the associated intercept function will be called instead. This feature is used by the prettyprinter to allow the form (QUOTE (A B C)) to be printed as '(A B C).

2. E. The Influence of MTS

The operating system and the available conversational terminals have greatly influenced the design and implementation of the Lisp Library system. The demands and requirements of the user are affected by how he normally interacts with other components of the MTS system. The flexibility of the MTS file system means that good facilities for reading and writing these files from Lisp are expected by the user.

The conversational terminals used at UBC are IBM 3270 display terminals. These have a very high data transfer rate. The speed at which these terminals can transfer information to the user means that the Lisp system can be verbose without seriously degrading the speed of interaction with the system. Users are also more willing to use prettyprinters and other
programs which produce many lines of output, since they do not have to wait while the lines are printed out.

One of the most helpful features of the MTS system is its use of prefix characters to prompt the user for input. Examples of this are seen in the examples of Chapter III where the characters *, >, :, and @ are used. By using different prefix symbols for different programs and processes, the user always knows what is expected of him.
(DEFUN LOADFUN (FS) (COND ((SETQ INDEX (SET LIB "INDEX." (LOADINDEX LIB))) LIST (CDR (ASSQ FN INDEX))) (RETURN NIL)) (BUFFER (GET LIB "BUFFER.") LOADSAP (GET LIB "LOADSAP.") TOPCALLS (GET LIB "TOPCALLS.") LIBRARY LIB) (RETURN INDEX)))

(DEFUN BP.ADFROH (BUFFER LINE) (PBOG (STATUS) (APPLY1 'STATUS '(16 BUFFER 2) (LIST 15 'BUFFER (FIX (TIKES 1000 LINE)))) (SETQ STATUS (STATUS (12 BUFFER (LAFLBDA X (RETURN NIL •BEADLINE))) | (BEADLINE BUFFER) (APPLY 1 'STATUS (LIST 12 'BUFFER STATUS) '(16 BUFFER 0)) (RETURN T))))

(SETQ LIBRARYLIST (LIST 'PSIL:LIBRARY))

(DEFUN *PUTOB FEPS (L) (RABC *(LAMBDA (L) (PUTOB (COPY L)) L)))

(DEFUN [ESN] IGN (AND (EQ (STATUS 29) 68) TAULOAD ([LIBRARY-ENR1])) (AND [DEBUG] ([DEBUG-ENR1]) (UNVAL [ESN] *(DEBUG 7)) 1000)

(SETQ TAULOAD [ESN] [TAULOAD T [ESN] [ESN]])

(DEFUN [LIBRARY-ENR1] IGN (PROG (FP #) (COND ((SETQ STATUS (29) 68)) (AND (ASSQ FP [GET LIB "INDEX." [LOADINDEX LIB]]) (RETURN T *(NAPC))))

  (LIBRARYLIST)
  (PRINT "LOADING")
  (PRINT FP#)
  (TEMP FP)
  (LOADFUN FP#)
  (UNVAL FP# T))))

END UP FILE

1 APR 13, 1974 10:39:02
 Appendix III - Library Listing

---

```lisp
; Literal Index

(defun break-f (expr (l))
  (setq break-sw t)
  (lambda (l) (debugbreak l) (restart) l)
)

defun case-p (expr (i)
  (eval (elt (cdr i) (eval (case i))))
)

defun conses (i)
  (cond ((atom i) 0)
        ((and (conses (car i)) (conses (cdr i)))))
)

defun debug-err ()
  (and eq (status 29) 96) (debugskip (status 28)) (step 6) (uneval '(err) '(debug 1 t))
)

defun debug-n (form)
  (setq breakpoint nil)
  (step 3)
  (setq form ("eval" form))
  (step 0)
  (form)
)

defun breakpoint (expr (l)
  (step 0)
  (ask (breaksw ("eval" (cadd3 l))) (setq breakpoint (cadd l)) (step 4))
  (on eval 'breakpoint (cadd l))
)

defun debug-levelset (l)
  (debuglevel (display (eval (list 'debugoutput l)))))
)

defun debug-nexp9 (form)
  (setq breakpoint nil)
  (step 3)
  (setq form ("eval" form))
  (step 0)
  (form)
)

defun debug-points (form)
  (setq breakpoint nil)
  (step 3)
  (setq form ("eval" form))
  (step 0)
  (form)
)
```

