MACLISP REFERENCE MANUAL

by

David A. Moon

Project MAC - M.I.T.
Cambridge, Massachusetts

REVISION Ø

April 1974

Revision 0
04/08/74

MACLISP REFERENCE MANUAL

by David A. Moon

c Copyright 1973, 1974, Massachusetts Institute of Technology
All rights reserved

Acknowledgements

Ira Goldstein, David P. Reed, Guy L. Steele, Alex Sunguroff, and Jon L. White contributed advice and wrote sections of this document. The runoff text preparation system on Multics helped with the clerical work. The document was printed on the MIT A.I. Lab's Xerox Graphic Printer.

Errors in this Manual

This is an initial version of the LISP Manual and all readers' comments are solicited. Please point out any errors, inaccuracies, inconsistencies, obscure points, etc. that you may find.

The following communication paths may be used to communicate with the authors:

Multics mail to Moon.AutoProg

ITS mail to MOON on the MathLab machine

ARPA Network mail to MOON @ MIT-ML
(host 198. decimal, 306 octal)

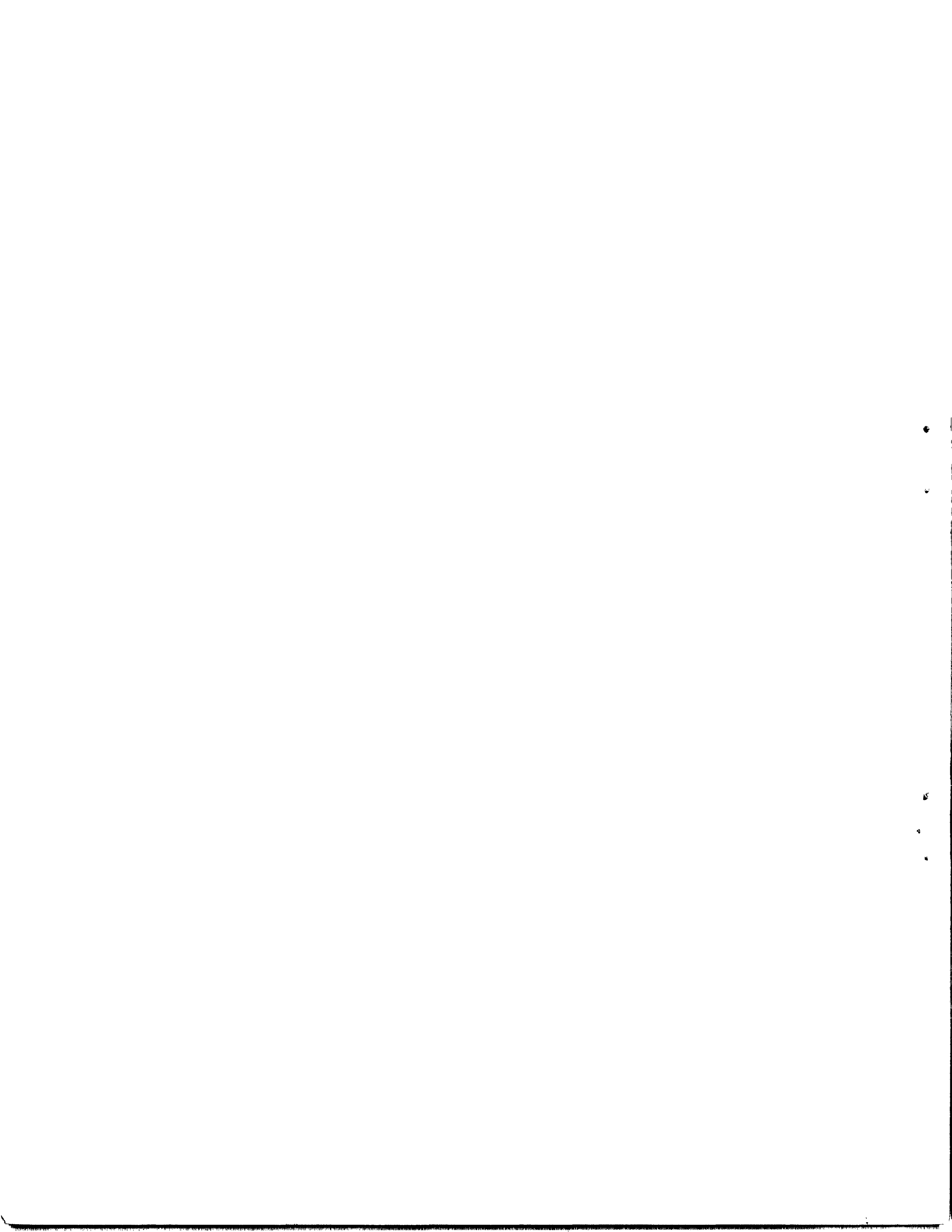
U.S. Mail to D. A. Moon, Rm. 505
545 Technology Square
Cambridge, Mass. 02139

Table of Contents

1.	General Information	1
1.1	The MACLISP Language	1
1.2	Structure of the Manual	1
1.3	Notational Conventions	2
2.	Data	5
2.1	Data Types	5
2.2	Predicates for Checking Types	9
3.	The Basic Actions of LISP	11
3.1	Binding	11
3.2	Evaluation	12
3.3	Application	13
3.4	A-list Pointers	16
3.5	Functions to Perform These Actions	17
4.	Functions for Manipulating List Structure	23
4.1	Examining Existing List Structure	23
4.2	Creating New List Structure	29
4.3	Modifying Existing List Structure	32
5.	Flow of Control	35
5.1	Conditionals	36
5.2	Iteration	38
5.3	Nonlocal Exits	43
5.4	Causing and Controlling Errors	44
6.	Manipulating the Constituents of Atomic Symbols	47
6.1	The Value Cell	47
6.2	The Property List	49
6.3	The Print-Name	52
6.4	Miscellaneous Functions	53
6.5	Defining Atomic Symbols as Functions	54
7.	Functions on Numbers	57
7.1	Number Predicates	57
7.2	Comparison	59
7.3	Conversion	61
7.4	Arithmetic	63
7.5	Exponentiation and Log Functions	69
7.6	Trigonometric Functions	70
7.7	Random Functions	71
7.8	Logical Operations on Numbers	72
8.	Character Manipulation	73
8.1	Character Objects	73
8.2	Functions on Strings	76
9.	Functions Concerning Arrays	79
10.	"Mapping" Functions	83

11.	Sorting Functions	87
12.	Functions for Controlling the Interpreter	89
12.1	The Top Level Function	89
12.2	Break Points	91
12.3	Control Characters	92
12.3.1	List of Control Characters	95
12.3.2	Control-Character Functions	97
12.4	Errors and User Interrupts	98
12.4.1	The LISP Error System	98
12.4.2	User Interrupts	100
12.4.3	User Interrupt Functions and Variables	104
12.4.4	Autoload	107
12.5	Debugging	108
12.5.1	Binding, Pdl Pointers, and the Evaluator	108
12.5.2	Functions for Debugging	109
12.5.3	An Example of Debugging in Maclisp	112
12.6	Storage Management	113
12.6.1	Garbage Collection	113
12.6.2	Spaces	114
12.6.3	Storage Control Functions	116
12.6.4	Dynamic Space and Pdl Expansion	117
12.6.5	Initial Allocation	118
12.7	The Functions status and sstatus	120
12.8	Miscellaneous Functions	127
12.8.1	Time	128
12.8.2	Getting into LISP	129
12.8.3	Getting Out of LISP	131
12.8.4	Sending Commands to the Operating System	134
13.	Input and Output	135
13.1	Basic I/O	135
13.2	Files	137
13.2.1	Naming Files	139
13.2.2	Opening and Closing	142
13.2.3	Specifying the Source or Destination for I/O	143
13.2.4	Handling End of File	148
13.3	Applying Defaults to File Names	150
13.4	Requests to the Operating System	152
13.4.1	Manipulating the Terminal	152
13.4.2	File System Operations	153
13.4.3	Random Access to Files	154
13.5	The Old "Uread" I/O System	155
13.6	Advanced Use of the Reader	158
13.6.1	The Obarray	158
13.6.2	The Readtable	159
13.7	Control of Printer Formatting	164
13.8	Input Format Expected by (read)	168
13.9	"Moby I/O"	170
14.	Compilation	177
14.1	Peculiarities of the Compiler	178
14.1.1	Variables	178

14.1.2	In-line Coding	179
14.1.3	Function Calling	180
14.1.4	Input to the Compiler	181
14.1.5	Functions Connected with the Compiler	182
14.2	Declarations	184
14.3	Running Compiled Functions	188
14.4	Running the Compiler	190
14.5	LAP on the pdp-10	194
14.5.1	The LAP Function	194
14.5.2	Valid LAP Code Forms	196
14.5.3	LAP Syllables	198
14.5.4	Functions and Variables Used by lap and faslap	201
14.5.5	Differences Between lap and faslap	206
14.5.6	Conventions for Functions in Lisp	207
14.5.7	Internal Routines for use by LAP Code	211
14.5.8	Routines For Use by Hand-Coded LAP	217
14.6	Internal Details of the Multics Implementation	221
15.	The Trace Facility	223
16.	Formatted Printing of LISP Data	227
16.1	Introduction	227
16.2	Top Level Functions	228
16.2.1	grind and grind0 - fexprs	228
16.2.2	grindef - fexpr	228
16.2.3	Formatting	229
16.2.4	remgrind - fexpr	230
16.2.5	Functions, Atoms, and Properties Used by Grind	230
16.3	Predefined Formats	231
16.3.1	Standard Formats	231
16.3.2	Special Grindfns	232
16.3.3	Inverting Read Macros	234
16.3.4	System Packages	234
16.4	Comments	235
16.4.1	Single Semicolons	235
16.4.2	Double Semicolons	235
16.4.3	Triple Semicolons	236
16.5	Grind Control	237
16.5.1	Defining New Formats	238
16.5.2	Vocabulary	238
16.5.3	Examples	239
16.5.4	Grindmacros	241
17.	The LISP "Indexer"	243
18.	The LISP Editor	245
Appendix A.	Glossary	249
Appendix B.	Index of Functions	268
Appendix C.	Index of Atomic Symbols	274
Appendix D.	Concept Index	275



1 - General Information

1.1 - The MACLISP Language

MACLISP is a dialect of LISP developed at M.I.T.'s Project MAC for use in artificial intelligence research and related fields. MACLISP is descended from the commonly-known LISP 1.5 dialect, however many features of the language have been changed or added.

This document is intended both as a reference source for the language and as a user's guide to three implementations. These are, in chronological order, the M.I.T. Artificial Intelligence Lab's implementation on the DEC pdp-10 computer under their operating system ITS, hereafter referred to as "the ITS implementation," Project MAC's implementation on Honeywell's version of the Multics system, hereafter referred to as "the Multics implementation," and the version that runs on the DEC pdp-10 under DEC's TOPS-10 operating system, hereafter called "the DEC-10 implementation." The DEC-10 implementation also runs under TENEX by means of a TOPS-10 emulator. Since the ITS and DEC-10 implementations are closely related, they are sometimes referred to collectively as the pdp-10 implementation.

These implementations are mostly compatible; however, some implementations have extra features designed to exploit peculiar features of the system on which they run, and some implementations are temporarily missing some features. Most programs will work on any implementation, although it is possible to write machine-dependent code if you try hard enough.

1.2 - Structure of the Manual

This manual is not specifically designed for LISP users of any particular level of ability, though it does make assumptions as to what an "average" user of LISP will require of a manual. Since it is intended for a general class of users it must satisfy some constraints of design in order to be of benefit to a user of some particular ability. The manual must provide as much information as a user might need, yet provide it in a manner such that a less experienced user might still have access to some of that information.

The general structuring of the manual is by meaning; things of similar meaning will be found grouped together. Explanatory text and function definitions are interspersed. Usually text will be found at the beginnings of chapters, sections, or subsections with definitions following.

Complexity tends to increase across each subdivision of the manual while usefulness to a new user tends to decrease. The chapters in the beginning of the manual are of more use to a new user, the later ones contain either more complex or less useful information. If a chapter is undivided then it will become more complex toward the end. If a chapter is subdivided, then each section or subsection again follows the same criteria.

Accessing information in the manual is dependant on both the user's level of ability and the purpose for which she or he is using the manual. Though cover to cover reading is not recommended (though not excluded), it is suggested that someone who has never previously seen this manual browse through it, touching the beginning of each subdivision that is listed in the Table of Contents, in order to familiarize himself or herself with the material that it contains. To find an answer to some particular question, one must use one of the provided access methods. Since the manual is structured by meaning one can use the Table of Contents that is found at the beginning of the manual, to find where information of a general class will be found. Entry into the manual by meaning is also facilitated by the Glossary and the Concept Index which are found at the end. Also at the end of the manual is a Function Index which is probably most useful to a regular and repeated user of the dialect, or to an experienced user of another dialect, who wishes to find out the answer to a question about a specific function. However since the manual is structured by meaning, it is useful to examine other functions in the vicinity of the first, for usefulness.

It is again suggested that a new reader of this manual should familiarize himself or herself with its contents.

1.3 - Notational Conventions

There are some conventions of notation that must be mentioned at this time, due to their being used in examples.

A combination of the characters, equal sign and greater then symbol, "=>" , will be used in examples of LISP code to mean evaluation (that is the application of the function, eval.).

All uses of the phrase, "LISP reader," unless further qualified, refer to that part of the LISP system which does input, and not to some person.

The two characters, accent acute, " ' ", and semi-colon, " ; ", are examples of what are called *macro characters*. Though the macro character facility, which is defined in the chapter on Input/Output, is not of immediate interest to a new user of the dialect, these two come preset by the LISP system and are useful. When the LISP reader encounters an accent accute, it reads in the next S-expression and encloses it in a call to the function, quote. That is:

```
'some-atom
```

turns into:

```
(quote some-atom)
```

and

```
'(cons 'a 'b)
```

turns into

```
(quote (cons (quote a) (quote b)))
```

The semi-colon is used as a commenting feature. When the LISP reader encounters it, it discards the rest of that line of input.

All LISP examples in this manual are written according to the conventions of the Multics implementation, which uses both upper and lower case letters and spells the names of most system functions in lower case. Some implementations of MACLISP only use upper case letters because they run on systems which are not equipped with terminals capable of generating and displaying the full ascii character set. However, these systems will accept input in lower case and translate it to upper case.

This page intentionally left blank.

2 - Data

2.1 - Data Types

LISP works with pieces of data called "objects" or "S-expressions." These can be simple "atomic" objects or complex objects compounded out of other objects. Functions, the basic units of a LISP program, are also objects and may be manipulated as data.

Objects come in several types. All types are self-evident, that is it is possible for the system to tell what type an object is just by looking at it, so it is not necessary to declare the types of variables as in some other languages. It should be noted that LISP represents objects as pointers, so that any object will fit in the same cell, and the same object may have several different usages -- for example the same identical object may be a component of two different compound objects.

The data-types are divided into three broad classes: the atomic types, the non-atomic types, and the composite types. Objects are divided into the same three classes according to their type. Atomic objects are basic units which cannot be broken down by ordinary chemical means, while non-atomic objects are structures constructed out of other objects, and composite objects are indivisible entities which have subcomponents which may be extracted and modified but not removed.

The atomic data types are numbers, atomic symbols, strings, and subr-objects.

In LISP numbers can be represented by three types of atomic objects: fixnums, flonums, and bignums. A fixnum is a fixed-point binary integer whose range of values is limited by the size of a machine word. A flonum is a floating-point number whose precision and range of values are machine-dependent. A bignum is an arbitrary-precision integer. It is impossible to get "overflow" in bignum arithmetic, as any finite integer can be represented as a bignum. However, fixnum and flonum arithmetic is faster than bignum arithmetic and bignums require more memory. Sometimes the word "fixnum" is used to include both fixnums and bignums; in this manual, however, the word "fixnum" will never include bignums unless that is explicitly stated.

The external representations for numbers are as follows: a fixnum is represented as a sequence of digits in a specified base, usually octal. A trailing decimal point indicates a decimal base. A flonum is represented as a set of digits containing an embedded or leading decimal point and/or a trailing exponent. The exponent is introduced by an upper or lower case "e". A bignum looks like a fixnum except that it has enough digits that it will not fit within the range available to fixnums. Any number may be preceded by a + or - sign. Some examples of fixnums are 4, -1232, -191., +46. An example of a bignum is 1565656565656565656565656565656565. Some

examples of flonums are: 4.0, .01, -6e5, 4.2e-1.

Another LISP data type is the string. This is a sequence of 0 or more characters. Strings are used to hold messages to be typed out and to manipulate text when the structure of the text is not appropriate for the use of "list processing." The external representation of a string is a sequence of characters enclosed in double-quotes, e.g. "foo". If a " is to be included in the string, it is written twice, e.g. "foo""bar" is foo"bar.

One of the most important LISP data types is the atomic symbol. In fact, the word "atom" is often used to mean just atomic symbols, and not the other atomic types. An atomic symbol has a name, a value, and possibly a list of "properties". The name is used to refer to the symbol in input and output. The external representation of an atomic symbol is just its name. This name is often called the "pname," or "print-name," as it is a sequence of characters that are printed out. For example, an atomic symbol with a pname of foo would be represented externally as foo; internally as a structure containing the value, the pname "foo", and the properties.

There are two special atomic symbols, t and nil. These always have themselves as values and their values may not be changed. nil is used as a "marker" is many contexts; it is essential to the maintenance of data structures such as lists. t is usually used when an antithesis to nil is required for some purpose.

The value of an atomic symbol is any object of any type. There are functions to set and get the value of a symbol. Because atomic symbols have values associated with them, they can be used as variables in programs and as "dummy arguments" in functions. It is also possible for an atomic symbol to have no value, in which case it is said to be "undefined" or "unbound."

The property list of an atomic symbol will be explained in section 6.2. It is used for such things as recording the fact that a particular atomic symbol is the name of a function.

An atomic symbol with less than two characters in its pname is often called a "character object" and used to represent an ascii character. The atomic symbol with a zero-length pname represents the ascii null character, and the symbols with 1-character pnames represent the character which is their pname. Functions which take character objects as input usually also accept a string one character long or a fixnum equal to the ascii-code value for the character. Character objects are always interned on the obarray (see section 6.3).

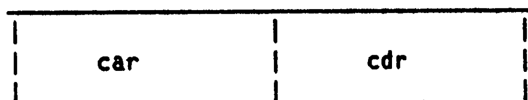
A "subr-object" is a special atomic data-type whose use is normally hidden in the implementation. A subr-object represents executable machine code. The functions built into the LISP system are subr-objects, as are user functions that have been compiled. A subr-object cannot be named directly, so each system function has an atomic symbol which serves as its name. The symbol has the subr-object as a "property."

One composite data type is the array. An array consists of a number of

cells, each of which may contain any LISP object. The cells of an array are accessed by subscripting. An array may have one or more dimensions; the upper limit on the number of dimensions is implementation-defined. An array is always associated with an atomic symbol which is its name. This atomic symbol has on its property list a property with the indicator array and a value, called an array-object or sometimes a "special array cell," which permits the implementation to access the array. See chapter 9 for an explanation of how to create, use, and delete arrays.

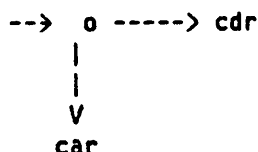
Another composite data type is the file-object, which is described in chapter 13.

The non-atomic data type is the "cons." A cons is a structure containing two components, called the "car" and the "cdr" for historical reasons. These two components may be any LISP object, even another cons (in fact, they could even be the same cons). In this way complex structures can be built up out of simple conses. Internally a cons is represented in a form similar to:



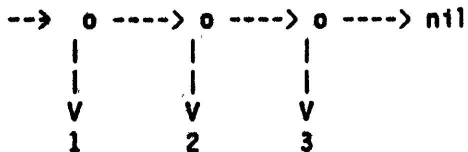
where the boxes represent cells of memory large enough to hold a pointer, and "car" and "cdr" are two pointers to objects. The external representation of a cons is the "dotted-pair" notation
(A . B) where A is the car and B is the cdr.

Another way to write the internal representation of a cons, which is more convenient for large structures, is:



There are three LISP functions associated with conses. The function cons combines its two arguments into a cons; (1 . 2) can be generated by (cons 1 2). The function car returns the car of its argument, and the function cdr returns the cdr of its argument.

One type of structure, built out of conses, that is used quite often, is the "list." A list is a row of objects of arbitrary length. A list of 3 things 1, 2, and 3 is constructed as (cons 1 (cons 2 (cons 3 nil))); nil is a special atom that is used to mark the end of a list. The structure of a list can be diagrammed as:



From this it can be seen that the car of a list is its first element, that the cdr of a list is the list of its elements other than the first, and that the list of no elements is the same as `nil`.

This list of 1, 2, and 3 could be represented in the dot-notation used for conses as `(1 . (2 . (3 . nil)))`, however a more convenient notation for the external representation of lists has been defined: the "list-notation" `(1 2 3)`. It is also possible to have a hybrid of the two notations which is used for structures which are almost a list except that they end in an atom other than `nil`. For example, `(A . (B . (3 . D)))` can be represented as `(A B C . D)`.