---

1: BREAKF
2: CASE
3: CONSES
4: DEBUG-ERR
5: DEBUG-N
6: BREAKPOINT
7: DEBUG-NLEVELSET
8: STATUS

; END OF INDEX

---
(COND ((NOTP 'PRINTTAIM) (STATUS nil) (READTAG NIL))
(READTAG (CONS (CAR READTAG) 'PRINTTAIM))

******DISKIN***************
(DISKIN (FILE))

******DISKOUT***************
(DISKOUT)

******DISKIN1***************
(DISKIN1 (FILE))

******DISKOUT1***************
(DISKOUT1)

******DISKIN2***************
(DISKIN2 (FILE))

******DISKOUT2***************
(DISKOUT2)

******DISKIN3***************
(DISKIN3 (FILE))

******DISKOUT3***************
(DISKOUT3)

******DISKIN4***************
(DISKIN4 (FILE))

******DISKOUT4***************
(DISKOUT4)

******DISKIN5***************
(DISKIN5 (FILE))

******DISKOUT5***************
(DISKOUT5)
(SETQ EDITCHAIN (LAST EDITCHAIN))

(PF (EDITERROR)
  ((EDITERROR 'UP) (EDITERROR 'OK))
  ((EDITERROR 'NX) (EDITERROR 'UP))
  ((EDITERROR 'SOUTH) (EDITERROR 'WEST)))

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)

(EDITCHAIN)
(EDITCHAIN) (CAR EDITCHAIN) (CEAR EDITCHAIN))

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)

(EDITCHAIN) (EDITCHAIN)
(defun fnsize (fexpr (l)
  (defun genlibhexpr (file oldlib)
    (progn (oldisdex ckso fslist done index external term line ehtbt gendue)
      (open (getbuf 255 "source") "...
        (status (10 getbuf t)
          (setq index (list 'index))
          (genlibinit)
          (while t
            (setq chbd (read getbuf))
            (selectq crsd
              (list (cerliddlist))
              (external (getlibext))
              (add (getlibadd))
              ((delete del) (ceblibdel))
              ((delete delfor) (ceblibdelfor))
              (clear ext (setq external bis))
              (addto (getlibaddso))
              (stop (return "stop")
                (if (return (cond ((diskerrset (genlibstop)) 'oone) (tails))))
                (eval crnd))
          (setq external (cons l external))))
    (defun genlibadd ()
      (progn (l bare)
        (while t
          (setq bare (read getbuf) (lambdab x (return nil)))
          l (read getbuf) (lambdab x (return nil)))
          bcosc enlist
          (list (coos hare
            (coosd (if (listp ) l)
              (get l
                (if (getlibassg "no fns specified" (l)) "bconc1"
                  (return (getlibassg "no fns specified" (l)) "bconc2"))))))
      (print (eval crnd))))
    (defun genliblist ()
      (progn (genlib hexpr (fexpr (ili)
        (teepbi)
        (print '* error*)
        (raptc " (lambdab ix1 (cond ((listp »xi) (print (eval (car ix1)))> ((phis! ix1)))) il1)
        (bcosc)))))))
  (defun fnsize (fexpr (l)
    (print t)
    (setq chbd (read getbuf))
    (selectq crsd
      (list (cerliddlist))
      (external (getlibext))
      (add (getlibadd))
      ((delete del) (ceblibdel))
      ((delete delfor) (ceblibdelfor))
      (clear ex (setq external bis))
      (addto (getlibaddso))
      (stop (return "stop")
        (if (return (cond ((diskerrset (genlibstop)) 'oone) (tails))))
        (eval crnd))
    (print (eval crnd))))
  (defun genlibadd ()
    (progn (l bare)
      (while t
        (setq bare (read getbuf) (lambdab x (return nil)))
        l (read getbuf) (lambdab x (return nil)))
        bcosc enlist
        (list (coos hare
          (coosd (if (listp ) l)
            (get l
              (if (getlibassg "no fns specified" (l)) "bconc1"
                (return (getlibassg "no fns specified" (l)) "bconc2"))))))
      (print (eval crnd))))
  (defun genliblist ()
    (genlib hexpr (fexpr (ili)
      (teepbi)
      (print '* error*)
      (raptc " (lambdab ix1 (cond ((listp »xi) (print (eval (car ix1)))> ((phis! ix1)))) il1)
      (bcosc)))))))
  (defun genlibhexpr (file oldlib)
    (progn (oldisdex ckso fslist done index external term line ehtbt gendue)
      (open (getbuf 255 "source") "...
        (status (10 getbuf t)
          (setq index (list 'index))
          (genlibinit)
          (while t
            (setq chbd (read getbuf))
            (selectq crsd
              (list (cerliddlist))
              (external (getlibext))
              (add (getlibadd))
              ((delete del) (ceblibdel))
              ((delete delfor) (ceblibdelfor))
              (clear ex (setq external bis))
              (addto (getlibaddso))
              (stop (return "stop")
                (if (return (cond ((diskerrset (genlibstop)) 'oone) (tails))))
                (eval crnd))
          (setq external (cons l external))))
    (defun genlibadd ()
      (progn (l bare)
        (while t
          (setq bare (read getbuf) (lambdab x (return nil)))
          l (read getbuf) (lambdab x (return nil)))
          bcosc enlist
          (list (coos hare
            (coosd (if (listp ) l)
              (get l
                (if (getlibassg "no fns specified" (l)) "bconc1"
                  (return (getlibassg "no fns specified" (l)) "bconc2"))))))
      (print (eval crnd))))
    (defun genliblist ()
      (genlib hexpr (fexpr (ili)
        (teepbi)
        (print '* error*)
        (raptc " (lambdab ix1 (cond ((listp »xi) (print (eval (car ix1)))> ((phis! ix1)))) il1)
        (bcosc)))))))
  (defun fnsize (fexpr (l)
    (print t)
    (setq chbd (read getbuf))
    (selectq crsd
      (list (cerliddlist))
      (external (getlibext))
      (add (getlibadd))
      ((delete del) (ceblibdel))
      ((delete delfor) (ceblibdelfor))
      (clear ex (setq external bis))
      (addto (getlibaddso))
      (stop (return "stop")
        (if (return (cond ((diskerrset (genlibstop)) 'oone) (tails))))
        (eval crnd))
    (print (eval crnd))))
  (defun genlibadd ()
    (progn (l bare)
      (while t
        (setq bare (read getbuf) (lambdab x (return nil)))
        l (read getbuf) (lambdab x (return nil)))
        bcosc enlist
        (list (coos hare
          (coosd (if (listp ) l)
            (get l
              (if (getlibassg "no fns specified" (l)) "bconc1"
                (return (getlibassg "no fns specified" (l)) "bconc2"))))))
      (print (eval crnd))))
  (defun genliblist ()
    (genlib hexpr (fexpr (ili)
      (teepbi)
      (print '* error*)
      (raptc " (lambdab ix1 (cond ((listp »xi) (print (eval (car ix1)))> ((phis! ix1)))) il1)
      (bcosc)))))))
  (defun fnsize (fexpr (l)
    (print t)
    (setq chbd (read getbuf))
    (selectq crsd
      (list (cerliddlist))
      (external (getlibext))
      (add (getlibadd))
      ((delete del) (ceblibdel))
      ((delete delfor) (ceblibdelfor))
      (clear ex (setq external bis))
      (addto (getlibaddso))
      (stop (return "stop")
        (if (return (cond ((diskerrset (genlibstop)) 'oone) (tails))))
        (eval crnd))
    (print (eval crnd))))
  (defun genlibadd ()
    (progn (l bare)
      (while t
        (setq bare (read getbuf) (lambdab x (return nil)))
        l (read getbuf) (lambdab x (return nil)))
        bcosc enlist
        (list (coos hare
          (coosd (if (listp ) l)
            (get l
              (if (getlibassg "no fns specified" (l)) "bconc1"
                (return (getlibassg "no fns specified" (l)) "bconc2"))))))
      (print (eval crnd))))
  (defun genliblist ()
    (genlib hexpr (fexpr (ili)
      (teepbi)
      (print '* error*)
      (raptc " (lambdab ix1 (cond ((listp »xi) (print (eval (car ix1)))> ((phis! ix1)))) il1)
      (bcosc)))))))
(DEFUN GENLIBINFO ()
  (PROG (TIME  MIN SEC)
    (SETQ TIEE (STATUS 39) NIN (IDIV TIME 60000)  SEC (IDIV (BEMAIN TIME 60000)  1000))
    (PBINTM3SG LISPOUT "ELAPSED TIME SINCE START OF BUFFER »" (MIN)  HINDI'S (SEC)
      SECONDS)
    (PBINTMSSG LISPOUT "CPU TIME = ((ICIV (ADD (STATUS 38) 500) 1000) ) "SECONDS.")
    (PRINKTMSSG LISPOUT "CURREN'T STACK ALLOCATION » ( (DIVIDE (STATUS 17) 0096)) "PAGES.")
    (MTS "END OF INFORMATION")
  )
)