A list not containing any elements is perfectly legal and frequently used. This zero-length list is the atom `nil`. It may be typed in as either `nil` or `()`.

2.2 - Predicates for Checking Types

A predicate is a function which tests for some condition involving its argument and returns t if that condition is true and nil if it is not true. These predicates return t if their argument is of the type specified by the name of the function, nil if it is of some other type. Note that the name of most predicates ends in the letter p, by convention.

atom SUBR 1 arg

The atom predicate is nil if its argument is a dotted-pair or a list, and t if it is any kind of atomic object such as a number, a character string, or an atomic symbol.

fixp SUBR 1 arg

The fixp predicate returns t if its argument is a fixnum or a bignum, otherwise nil.

floatp SUBR 1 arg

The floatp predicate returns t if its argument is a flonum, nil if it is not.

numberp SUBR 1 arg

The numberp predicate returns t if its argument is a number, nil if it is not.

typep SUBR 1 arg

typep is a general type-predicate. It returns an atomic symbol describing the type of its argument, chosen from the list

```
(fixnum flonum bignum list symbol string random)
```

symbol means atomic symbol. random is for all types that don't fit in any other category. Thus numberp could have been defined by:

```
(defun numberp (x)
   (and (memq (typep x) '(fixnum flonum bignum))
        t))
```

--- These two functions only exist in the Multics implementation. ---

stringp SUBR 1 arg

The `stringp` predicate returns `t` if its argument is a string, otherwise `nil`.

subrp SUBR 1 arg

The `subrp` predicate returns `t` if its argument is a "subr" object, i.e. a pointer to the machine code for a compiled or system function. Example:

```
(subrp (get 'car 'subr)) => t
```

3 - The Basic Actions of LISP

NOTE: This chapter is obsolete, inaccurate, and misleading. It will be re-written in the next revision of this document.

3.1 - Binding

In LISP "binding" occurs when a variable is needed which is temporary and local to a particular form, for example a temporary variable in a prog or a do, or a lambda variable (what some other languages call a "parameter" or a "dummy variable.") All variables in (interpreted) LISP are implemented as the value cells of atomic symbols; binding an atomic symbol to a local value consists of saving the old value, which is a global or at least less local variable, and then giving the variable its new, local value. The saving of the old value is done in such a way that it will automatically be restored when control leaves the form in question, whether normally or through an error or a throw.

An unusual feature of the binding of LISP variables is that while the form in which the variable is bound is being evaluated, even if it has called upon some function and control is nested deep within it, that variable maintains its local value, rather than its global value. This can be both useful and a source of problem

3.2 - Evaluation

Evaluation is a transformation which takes one object, called a form, and produces another, called the value of that form. Evaluation is used internally by LISP in processing typed input and in (recursively) evaluating portions of a form in the process of evaluating that form. Evaluation is available for explicit use as the function eval.

Evaluation is performed by the LISP interpreter, following the rules set forth below.

Numbers and strings always evaluate to themselves.

Atomic symbols evaluate to the 'value' associated with them. It is possible for an atomic symbol to have no value, in which case the process of evaluation encounters an error, which is handled as described in section 12.4. The special atomic symbols t and nil always have themselves as values, consequently they evaluate like numbers rather than like atomic symbols.

Random objects, such as subr-objects, files, and array-objects evaluate to something random; often themselves. Note that an array-object is different from an atomic symbol which is the name of an array; such an atomic symbol is evaluated the same as any other atomic symbol.

The evaluation of non-atomic forms is more complex. The evaluator regards a non-atomic form as a list, whose first element (car) is a function and whose remaining elements (cdr) are arguments. The value of the form is the result of the function when applied to those arguments, according to the "application" procedure described below.

3.3 - Application

"Application" is the procedure by which a function is invoked with specified arguments to produce a value (and possibly side effects.)

The first step in application is an examination of the function. If it is atomic, then it is required to be an atomic symbol. The atomic symbol's property list is searched for one of the following properties, called "functional properties":

expr, fexpr, macro, subr, fsubr, lsubr, array, autoload

If none of these is found, then the atomic symbol is evaluated and its value is taken to be the function being called and application is restarted at the beginning.

If an array property is found, there is a subscripted reference to the array. The arguments are evaluated from left to right, and used as subscripts to the array. Consequently they must be fixnums and there must be the same number of them as there are dimensions in the array and they must lie within the bounds of the array. The result is the contents of the array cell specified by the subscripts.

If a subr, lsubr, or fsubr property is found, the value of the property must be a subr-object. The subr-object represents a machine-code subroutine, which may be a function built in to the interpreter, a compiled LISP function, or a function written in some other language made known to LISP by some facility such as defsubr or lap, depending on the implementation. The evaluator calls this subroutine, giving it the specified arguments, and the result is the LISP object returned by the subroutine. If the subr-object was in an fsubr property, the subroutine is called with one argument which is the cdr of the list being evaluated. Thus the arguments are not evaluated. If it was a subr or lsubr property, the arguments are evaluated from left to right before they are passed to the subroutine. A subr requires a certain fixed number of arguments, but an lsubr can take a variable number of arguments, between two bounds which depend on the particular lsubr. (See the args function).

If a fexpr property is found, the value of the property must be a list whose car is the atom lambda and whose cadr is a list of one or two atomic symbols, referred to as lambda-variables. The cddr is a list of zero or more objects, referred to as the body. The first lambda-variable is bound to the cdr of the form being evaluated, i.e. to the list of unevaluated arguments. Of course, the body of the fexpr may evaluate the arguments itself by explicitly using the function eval. The second lambda-variable, if present, is bound to an "a-list pointer" which represents the binding state which existed just before the fexpr's lambda-variables were bound. After the lambda-variables have been bound, the body of the fexpr is evaluated from left to right. The result is the value of the last form evaluated.

If an expr property is found, there are three cases: 1) If the value of the property is an atomic symbol, that atomic symbol is taken as the

function and application is restarted from the beginning. 2) If the value of the property is a list whose car is the atom lambda and whose cadr is a list of zero or more atomic symbols, called lambda-variables, then the evaluation is the invocation of an interpreted function called an 'expr.' There must be the same number of arguments as lambda-variables. The arguments are evaluated from left to right and then the lambda-variables are bound to the values of the arguments. Next, the body of the expr is evaluated in the same way as for a fexpr. 3) If the value of the property is a list whose car is the atom lambda and whose cadr is an atomic symbol other than nil, the evaluation is an invocation of an interpreted function called a 'lexpr,' which is the interpreted version of lsubr. The arguments are evaluated from left to right and saved in a place where the functions arg and setarg can find them. Then the single lambda-variable is bound to the number of arguments, and the body is evaluated in the same way as for an expr.

If a macro property is found, its value must look like the value of a fexpr property. The difference between a macro and a fexpr is twofold. One, the first lambda-variable is bound to the whole form, instead of just the cdr of the form (the list of arguments). Two, the object returned by the macro is treated as a form and re-evaluated and its value becomes the result of the application.

If an autoload property is found, the definition of the function is loaded in from an external file and then used. See section 12.4.4 for details.

If the function is not an atom, it is what is sometimes called a "functional form." Note that actually a functional form is anything that can be applied, including atoms. One kind of functional form is a list whose car is the atomic symbol lambda. This is often called a "lambda" expression. It is applied exactly the same way as if the function had been an atomic symbol which had an expr property whose value was the lambda-expression.

Another kind of functional form is a list of the atom label, an atomic symbol, and a functional form. The atomic symbol is bound to the functional form, and then the functional form is applied to the arguments. This is used for strange things like recursive lambda-expressions. Generally a permanently-defined function is better than a label.

The third kind of functional form is a "funarg," which can be produced by the *function function. The funarg contains an a-list pointer. The binding context is temporarily set to the previous context indicated by this a-list pointer, and then the functional form contained in the funarg is applied.

If the functional form does not fall into any of the above cases, it is evaluated and the value is then used as the function being applied and application starts over again at the beginning.

In addition to the variety of application which has just been described,

which is used internally by the evaluation procedure, there is a similar but not identical application procedure available through the function apply. The main difference is that the function and the arguments are passed to apply seperately. They are not consed up into a form. Consequently macros are not accepted by this version of application. In addition, the arguments to exprs, lexprs, subrs, lsubrs, arrays, etc. are not evaluated, since the caller of apply is presumed to have prepared the arguments.

3.4 - A-list Pointers

There is a special type of object called an *"a-list pointer"* which can be used to refer to a binding context. Due to the stack implementation of MACLISP, an a-list pointer is only valid while control is nested within the binding context it names. It is not possible to exit from within a binding context but keep it around by keeping an a-list pointer to it.

An a-list pointer is either a negative fixnum or `nil`. `nil` means the "global" or "top level" binding context. The negative fixnum is a special value of implementation dependent meaning which should be obtained only from one of the three following sources: the function `evalframe`, the function `errframe`, or the second lambda-variable of a fexpr.

The only use for a-list pointers is to pass them to the functions `eval` and `apply` to specify the binding context in which variables are to be evaluated during that evaluation or application. A-list pointers are also used internally by `*function`. When it generates a funarg, it puts in the funarg the functional form it was given and an a-list pointer designating the binding environment current at the time *function was called.

3.5 - Functions to Perform These Actions

eval LSUBR 1 or 2 args

(eval *x*) evaluates *x*, just as if it had been typed in at top level, and returns the result. Note that since eval is an lsubr, its argument actually will be evaluated twice.

(eval *x y*) evaluates *x* in the binding context specified by the a-list pointer *y*.

apply LSUBR 2 or 3 args

(apply *f y*) applies the function *f* to the list of arguments *y*. Unless *f* is an fsubr or fexpr such as cond or and which evaluates its arguments in a funny way, the arguments in the list *y* are used without being evaluated.

Examples:

```
    (setq f '+) (apply f '(1 2 3)) => 6
    (setq f '-) (apply f '(1 2 3)) => -4
    (apply 'cons '((+ 2 3) 4)) =>
        ((+ 2 3) . 4) not (5 . 4)
```

(apply *f y p*) works like apply with 2 arguments except that the application is done in the binding environment specified by the "a-list" pointer *p*.

quote FSUBR

quote returns its unevaluated argument. quote is used to include constants in a form. For convenience, the read function normally converts any S-expression preceded by the apostrophe (acute accent) character into the form (quote <S-expression>). For example, the form:

```
(setq x '(some list))
```

is converted by the reader to:

```
(setq x (quote (some list)))
```

which causes the variable x to be set to the constant list value shown upon evaluation. For more information on input syntax, see the detailed discussion in chapter 13.

function FSUBR

function is like quote except that its argument is a function. It is used when passing a functional argument.
Example:

```
(f00 x (function (lambda (p q)
                    (cond ((numberp q) p)
                          ((numberp p) q)
                    ;or any other random function in here
                          (t (cons p q)) )))
        y)
```

calls f00 with 3 arguments, the second of which is the function defined by the lambda-expression.

Note: quote and function are completely equivalent in the interpreter. The compiler sometimes needs function to tell it that a lambda-expression is a function to be compiled rather than a constant.

function makes no attempt to solve the "funarg problem." *function should be used for this purpose.

*function FSUBR

The value of (*function f) is a "funarg" of the function f. A funarg can be used like a function. It has the additional property that it contains an a-list pointer so that the values of variables are bound the same during the application of the funarg as at the time it was created, provided that the binding environment in which the funarg was created still exists on the stack. Hence if foo is a function that accepts a functional argument, such as

```
(defun foo (f)
    (append one-value (f the-other-value) ))
```

then

```
(foo (*function bar))
```

works, but

```
(foo (prog (x y z)
          (do something)
          (return (*function bar)) ))
```

does not if bar intends to reference the prog variables x, y, and z. *function is intended to help solve the "funarg problem," however it only works in some easy cases. In particular, two general cases of the funarg problem are not solved. funarg's generated by *function are intended for use as functional *arguments*, and cannot be returned

as values of functional applications. Also, due to the implementation, which essentially generates a copy of the binding context at the time *function is applied, assignments to variables in this copied binding context do not affect the values of those variables in the binding context which exists at the time *function is applied. Thus, the user should be careful in his use of *function to make sure that his use does not exceed the limitations of the funarg mechanism.

A funarg has the form

(funarg <function> . <pdl-ptr>)

comment FSUBR

comment ignores its arguments and returns the atomic symbol comment.
Example:

```
(defun foo (x)
    (cond ((null x) 0)
          (t (comment x has something in it)
             (1+ (foo (cdr x))))))
```

Usually it is preferable to comment code using the semicolon-macro feature of the standard input syntax. This allows the user to add comments to his code which are ignored by the input package.
Example:

```
(defun foo (x)
    (cond ((null x) 0)
          (t (1+ (foo (cdr x))))        ;x has something in it
          ))
```

prog2 LSUBR 2 or more args

prog2 evaluates its arguments from left to right, like any lsubr, and returns the value of its second argument.
Examples:

```
    (prog2 (do-this) (do-that))         ;get 2 things evaluated

    (setq x (prog2 nil y
                   (setq y x)))         ;parallel assignment.
                                        ;exchanges x and y.
```

progn LSUBR 1 or more args

progn essentially evaluates all of its arguments and returns the value of the last one. Although lambda-expressions, prog-forms, do-forms, cond-forms, and iog-forms all use progn implicitly, there are occasions upon which one needs to evaluate a number of forms for side-effects when not in these forms. progn serves this purpose. Example:

```
(progn (setq a 1) (cons a '(stuff))) => (1 stuff)
```

arg SUBR 1 arg

(arg nil), when evaluated inside a lexpr, gives the number of arguments supplied to that lexpr.
(arg *i*), when evaluated inside a lexpr, gives the value of the *i*'th argument to the lexpr. *i* must be a fixnum in this case. It is an error if *i* is less than 1 or greater than the number of arguments supplied to the lexpr.
Example:

```
(defun foo nargs              ; define a lexpr foo.
    (print (arg nil))         ; number of args supplied.
    (print (arg 2))           ; print the second argument.
    (+ (arg 1) (arg 3)))      ; return the sum of the
                              ; 1st and 3rd args.
```

setarg SUBR 2 args

setarg is used only inside a lexpr. (setarg *i* *x*) sets the lexpr's *i*'th argument to *x*. *i* must be greater than zero and not greater than the number of arguments passed to the lexpr. After (setarg *i* *x*) has been done, (arg *i*) will return *x*.

listify SUBR 1 arg

listify is a function which efficiently manufactures a list of *n* of the arguments of a lexpr. With a positive argument *n*, it returns a list of the first *n* arguments of the lexpr. With a negative argment *n*, it returns a list of the last (abs *n*) arguments of the lexpr. Basically, it works as if defined as follows:

```
(defun listify (n)
     (cond ((minusp n)
            (*listify (arg nil) (+ (arg nil) n 1)))
           (t
            (*listify n 1)) ))

(defun *listify (n m)        ; auxiliary function.
     (do ((i n (1- i))
          (l nil (cons (arg i) l)))
         ((< i m) l) ))
```

funcall LSUBR 1 or more args

(funcall *f* *a*1 *a*2 ... *an*) calls the function *f* with the arguments *a*1, *a*2, ..., *an*. It is similar to apply except that the separate arguments are given to funcall, rather than a list of arguments. If *f* is an expr, a lexpr, a subr, or an lsubr, the arguments are not re-evaluated. If *f* is a fexpr or an fsubr there must be exactly one argument. *f* may not be a macro.

Example:

```
(setq cons 'plus)
(cons 1 2) => (1 . 2)
(funcall cons 1 2) => 3
```

This page intentionally left blank.

4 - Functions for Manipulating List Structure

4.1 - Examining Existing List Structure

car SUBR 1 arg

Takes the first part of a cons.
Examples:

```
(car '(a b)) => a
(car '(1 . 2)) => 1
```

cdr SUBR 1 arg

Takes the second part of a cons.
Example: (cdr '(a b c)) => (b c)
Note: the cdr of an atomic symbol is its property list.

c...r SUBR 1 arg

All the compositions of up to four car's and cdr's are defined as functions in their own right. The names begin with c and end with r, and in between is a sequence of a's and d's corresponding to the composition performed by the function.

For example,

```
(cddadr x) = (cdr (cdr (car (cdr x))))
```

Some of the most commonly used ones are: cadr, which gets the second element of a list. caddr, which gets the third element of a list. cadddr, which gets the fourth element of a list.

eq SUBR 2 args

eq is a predicate. (eq *x* *y*) is t if *x* and *y* are exactly the same object, nil otherwise. (cf. equal). It should be noted that things that print the same are not necessarily eq to each other. In particular, numbers with the same value need not be eq, and two similar lists are usually not eq. In general, two atomic symbols with the same print-name are eq, but it is possible with maknam or variable obarrays to generate symbols which have the same print-name but are not eq.

Examples:

```
(eq 'a 'b) => nil
(eq 'a 'a) => t
(eq '(a b) '(a b)) => nil (usually)
(setq x '(a b)) (eq x x) => t since it is
                  the same '(a b) in both arguments.
(eq 1 1) => t or nil depending on the implementation.
```

equal SUBR 2 args

The equal predicate returns t if its arguments are similar objects. (cf. eq) Two numbers are equal if they have the same value (a flonum is never equal to a fixnum though). Two strings are equal if they have the same length, and the contents are the same. All other atomic objects are equal if and only if they are eq. For dotted pairs and lists, equal is defined recursively as the two car's being equal and the two cdr's being equal. Thus equal could have been defined by:

```
(defun equal (x y)
      (or (eq x y)
          (and (equal (car x) (car y))
               (equal (cdr x) (cdr y)))))
```

As a consequence of this definition, it may be seen that equal need not terminate when applied to looped list structure. In addition, eq always implies equal. An intuitive definition of equal (which is not quite correct) is that two objects are equal if they look the same when printed out.

assoc SUBR 2 args

(assoc *x y*) looks up *x* in the association list (list of dotted pairs) *y*. The value is the first dotted pair whose car is equal to *x*, or nil if there is none such.

Examples:

```
(assoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
     => (r . x)
(assoc 'fooo '((foo . bar) (zoo . goo))) => nil
```

It is okay to rplacd the result of assoc as long as it is not nil.

Example:

```
(setq values '((x . 100) (y . 200) (z . 50)))
(assoc 'y values) => (y . 200)
(rplacd (assoc 'y values) 201)
(assoc 'y values) => (y . 201) now
```

(One should always be careful about using rplacd however)

assoc could have been defined by:

```
(defun assoc (x y)
    (cond ((null y) nil)
          ((equal x (caar y)) (car y))
          ((assoc x (cdr y))) ))
```

assq SUBR 2 args

assq is like assoc except that the comparison uses eq instead of equal. assq could have been defined by:

```
(defun assq (x y)
    (cond ((null y) nil)
          ((eq x (caar y)) (car y))
          ((assq x (cdr y))) ))
```

sassoc SUBR 3 args

(sassoc *x* *y* *z*) is like (assoc *x* *y*) except that if *x* is not found in *y*, instead of returning nil sassoc calls the function *z* with no arguments. sassoc could have been defined by:

```
(defun sassoc (x y z)
    (or (assoc x y)
        (apply z nil)))
```

sassq SUBR 3 args

(sassq *x* *y* *z*) is like (assq *x* *y*) except that if *x* is not found in *y*, instead of returning nil sassq calls the function *z* with no arguments. sassq could have been defined by:

```
(defun sassq (x y z)
    (or (assq x y)
        (apply z nil)))
```

last SUBR 1 arg

last returns the last dotted pair of the list which is its argument.
Example:

```
(setq x '(a b c d))
(last x) => (d)
(rplacd (last x) '(e f))
x => (a b c d e f)
```

last could have been defined by:

```
(defun last (x)
    (cond ((null x) x)
          ((null (cdr x)) x)
          ((last (cdr x))) ))
```

In some implementations, the null check above may be replaced by an atom check, which will catch dotted lists. Code which depends on this fact should not be written though, because all implementations are subject to change on this point.

length SUBR 1 arg

length returns the length of its argument, which must be a list. The length of a list is the number of top-level conses in it.
Examples:

```
(length nil) => 0
(length '(a b c d)) => 4
(length '(a (b c) d)) => 3
```

length could have been defined by:

```
(defun length (x)
    (cond ((null x) 0)
          ((1+ (length (cdr x)))) ))
```

The warning about dotted lists given under last applies also to length.

member SUBR 2 args

(member *x* *y*) returns nil if *x* is not a member of the list *y*. Otherwise, it returns the portion of *y* beginning with the first occurrence of *x*. The comparison is made by equal. *y* is searched on the top level only.