;***LOADINFO********************************************************************
(PRIN1 "^" "^" BUFFER 2))
(DEFUN PPBREAKPOINT (FORM)
  (COND ((EQ (LENGTH FORM) 0) (AND S?(PRIN1 i BUFFER 2)) (PPFORM1 (CADR FORM)))
  (RETURN NIL "PPSPECIAL")))

(DEFUN PPBREAKPOINT (FORM)
  (COND ((EQ (LENGTH FORM) 0) (AND S?(PRIN1 i BUFFER 2)) (PPFORM1 (CADR FORM)))
  (RETURN NIL "PPSPECIAL")))

(DEFUN PRINTFILE (FILE)
  (PROC (SEQ DETPBOPS 'CDB) (SORTLIST (SORTLIST DETPBOPS 'CDB) "CDB) (SORTLIST AS-IS 'CAB))))

(DEFUN SETFILE (LIST N VALUE)
  (RPLACA (UTH LIST N) VALUE))

(DEFUN SETFILE (LIST N VALUE)
  (RPLACA (UTH LIST N) VALUE))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL)))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL])))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL))))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL))))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL))))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL))))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL))))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL))))

(DEFUN UNBREAKF FEXPB (L)
  (COHO (L (BAPC •(LASBDA (L) (PROG NIL (DEBUGUNBB L) IBESTABT)) L))
  ((SETQ BREAKSU NIL))))
(DEFUN XBEFNEWFN (FUN) ...)
(DEFUN XBEFOUOTE (F) ...)
(DEFUN XBEFVAB (V) ...)
(DEFUN XBEFQFORR (F) ...)
(DEFUN XBEFSOBT1 (FUN) ...)
(DEFUN XBEFSOBT (FUN) ...)
(DEFUN XBEFADOFN (FUN TYPE) ...)
(DEFUN XREFFN ...)
(DEFUN XREFADDL ...)
(DEFUN XBEFSUBBS (HOT (RERQ FUN XSEPHOT)) (OR (ATL ...)
(DEFUN IBEPF0BI1 (F PROGSW) ...)
(DEFUN IBEPF0BI2 (F PROGSW) ...)
(PBOG (PLIST) ...)
(AND (LISTP F)...)
(COND ...)
(PBOG (XREFADOL CVA3S (CADB F)) ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...
(SELECTQ FN ...
(PBOG (XREFADOL CVA3S (CADB F)) ...)
```
105
1.0
0.9
1.0.1
1.0.2
(LAMBDA (LET (DEFUN FUN (LET))) (OR (ATOM LET) (DEFUN LET)))
1.1
(CADDR (DEFUN FUN LIBRARY) LAMBDA)
1.2.4
1.3.10
(DEFUN IMPROSELECT (P)
1.4.0.2
(FUNC (DEF)
1.4.1.1
(KREPFORM (CASH F) FEL)
1.4.1.5
(KREP QUERY F (CASH F) KREPFORM F (CASH F))
1.4.1.5
(REPEAT * (FUNC (DEFUN LAMBDA (L) (KREPFORM L NIL)) (CONS (Q? (CASH F)) (CASH F)))))
1.4.1.7
(SUBS (LENGTH F))
1.4.1.8
(KREPFORM (CASH F) NIL))
1.4.1.9
1.4.2.0
1.4.2.1
(DEFUN IMPRELIS ()
1.4.2.2
(IF (DEFUN LAMBDA (LIB) (COPY (CDS (GET LIB "TOPCALLS" ) (RETURN NIL 'COPT)))) LIBRARY)
1.4.2.3
1.4.2.4
1.4.7.1
(*I***FILE*******************************)
1.4.7.2
(DEFUN IMPRELIS FEXPR (FLA)
1.4.7.3
(FUNC (DEF)
1.4.7.4
((DEFUNC (LAMBDA (FLA) (AND (ATOM FEXPR) (SETQ FEXPR (CONS (TAG FEXPR)))))
1.4.7.5