Example:

```
(member 'x '(1 2 3 4)) => nil
(member 'x '(a (x y) c x d e x f)) => (x d e x f)
```

Note that the value returned by member is eq to the portion of the list beginning with *x*. Thus rplaca on the result of member may be used, if you first check to make sure member did not return nil.
e.g.

```
(rplaca (or (member x z)
            (throw nil 'woops))
        y)
```

member could have been defined by:

```
(defun member (x y)
    (cond ((null y) nil)
          ((equal x (car y)) y) ((member x (cdr y))) ))
```

memq SUBR 2 args

memq is like member, except eq is used for the comparison, instead of equal. memq could have been defined by:

```
(defun memq (x y)
    (cond ((null y) nil)
          ((eq x (car y)) y) ((memq x (cdr y))) ))
```

not SUBR 1 arg

not returns t if its argument is nil, otherwise it returns nil.

null SUBR 1 arg

This is the same as not.

sxhash SUBR 1 arg

sxhash computes a hash code of an S-expression, and returns it as a fixnum which may be positive or negative. A property of sxhash is that (equal x y) implies (= (sxhash x) (sxhash y)). The number returned by sxhash is some possibly large number in the range allowed by fixnums. It is guaranteed that:
1) sxhash for an atomic symbol will always be positive.
2) sxhash of any particular expression will be constant in a particular implementation for all time.

3) Two different implementations may hash the same expression into different values.
4) sxhash of any object of type random will be zero.
5) sxhash of a fixnum will = that fixnum.

4.2 - Creating New List Structure

cons SUBR 2 args

This is a primitive function to construct a new dotted pair whose car is the first argument to cons, and whose cdr is the second argument to cons. Thus the following identities hold:

```
(eq (car (cons x y)) x) => t
(eq (cdr (cons x y)) y) => t
```

Examples:

```
(cons 'a 'b) => (a . b)
(cons 'a (cons 'b (cons 'c nil))) => (a b c)
(cons 'a '(b c d e f)) => (a b c d e f)
```

ncons SUBR 1 arg

(ncons *x*) = (cons *x* nil) = (list *x*)

xcons SUBR 2 args

xcons ("exchange cons") is like cons except that the order of arguments is reversed.
Example:

```
(xcons 'a 'b) => (b . a)
```

xcons could have been defined by: (defun xcons (x y) (cons y x))

list LSUBR 0 or more args

list constructs and returns a list of the values of its arguments.
Example:

```
(list 3 4 'a (car '(b . c)) (+ 6 -2)) => (3 4 a b 4)
```

append LSUBR 0 or more args

The arguments to append are lists. The result is a list which is the concatenation of the arguments. The arguments are not changed (cf. nconc). For example,

```
(append '(a b c) '(d e f) nil '(g)) => (a b c d e f g)
```

A version of append which only accepts two arguments could have been defined by:

```
(defun append (x y)
    (cond ((null x) y)
          ((cons (car x) (append (cdr x) y)) )))
```

The generalization to any number of arguments could then be made using a lexpr:

```
(defun full-append expr argcount
    (do ((i (1- argcount) (1- i))
         (val (arg argcount) (append (arg i) val)))
        ((zerop i) val) ))
```

reverse SUBR 1 arg

Given a list as argument, reverse creates a new list whose elements are the elements of its argument taken in reverse order. reverse does not modify its argument, unlike nreverse which is faster but does modify its argument. Example:

```
(reverse '(a b c d)) => (d c b a)
```

reverse could have been defined by:

```
(defun reverse (x)
    (do ((l x (cdr l))          ; scan down argument,
         (r nil
            (cons (car l) r))) ;putting each element into
        ((null l) r)))          ;list until no more elements.
```

subst SUBR 3 args

(subst *x* *y* *z*) substitutes *x* for all occurrences of *y* in *z*, and returns the modified *z*. The original *z* is unchanged, as subst recursively copies all of *z* replacing elements eq to *y* as it goes. If *x* and *y* are nil, *z* is completely copied, which is a convenient way to copy arbitrary list structure.

Example:

```
(subst 'Tempest 'Hurricane
       '(Shakespeare wrote (The Hurricane)))
  => (Shakespeare wrote (The Tempest))
```

subst could have been defined by:

```
(defun subst (x y z)
   (cond ((eq y z) x)    ;if item eq to y, replace.
         ((atom z) z) ;if no substructure, return arg.
         ((cons (subst x y (car z))
                (subst x y (cdr z))))))
```

sublis SUBR 2 args

sublis makes substitutions for atomic symbols in an S-expression. The first argument to sublis is a list of dotted pairs. The second argument is an S-expression. The return value is the S-expression with atoms that are the car of a dotted pair replaced by the cdr of that dotted pair. The argument is not modified - new conses are created where necessary and only where necessary, so the newly created structure shares as much of its substructure as possible with the old. For example, if no successful substitutions are made, the result is eq to the second argument.

Example:

```
(sublis '((x . 100) (z . zprime))
        '(plus x (minus g z x p) 4))
   => (plus 100 (minus g zprime 100 p) 4)
```

4.3 - Modifying Existing List Structure

rplaca SUBR 2 args

(rplaca *x* *y*) changes the car of *x* to *y* and returns (the modified) *x*.
Example:

```
    (setq g '(a b c))
    (rplaca (cdr g) 'd) => (d c)
   Now g => (a d c)
```

rplacd SUBR 2 args

(rplacd *x* *y*) changes the cdr of *x* to *y* and returns (the modified) *x*.
Example:

```
    (setq x '(a b c))
    (rplacd x 'd) => (a . d)
   Now x => (a . d)
```

nconc LSUBR 0 or more args

nconc takes lists as arguments. It returns a list which is the arguments concatenated together. The arguments are changed, rather than copied. (cf. append)
Example:

```
(nconc '(a b c) '(d e f)) => (a b c d e f)
```

Note that the constant (a b c) has now been changed to (a b c d e f). If this form is evaluated again, it will yield (a b c d e f d e f).

nconc could have been defined by:

```
(defun nconc (x y)         ;for simplicity,  this definition
    (cond ((null x) y)     ;only works for 2 arguments.
      (t
       (rplacd (last x) y);hook y onto x
       x)))                ;and return the modified x.
```

nreverse SUBR 1 arg

nreverse reverses its argument, which should be a list. The argument is destroyed by rplacd's all through the list (cf. reverse).
Example:

```
(nreverse '(a b c)) => (c b a)
```

nreverse could have been defined by:

```
(defun nreverse (x)
    (cond ((null x) nil)
          ((*nrev x nil))))

(defun *nrev (x y)                    ;auxiliary function
    (cond ((null (cdr x)) (rplacd x y))
          ((*nrev (cdr x) (rplacd x y)))))
;; this last call depends on order of argument evaluation.
```

delete LSUBR 2 or 3 args

(delete *x* *y*) returns the list *y* with all top-level occurrences of *x* removed. equal is used for the comparison. The argument *y* is actually modified (rplacd'ed) when instances of *x* are spliced out.

(delete *x* *y* *n*) is like (delete *x* *y*) except only the first *n* instances of *x* are deleted. *n* is allowed to be zero. If *n* is greater than the number of occurences of *x* in the list, all occurrences of *x* in the list will be deleted.

Example: (delete 'a '(b a c (a b) d a e)) => (b c (a b) d e)

delete could have been defined by:

```
(defun delete nargs          ; lexpr definition for 2 or 3 args
    (*delete (arg 1)         ; pass along arguments...
             (arg 2)
             (cond ((= nargs 3) (arg 3))
                   (123456789.)))) ; infinity

(defun *delete (x y n)    ;auxiliary function
    (cond ((or (null y) (zerop n)) y)
          ((equal x (car y)) (*delete x
                                      (cdr y)
                                      (1- n)))
          ((rplacd y (*delete x (cdr y) n)))))
```

delq LSUBR 2 or 3 args

delq is the same as delete except that eq is used for the comparison instead of equal. See delete.

This page intentionally left blank.

5 - Flow of Control

MACLISP provides a variety of structures for flow of control. Conditionals allow control to branch depending on the value of a predicate. and and or are basically one-arm conditionals, while cond is a generalized multi-armed conditional.

Recursion consists of doing part of the work that is to be done oneself, and handing off the rest to someone else to take care of, when that someone else happens to be (another invocation of) oneself.

Iteration is a control structure present in most languages. It is similar to recursion but sometimes more useful and sometimes less useful. MACLISP contains a generalized iteration facility. The iteration facility also permits those who like "gotos" to use them.

Nonlocal exits are similar to a return, except that the return is from several levels of function calling rather than just one, and is determined at run time. These are mostly used for applications like escaping from the middle of a function when it is discovered that the algorithm is not applicable.

Errors are a type of non-local exit used by the lisp interpreter when it discovers a condition that it does not like. Errors have the additional feature of correctability, which allows a user-specified function (most often a break loop), to get a chance to come in and correct the error or at least inspect what was happening and determine what caused it, before the nonlocal exit occurs. This is explained in detail in section 12.4.

5.1 - Conditionals

and FSUBR

and evaluates its arguments one at a time, from left to right. If any argument evaluates to nil, and immediately returns nil without evaluating the remaining arguments. If all the arguments evaluate non-nil, and returns the value of its last argument. and can be used both for logical operations, where nil stands for False and t stands for True, and as a conditional expression.
Examples:

```
(and x y)

(and (setq temp (assq x y))
     (rplacd temp z))

(and (null (errset (something)))
     (princ "There was an error."))
```

Note: (and) => t, which is the identity for this operation.

or FSUBR

or evaluates its arguments one by one from left to right. If an argument evaluates to nil, or proceeds to evaluate the next argument. If there are no more arguments, or returns nil. But if an argument evaluates non-nil, or immediately returns that value without evaluating any remaining arguments. or can be used both for logical operations, where nil stands for False and t for True, and as a conditional expression.
Note: (or) => nil, the identity for this operation.

cond FSUBR

cond processes its arguments, called "clauses," from left to right. The car of each clause, called the "antecedent," is evaluated. If it is nil, cond advances to the next clause. Otherwise, the cdr of the clause is treated as a list of forms, called "consequents," which are evaluated from left to right. After evaluating the consequents, cond returns without inspecting any remaining clauses. The value is the value of the last consequent evaluated, or the value of the antecedent if there were no consequents in the clause. If cond runs out of clauses (i.e. if every antecedent is nil), the value of the cond is nil.

Example:

```
    (cond ((zerop x) (+ y 3)) ;first clause.
                              ;(zerop x) is antecedent.
                              ;(+ y 3) is consequent.
          ((null y)
           (setq x 4)
           (cons x z))        ;a clause with 2 consequents
          (z)                 ;a clause with no consequents.
                              ;the antecedent is just z.
          )                   ;this is the end of the cond.
```

This is like the traditional LISP 1.5 cond except that it is not necessary to have exactly one consequent in each clause, and it is permissible to run out of clauses.

5.2 - Iteration

prog FSUBR

prog is the "program" function. It provides temporary variables, sequential evaluation of statements, and the ability to do "gotos." The form of a prog is:

```
                     _                 _
                    |      <tag>        |
(prog (<var>...) <                        > ...)
                    |_   <statement>   _|
```

The first thing in a prog is a list of temporary variables <var>. Each variable has its value saved when the prog is entered and restored when the prog is left. The variables are initialized to nil when the prog is entered, thus they are said to be "bound to nil" by the prog.

The rest of a prog is the body. An item in the body may be an atomic symbol which is a <tag> or a non-atomic <statement>.

prog, after binding the temporary variables, evaluates its body sequentially. <tag>s are skipped over; <statement>s are evaluated but the values are ignored. If the end of the body is reached, prog returns nil. If (return *x*) is evaluated, prog stops evaluating its body and returns the value of *x*. If (go ***tag***) is seen, prog jumps to the part of the body labelled with the *tag*. The argument to go is not evaluated unless it is non-atomic.

It should be noted that the prog function is an extension of the LISP 1.5 prog function, in that go's and return's may occur in more places than LISP 1.5 allowed. However, the LISP compilers implemented on ITS, Multics, and the DECsystem 10 for MACLISP require that go's and return's be lexically within the scope of the prog. This makes a function which does not contain a prog, but which does contain a go or return uncompilable.

See also the do function, which uses a body similar to prog. The do function and the catch and throw functions are included in MACLISP as an attempt to encourage goto-less programming style, which leads to more readable, more easily maintained code. The programmer is recommended to use these functions instead of prog wherever possible.

Example:

```
(prog (x y z)   ;x, y, z are prog variables   - temporaries.
   (setq y (car w) z (cdr w))       ;w is a free variable.
loop
   (cond ((null y) (return x))
         ((null z) (go err)))
rejoin
   (setq x (cons (cons (car y) (car z))
                 x))
   (setq y (cdr y)
         z (cdr z))
   (go loop)
err
   (break are-you-sure? t)
   (setq z  y)
   (go rejoin))
```

do FSUBR

do provides a generalized "do loop" facility, with an arbitrary number of "control variables" whose values are saved when the do is entered and restored when it is left, i.e. they are bound by the do. The control variables are used in the iteration performed by do. At the beginning they are initialized to specified values, and then at the end of each trip around the loop the values of the control variables are changed according to specified rules and iteration continues until a specified end condition is satisfied. do comes in two forms.

The newer form of do is:

```
(do ((<var> <init> <repeat>)...)
    (<end-test> <exit-form>...)
    <body>...)
```

The first argument of do is a list of zero or more control variable specifiers. Each control variable specifier has as its car the name of a variable, as its cadr an initial value <init>, which defaults to nil if it is omitted, and as its caddr a repeat value <repeat>. If <repeat> is omitted, the <var> is not changed between loops.

All assignment to the control variables is done in parallel. At the beginning of the first iteration, all the <init>s are evaluated, then the <var>s are saved, then the <var>s are setq'ed to the <init>s. Note that the <init>s are evaluated *before* the <var>s are bound. At the beginning of each succeeding iteration those <var>s that have <repeat>s get setq'ed to their respective <repeat>s. Note that all the <repeat>s are evaluated before any of the <var>s are changed.

The second argument of do is a list of an end testing predicate

<end-test> and zero or more forms, the <exit-form>s. At the beginning of each iteration, after processing of the <repeat>s, the <end-test> is evaluated. If the result is nil, execution proceeds with the body of the do. If the result is not nil, the <exit-forms> are evaluated from left to right and then do returns. The value of the do is the value of the last <exit-form>, or nil if there were no <exit-form>s. Note that the second argument to do is similar to a cond clause.

If the second argument to do is nil, there is no <end-test> or <exit-form>s, and the the body of the do is executed only once. In this type of do it is an error to have <repeat>s. This type of do is a "prog with initial values."

The remaining arguments to do constitute a prog body. When the end of the body is reached, the next iteration of the do begins. If return is used, do returns the indicated value and no more iterations occur.

The older form is:

```
(do <var> <init> <repeat> <end-test> <body>...)
```

The first time through the loop <var> gets the value of <init>; the remaining times through the loop it gets the value of <repeat>, which is re-evaluated each time. Note that <init> is evaluated before the value of <var> is saved. After <var> is set, <end-test> is evaluated. If it is non-nil, the do finishes and returns nil. If the <end-test> is nil, the <body> of the loop is executed. The <body> is like a prog body. go may be used. If return is used, its argument is the value of the do. If the end of the prog body is reached, another loop begins.

Examples of the old form of do:

```
(do i 0 (1+ i) (> i (cadr (arraydims x)))
        (store (x i) 0))                ;zeroes out the array x

(do zz x (cdr zz) (or (null zz) (zerop (f (car zz)))))
                    ; this applies f to each element of x
                    ; continuously until f returns zero.
```

Examples of the new form of do:

```
    (do ((x) (y) (z)) (nil) <body>)
```

is like

```
    (prog (x y z) <body>)
```

except that when it runs off the end of the <body>, do loops but prog returns nil.

```
(do ((x y (f x))) ((p x)) <body>)
```
is like
```
(do x y (f x) (p x) <body>)
```

```
(do ((x x (cdr x))
     (y y (cdr y))
     (z nil (cons (f x y) z))) ; exploits parallel assignment
    ((or (null x) (null y))
     (nreverse z)))            ; typical use of nreverse
                               ; body has been omitted
```
is like (maplist ′f x y).

```
(do ((x e (cdr x)) (oldx x x)) ((null x)) <body>)
```

This exploits the parallel assignment to control variables. On the first iteration, the value of oldx is whatever value x had before the do was entered. On succeeding iterations, oldx contains the value that x had on the previous iteration.

In either form of do, the <body> may contain no forms at all.

go FSUBR

The go function is used to do a "go-to" within the body of a do or a prog. If the argument is an atom, it is not evaluated. Otherwise it is repeatedly evaluated until it is an atom. Then go transfers control to the point in the body labelled by a tag eq or = to the argument. (Tags may be either atomic symbols or numbers). If there is no such tag in the body, it is an unseen-go-tag error.
Example:

```
(prog (x y z)
  (setq x something)
loop
  <do something>
  (and <some predicate> (go loop))     ;regular go
  <do something more>
  (go (cond ((minusp x) ′loop)         ;"computed go"
            (t ′endtag)))
endtag
  (return z))
```

`return` SUBR 1 arg

`return` is used to return from a `prog` or a `do`. The value of `return`'s argument is returned by `prog` or `do` as its value. In addition, `break` recognizes the top level form (`return` *value*) specially. If this form is typed at a `break`, *value* will be evaluated and returned as the value of `break`. If not at the top level of a form typed at a `break`, and not inside a `prog` or `do`, `return` will cause a `fail-act` error.

Example:

```
(prog (x)
        (setq x (reverse y))
        (or (cddr x) (return (cadr x)))
        (return (caddr x)))
```

If `y` is `'(z y x w y u t s)`, this returns `u`. If `y` is `'(a b)`, this returns `a`.

5.3 - Nonlocal Exits

catch FSUBR

catch is the LISP function for doing structured non-local exits. (catch *x*) evaluates *x* and returns its value, except that if during the evaluation of *x* (throw *y*) should be evaluated, catch immediately returns *y* without further evaluating *x*.

catch may also be used with a second argument, not evaluated, which is used as a tag to distinguish between nested catches. (catch *x b*) will catch a (throw *y b*) but not a (throw *y z*). throw with only one argument always throws to the innermost catch. catch with only one argument catches any throw. It is an error if throw is done when there is no suitable catch.

Example:

```
(catch
  (mapcar (function (lambda (x)
                  (cond ((minusp x)
                         (throw x negative))
                        (t (f x)) )))
       y)
    negative)
```

which returns a list of f of each element of y if y is all positive, otherwise the first negative member of y.

The user of catch and throw is recommended to stick to the 2 argument versions, which are no less efficient, and tend to reduce the likelihood of bugs. The one argument versions exist primarily as an easy way to fix old LISP programs which use errset and err for non-local exits. This latter practice is rather confusing, because err and errset are supposed to be used for error handling, not general program control.

throw FSUBR

throw is used with catch as a structured nonlocal exit mechanism.

(throw *x*) evaluates *x* and throws the value back to the most recent catch.

(throw *x* <tag>) throws the value of *x* back to the most recent catch labelled with <tag> or unlabelled. catch'es with tags not eq to <tag> are skipped over. *x* is evaluated but <tag> is not.
See the description of catch for further details.

5.4 - Causing and Controlling Errors

`error` LSUBR 0 to 3 args

This is a function which allows user functions to signal their own errors using the LISP error system.

`(error)` is the same as `(err)`.

`(error` ***message***`)` signals a simple error - no datum is printed and no user interrupt is signalled. The error message typed out is ***message***.

`(error` ***message datum***`)` signals an error with ***message*** as the message to be typed out, ***datum*** as the LISP object to be printed in the error message. No user interrupt is signalled.

`(error` ***message datum uint-chn***`)` signals an error but first signals a user interrupt on channel ***uint-chn***, provided that there is such a channel and it has a non-`nil` service function. ***uint-chn*** may be the channel number or the atomic symbol whose value is the interrupt service function for the channel -- see section 12.4.2. If the service function returns an atom, `error` goes ahead and signals a regular error. If the service function returns a list, `error` returns as its value the `car` of that list. In this case it was a "correctable" error. This is the only case in which `error` will return.

`errset` FSUBR

`errset` evaluates its first argument. If no errors occur, the result is `cons`'ed with `nil` and returned. If an error occurs during the evaluation of the first argument, the error is prevented from escaping from inside the `errset` and `errset` returns `nil`. `errset` may also be made to return any arbitrary value by use of the `err` function.

If a second argument is given to `errset`, it is not evaluated. If it is `nil`, no error message will be printed if an error occurs during the evaluation of `errset`'s first argument. If the second argument is not `nil`, or if `errset` is used with only one argument, any error messages generated will be printed.
Examples:

If you are not sure `x` is a number:

```
(errset (setq x (add1 x)))
```

This last example may not work in compiled code if the compiler chooses to open-code the `add1` rather than calling the `add1` subroutine. The user of such code must be extremely careful if he wishes to use it

compiled.

To suppress message if the value of a is not an atomic symbol:

```
(errset (set a b) nil)
```

To do the same but generate one's own message:

```
(or (errset (set a b) nil)
    (print (list a 'is 'not 'a 'variable)))
```

err FSUBR

(err) causes an error which is handled the same as a LISP error except that there is no preliminary user interrupt, and no message is typed out.

(err *x*) is like (err) except that if control returns to an errset, the value of the errset will be the result of evaluating *x*, instead of nil.

(err *x* nil) is the same as (err *x*). (err *x* t) is like (err *x*) except that *x* is not evaluated until just before the errset returns it. That is, *x* is evaluated *after* unwinding the pdl and restoring the bindings.

Note: some people use err and errset where catch and throw are indicated. This is a very poor programming practice. See writeups of catch and throw for details.

This page intentionally left blank.