(IF (DEFUN FEXPR ""FEXPR" (DEFUNC (LIST FEXPR) (CDS (CONS (TAG FEXPR) (CDS (CONS (TAG FEXPR) (CDS (CONS (TAG FEXPR) (CDS (CONS (TAG FEXPR)) (CDS (CONS (TAG FEXPR) (CDS (CONS (TAG FEXPR)) (CDS (CONS (TAG FEXPR))))))))))))))
1.4.7.6
1.4.7.7
1.4.7.8
1.4.7.9
1.4.0.1
1. APR. 15, 1974 11:43:55
END OF FILE
```
<table>
<thead>
<tr>
<th>Command</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>?AUTOLOAD</td>
<td>20</td>
</tr>
<tr>
<td>?BREAKSW</td>
<td>44</td>
</tr>
<tr>
<td>?BROKEN</td>
<td>45</td>
</tr>
<tr>
<td>?DEBUG</td>
<td>43</td>
</tr>
<tr>
<td>?DISKMAXLEN</td>
<td>36</td>
</tr>
<tr>
<td>?MAXLEN</td>
<td>37</td>
</tr>
<tr>
<td>LIBRARYLIST#</td>
<td>19,20</td>
</tr>
<tr>
<td>&quot;BUFFER.&quot;</td>
<td>18</td>
</tr>
<tr>
<td>&quot;INDEX.&quot;</td>
<td>18</td>
</tr>
<tr>
<td>&quot;LOADMAP.&quot;</td>
<td>19</td>
</tr>
<tr>
<td>&quot;TOPCALLS.&quot;</td>
<td>19</td>
</tr>
<tr>
<td>Autoloader</td>
<td>20</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>43</td>
</tr>
<tr>
<td>Cross Referencing</td>
<td>62,63</td>
</tr>
<tr>
<td>DEBUG PACKAGE</td>
<td>41</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>43,48,50,58</td>
</tr>
<tr>
<td>Commands</td>
<td>53</td>
</tr>
<tr>
<td>DEBUG Function</td>
<td>50</td>
</tr>
<tr>
<td>Examples</td>
<td>50,60</td>
</tr>
<tr>
<td>Handling Of Lisp Errors</td>
<td>42</td>
</tr>
<tr>
<td>HELP Command</td>
<td>54</td>
</tr>
<tr>
<td>Pseudo-Breakpoints</td>
<td>46</td>
</tr>
<tr>
<td>Stackpointer</td>
<td>47,55,57,58</td>
</tr>
<tr>
<td>Editor</td>
<td>71</td>
</tr>
<tr>
<td>Function Analysis</td>
<td>62</td>
</tr>
<tr>
<td>FILE I/O</td>
<td>30</td>
</tr>
<tr>
<td>Comments</td>
<td>32</td>
</tr>
<tr>
<td>DISKIN</td>
<td>33</td>
</tr>
<tr>
<td>DISKOUT</td>
<td>34</td>
</tr>
<tr>
<td>FNS List</td>
<td>22,31,67,75</td>
</tr>
<tr>
<td>Prettyprinter</td>
<td>36</td>
</tr>
<tr>
<td>UPDATE</td>
<td>35</td>
</tr>
<tr>
<td>FNS List</td>
<td>31</td>
</tr>
<tr>
<td>FUNCTIONS</td>
<td>12</td>
</tr>
<tr>
<td>*PUTOB</td>
<td>69</td>
</tr>
<tr>
<td>*REMOB</td>
<td>70</td>
</tr>
<tr>
<td>BREAKF</td>
<td>48</td>
</tr>
<tr>
<td>CASE</td>
<td>70</td>
</tr>
<tr>
<td>CONSES</td>
<td>71</td>
</tr>
<tr>
<td>DEBUG</td>
<td>50</td>
</tr>
<tr>
<td>DISKIN</td>
<td>33</td>
</tr>
<tr>
<td>DISKOUT</td>
<td>34</td>
</tr>
<tr>
<td>DREVERSE</td>
<td>71</td>
</tr>
<tr>
<td>EDITE</td>
<td>71</td>
</tr>
<tr>
<td>EDITF</td>
<td>72</td>
</tr>
<tr>
<td>ELEM</td>
<td>72</td>
</tr>
<tr>
<td>EXTRACTFN</td>
<td>22</td>
</tr>
<tr>
<td>FNMAP</td>
<td>62</td>
</tr>
<tr>
<td>FNSIZE</td>
<td>63</td>
</tr>
</tbody>
</table>