6 - Manipulating the Constituents of Atomic Symbols

6.1 - The Value Cell

Each atomic symbol has associated with it a "value cell," which is a piece of storage that can hold a LISP object. Initially this value cell is "unbound" or "undefined," i.e. empty. An object can be placed into an atomic symbol's value cell by setq'ing or binding. Once this has been done, this object will be returned when the atomic symbol is evaluated. The atomic symbol is said to have this object as its value.

setq FSUBR

setq is used to assign values to variables (atomic symbols.) setq takes its arguments in pairs, and processes them sequentially, left to right. The first member of each pair is the variable, the second is the value. The value is evaluated but the variable is not. The value of the variable is set to the value specified. You must not setq the special atomic-symbol constants t and nil. The value returned by setq is the last value assigned, i.e. the value of its last argument.

Example: (setq x (+ 1 2 3) y (cons x nil))
returns (6) and gives x a value of 6 and y a value of (6).

Note that the first assignment is processed before the second assignment is done, resulting in the second use of x getting the value assigned in the first pair of the setq.

set SUBR 2 args

set is like setq except that the first argument is evaluated; also set only takes one pair of arguments. The first argument must evaluate to an atomic symbol, whose value is changed to the value of the second argument. set returns the value of its second argument. Example:

```
(set (cond ((predicate) 'atom1) (t 'atom2)) 'stba)
```

evaluates to stba and gives either atom1 or atom2 a value of stba.

set could have been defined by:

```
(defun set (x y)
    (apply 'setq (list x (list 'quote y)) ))
```

boundp SUBR 1 arg

The argument to boundp must be an atomic symbol. If it has a value, cons of nil with that value is returned. Otherwise nil is returned. Example:

```
(boundp 't) => (nil . t)          ;since the value of t is t
```

definedp SUBR 1 arg

This predicate returns t if its argument (a symbol) has a value, and nil otherwise.

makunbound SUBR 1 arg

The argument to makunbound must be an atomic symbol. Its value is removed and it becomes undefined, which is the initial state for atomic symbols.
Example:

```
(setq a 1)
a => 1
(makunbound 'a)
a => unbnd-vrbl error.
```

makunbound returns its argument.

6.2 - The Property List

A property-list is a list with an even number of elements. Each pair of elements constitutes a property: the first element is called the "indicator" and the second is called the "value." The indicator is generally an atomic symbol which serves as the name of the property. For example, one type of functional property uses the atom expr as its indicator. The value is a LISP object. In the case of an expr-property, the value is a list beginning with lambda. An example of a property list with two properties on it is:

```
(expr (lambda (x) (plus 14 x)) foobar t)
```

The first property has indicator expr and value (lambda (x) (plus 14 x)), the second property has indicator foobar and value t.

Each atomic symbol has associated with it a property-list which is kept on its cdr. It is also possible to have "disembodied" property lists which are not associated with any atom. These also keep the property list on their cdr, as the form of a disembodied property list is (<anything> . plist). The way to create a disembodied property list is (ncons nil).

It is all right to ask for a property of a number, using the get and getl functions described below. nil will always be returned since numbers have no properties.

The user familiar with LISP 1.5 will want to note that the property list "flags" which are allowed on LISP 1.5 property lists do not exist in MACLISP.

Some implementations have special args, pname, and value properties which are used to store internal information. Consequently user programs should never use properties with these names.

get SUBR 2 args

(get *x* *y*) gets *x*'s *y*-property. *x* can be an atomic symbol or a disembodied property list. The value of *x*'s *y*-property is returned,

unless *x* **has no** *y***-property in which case** nil **is returned. Example:**

```
(get 'foo 'bar)
  => nil             ;foo has no bar property
(putprop 'foo 'zoo 'bar) ;give foo a bar property
  => zoo
(get 'foo 'bar)
  => zoo
(cdr 'foo)           ;look at foo's property list.
  => (bar zoo ...other properties...)
```

getl   SUBR 2 args

(getl *x* *y*) **is like** get **except that** *y* **is a list of properties rather than just a single property.** getl **searches** *x***'s property list until a property in the list** *y* **is found. The portion of** *x***'s property list beginning with this property is returned. The** car **of this is the property name and the** cadr **is what** get **would have returned.** getl **returns** nil **if none of the properties in** *y* **appear on the property list of** *x***.** getl **could have been defined by:**

```
(defun getl (x pl)
   (do ((q (cdr x) (cddr q))) ; scan down cdr of x
       ((or (null q) (memq (car q) pl)) q)))
```

putprop   SUBR 3 args

(putprop *x* *y* *z*) **gives** *x* **a** *z***-property of** *y* **and returns** *y***.** *x* **may be an atomic symbol or a disembodied property list. After somebody does** (putprop *x* *y* *z*), (get *x* *z*) **will return** *y***.**

Example: (putprop 'foo 'bar 'often-with)

defprop   FSUBR

(defprop *x* *y* *z*) **gives** *x* **a** *z***-property of** *y***. The arguments are not evaluated. Example:**

(defprop foo bar often-with)

`remprop` SUBR 2 args

(remprop *x* *y*) removes *x*'s *y*-property, by splicing it out of *x*'s property list. The value is t if *x* had a *y*-property, nil if it didn't. *x* may be an atomic symbol or a disembodied property list. Example:

```
(remprop 'foo 'expr)
```

undefines the function foo if it was defined by

```
(defun foo (x) ... )
```

6.3 - The Print-Name

Each atomic symbol has an associated character string called its "print-name," or "pname" for short. This character string is used as the external representation of the symbol. If the string is typed in, it is read as a reference to the symbol. If the symbol is asked to be print'ed, the string is typed out. Generally pnames are unique - there is only one atomic symbol whose pname is a particular string of characters. However, by using multiple obarrays (see section 12.4) or "uninterned" atomic symbols (ones whose pnames are not "interned" or registered in an obarray), it is possible to get two atoms with the same pname.

See also Chapter 8, on strings, for some other functions which have to do with pnames.

samepnamep SUBR 2 args

The arguments to samepnamep must evaluate to atomic symbols or to character strings. The result is t if they have the same pname, nil otherwise. The pname of a character string is considered to be the string itself.
Examples:

```
(samepnamep 'xyz (maknam '(x y z))) => t

(samepnamep 'xyz (maknam '(w x y))) => nil

(samepnamep 'x "x") => t
```

alphalessp SUBR 2 args

(alphalessp *x* *y*), where *x* and *y* evaluate to atomic symbols or character strings, returns t if the pname of *x* occurs earlier in alphabetical order than the pname of *y*. The pname of a character string is considered to be the string itself. Examples:

```
(alphalessp 'x 'x1) => t
(alphalessp 'z 'q) => nil
(alphalessp "x" 'y) => t
```

Note that the "alphabetical order" used by alphalessp is actually the ASCII collating sequence. Consequently all upper case letters sort before all lower case letters.

6.4 - Miscellaneous Functions

getchar SUBR 2 args

(getchar *x n*), where *x* is an atomic symbol and *n* is a fixnum, returns the *n*'th character of *x*'s pname, where *n* = 1 selects the leftmost character. The character is returned as a character object. nil is returned if *n* is out of bounds.

intern SUBR 1 arg

(intern *x*), where *x* is an atomic symbol, returns an atomic symbol which is "interned on the obarray" and has the same pname as *x*. If *x* is not already interned on the current obarray, this will be a copy of it. It is as if *x* was printed out and read back in.

remob SUBR 1 arg

The argument to remob must be an atomic symbol. It is removed from the current obarray if it is interned on that obarray. This makes the atomic symbol inaccessible to any S-expressions that may be read in or loaded in the future. remob returns nil.

gensym LSUBR 0 or 1 args

gensym creates and returns a new atomic symbol, which is ***not*** interned on the obarray (is not recognized by read.) The atomic symbol's pname is of the form <prefix><number>, e.g. g0001. The <number> is incremented each time.

If gensym is given an argument, a numeric argument is used to set the <number>. The pname of an atomic-symbol argument is used to set the <prefix>. For example:

```
if   (gensym) => g0007
then (gensym 'foo) => f0008
     (gensym 40) => f0032
and  (gensym) => f0033
```

Note that the <number> is in decimal and always four digits, and the <prefix> is always one character.

6.5 -. Defining Atomic Symbols as Functions

Atomic symbols may be used as names for functions. This is done by putting the actual function (a subr-object or a lambda-expression) on the property list of the atomic symbol as a "functional property," i.e. under one of the indicators expr, fexpr, macro, subr, lsubr, or fsubr.

Array properties (see chapter 9) are also considered to be functional properties, so an atomic symbol which is the name of an array is also the name of a function, the accessing function of that array.

When an atomic symbol which is the name of a function appears in function position in a form being evaluated, or is "applied," the function which it names is used.

args LSUBR 1 or 2 args

(args *f*) determines the number of arguments expected by the function *f*. If *f* wants *n* arguments, args returns (nil . *n*). If *f* can take from *m* to *n* arguments, args returns (*m* . *n*). If *f* is an fsubr or a lexpr, expr, or fexpr, the results are meaningless.

(args *f x*), where *x* is (nil . *n*) or (*m* . *n*), sets the number of arguments desired by the function *f*. This only works for compiled, non-system functions.

defun FSUBR

defun is used for defining functions. The general form is:

```
(defun <name> <type>
       ( <lambda-variable>...)
       <body>...)
```

however, <name> and <type> may be interchanged. <type>, which is optional, may be expr, fexpr, or macro. If it is omitted, expr is assumed. Examples:

```
(defun addone (x) (1+ x))              ;defines an expr

(defun quot fexpr (x) (car x))         ;defines a fexpr

(defun fexpr quot (x) (car x))         ;is the same

(defun zzz expr x
    (foo (arg 1)(arg 2)))      ;this is how you define a lexpr.
```

Note: the functions defprop and putprop may also be used for defining functions.

This page intentionally left blank.

7 - Functions on Numbers

For a description of the various types of numbers used in MACLISP, see chapter 2.

7.1 - Number Predicates

`bigp` SUBR 1 arg

The predicate `bigp` returns `t` if its argument is a bignum, and `nil` otherwise.

`zerop` SUBR 1 arg

The `zerop` predicate returns `t` if its argument is fixnum zero or flonum zero. (There is no bignum zero.) Otherwise it returns `nil`.

`plusp` SUBR 1 arg

The `plusp` predicate returns `t` if its argument is strictly greater than zero, `nil` if it is zero or negative. It is an error if the argument is not a number.

`minusp` SUBR 1 arg

The `minusp` predicate returns `t` if its argument is a negative number, `nil` if it is a non-negative number. It is an error if the argument is not a number.

`oddp` SUBR 1 arg

The `oddp` predicate returns `t` if its argument is an odd number, otherwise `nil`. The argument must be a fixnum or a bignum.

signp FSUBR

The signp predicate is used to test the sign of a number. (signp *c* *x*) returns t if *x*'s sign satisfies the test *c*, nil if it does not. *x* is evaluated but *c* is not. It is an error if *x* is not a number. *c* can be one of the following:

l	means	x<0
le	"	x≤0
e	"	x=0
n	"	x≠0
ge	"	x≥0
g	"	x>0

Examples:

```
(signp le -1) => t
(signp n 0) => nil
```

haulong SUBR 1 arg

(haulong *x*) returns the number of significant bits in *x*. *x* can be a fixnum or a bignum. The result is the least integer not less than the base-2 logarithm of |*x*|+1. Examples:

```
(haulong 0) => 0
(haulong 3) => 2
(haulong -7) => 3
(haulong 12345671234567) => 40.
```

7.2 - Comparison

= SUBR 2 args

(= *x* *y*) is t if *x* and *y* are numerically equal. *x* and *y* must be both fixnums or both flonums.

greaterp LSUBR 2 or more args

greaterp compares its arguments, which must be numbers, from left to right. If any argument is not greater than the next, greaterp returns nil. But if the arguments to greaterp are strictly decreasing, the result is t. Examples:

```
(greaterp 4 3) => t
(greaterp 1 1) => nil
(greaterp 4.0 3.6 -2) => t
(greaterp 4 3 1 2 0) => nil
```

> SUBR 2 args

(> *x* *y*) is t if *x* is strictly greater than *y*, and nil otherwise. *x* and *y* must be both fixnums or both flonums.

lessp LSUBR 2 or more args

lessp compares its arguments, which must be numbers, from left to right. If any argument is not less than the next, lessp returns nil. But if the arguments to lessp are strictly increasing, the result is t. Examples:

```
(lessp 3 4) => t
(lessp 1 1) => nil
(lessp -2 3.6 4) => t
(lessp 0 2 1 3 4) => nil
```

< SUBR 2 args

(< *x* *y*) is t if *x* is strictly less than *y*, and nil otherwise. *x* and *y* must be both fixnums or both flonums.

`max` LSUBR 1 or more args

`max` returns the largest of its arguments, which must be numbers.

`min` LSUBR 1 or more args

`min` returns the smallest of its arguments, which must be numbers.

7.3 - Conversion

`fix` SUBR 1 arg

(fix *x*) converts *x* to a fixnum or a bignum depending on its magnitude. Examples:

```
(fix 7.3) => 7
(fix -1.2) => -2
```

`float` SUBR 1 arg

(float *x*) converts *x* to a flonum. Example:

```
(float 4) => 4.0
```

`abs` SUBR 1 arg

(abs *x*) => $|x|$, the absolute value of the number *x*. abs could have been defined by:

```
(defun abs (x) (cond ((minusp x) (minus x))
                     (x) ))
```

`minus` SUBR 1 arg

minus returns the negative of its argument, which can be any kind of number. Examples:

```
(minus 1) => -1
(minus -3.6) => 3.6
```

`haipart` SUBR 2 args

(haipart *x n*) extracts *n* leading or trailing bits from the internal representation of *x*. *x* may be a fixnum or a bignum. *n* must be a fixnum. The value is returned as a fixnum or a bignum. If *n* is positive, the result contains the *n* high-order significant bits of abs(*x*). If *n* is negative, the result contains the abs(*n*) low-order bits of abs(*x*). If abs(*n*) is bigger than the number of significant bits in *x*, abs(*x*) is returned.

Examples:

```
(haipart 34567 7) => 162

(haipart 34567 -5) => 27

(haipart -34567 -5) => 27
```

7.4 - Arithmetic

General Arithmetic

plus LSUBR 0 or more args

plus returns the sum of its arguments, which may be any kind of numbers. Conversions to flonum or bignum representation are done as needed. Flonum representation will be used if any of the arguments are flonums; otherwise fixnum representation will be used if the result can fit in fixnum form. If it cannot, bignum representation will be used.

difference LSUBR 1 or more args

difference returns its first argument minus the rest of its arguments. It works for any kind of numbers.

times LSUBR 0 or more args

times returns the product of its arguments. It works for any kind of numbers.

quotient LSUBR 1 or more args

quotient returns its first argument divided by the rest of its arguments. The arguments may any kind of number. (cf. / and /$)
Examples:

```
(quotient 3 2)  => 1      ;fixnum division truncates

(quotient 3 2.0) =>  1.5  ;but flonum division does not.

(quotient 6.0 1.5  2.0) => 2.0
```

add1 SUBR 1 arg

(add1 *x*) => *x*+1. *x* may be any kind of number.

sub1 SUBR 1 arg

(sub1 x) => x-1. x may be any kind of number.

remainder SUBR 2 args

(remainder x y) => the remainder of the division of x by y. The sign of the remainder is the same as the sign of the dividend. The arguments must be fixnums or bignums.

gcd SUBR 2 args

(gcd x y) => the greatest common divisor of x and y. The arguments must be fixnums or bignums.

expt SUBR 2 args

(expt x y) = x^y

The exponent y may be a bignum if the base x is 0, 1, or -1; otherwise y must be a fixnum. x may be any kind of number.

Fixnum Arithmetic

+ LSUBR 0 or more args

+ returns the sum of its arguments. The arguments must be fixnums, and the result is always a fixnum. Overflow is ignored. Examples:

```
(+ 2 6 -1) => 7
(+ 3) => 3          ;trivial case
(+) => 0            ;identity element
```

- LSUBR 0 or more args

This is the fixnum-only subtraction function. It does not detect overflows.

```
(-) => 0, the identity element
(- x) => the negative of x.
(- x y) => x - y.
(- x y z) => x - y - z
```

etc.

* LSUBR 0 or more args

* returns the product of its arguments. The arguments must be fixnums. The result is always a fixnum. Overflow is not detected. Examples:

```
(* 4 5 -6) => -120.
(* 3) => 3          ;trivial case
(*) => 1            ;identity element
```

/ LSUBR 0 or more args

This is the fixnum-only division function. The arguments must be fixnums and the result of the division is truncated to an integer and returned as a fixnum. Note that the name of this function must be typed in as //, since LISP uses / as an escape character.

(//) => 1, the identity element.
(// x) => the fixnum reciprocal of x, which is 0 if $|x| > 1$.
(// x y) => x/y.
(// x y z) => $(x/y)/z$.

etc.

1+ SUBR 1 arg

(1+ x) => x+1. x must be a fixnum. The result is always a fixnum. Overflow is ignored.

1- SUBR 1 arg

(1- x) => x-1. x must be a fixnum. The result is always a fixnum and overflow is not detected.

\ SUBR 2 args

(\ x y) returns the remainder of x divided by y, with the sign of x. x and y must be fixnums. Examples:

```
(\ 5 2) => 1
(\ 65. -9.) => 2
(\ -65. 9.) => -2
```

Flonum Arithmetic

+$ LSUBR 0 or more args

+$ returns the sum of its arguments. The arguments must be flonums and the result is always a flonum. Examples:

```
(+$ 4.1 3.14) => 7.24
(+$  2.0 1.5  -3.6)  => -0.1
(+$ 2.6) => 2.6                 ;trivial case
(+$)  => 0.0                    ;identity element
```

-$ LSUBR 0 or more args

This is the flonum-only subtraction function.

(-$) => 0.0, the identity element
(-$ *x*) => the negation of *x*.
(-$ *x y*) => $x - y$.
(-$ *x y z*) => $x - y - z$.
etc.

***$** LSUBR 0 or more args

*$ returns the product of its arguments. The arguments must be flonums and the result is always a flonum. Examples:

```
(*$ 3.0 2.0 4.0) => 24.0
(*$ 6.1) => 6.1     ;trivial case
(*$) => 1.0         ;identity element
```

/$ LSUBR 0 or more args

This is the flonum-only division function. Note that the name of this function must be typed in as //$, since LISP uses / as an escape character.

(//$) => 1.0, the identity element
(//$ *x*) => the reciprocal of *x*.
(//$ *x y*) => x/y
(//$ *x y z*) => $(x/y)/z$.
etc.

1+$ SUBR 1 arg

(1+$ *x*) => *x*+1.0. *x* must be a flonum. The result is always a flonum.

1-$ SUBR 1 arg

(1-$ *x*) => *x*-1.0. *x* must be a flonum. The result is always a flonum.

7.5 - Exponentiation and Log Functions

sqrt SUBR 1 arg

(sqrt *x*) => a flonum which is the square root of the number *x*.

isqrt SUBR 1 arg

(isqrt *x*) => a fixnum which is the square root of *x*, truncated to an integer.

exp SUBR 1 arg

(exp *x*) = e^x

log SUBR 1 arg

(log *x*) = the natural log of *x*.

7.6 - Trigonometric Functions

sin SUBR 1 arg

(sin *x*) gives the trigonometric sine of *x*. *x* is in radians. *x* may be a fixnum or a flonum.

cos SUBR 1 arg

(cos *x*) returns the cosine of *x*. *x* is in radians. *x* may be a fixnum or a flonum.

atan LSUBR 1 or 2 args

(atan *x*) returns the arctangent of *x*, in radians. *x* and *y* may be fixnums or flonums. (atan *x* *y*) returns the arctangent of *x*/*y*, in radians. *y* may be 0 as long as *x* is not also 0.

7.7 - Random Functions

random LSUBR 0 or 1 arg

(random) returns a random fixnum.

(random nil) restarts the random sequence at its beginning.

(random x), where x is a fixnum, returns a random fixnum between 0 and x-1 inclusive.

zunderflow SWITCH

If the value of zunderflow is non-nil, floating point arithmetic underflow will produce a result of 0.0. If the value of zunderflow is nil, any underflows that occur will cause fail-acts.

7.8 - Logical Operations on Numbers

boole LSUBR 3 or more args

(boole *k x y*) computes a bit by bit Boolean function of the fixnums *x* and *y* under the control of *k*. *k* must be a fixnum between 0 and 17 (octal). If the binary representation of *k* is abcd, then the truth table for the Boolean operation is:

```
        y
     | 0  1
   --+------
    0| a  c
x    |
    1| b  d
```

If boole has more than three arguments, it goes from left to right; thus

```
(boole k x y z) = (boole k (boole k x y) z)
```

The most common values for *k* are 1 (and), 7 (or), 6 (xor). You can get the complement, or logical negation, of *x* by (boole 6 *x* -1).

lsh SUBR 2 args

(lsh *x y*), where *x* and *y* are fixnums, returns *x* shifted left *y* bits if *y* is positive, or *x* shifted right $|y|$ bits if *y* is negative. 0 bits are shifted in to fill unused positions. The result is undefined if $|y| > 36$. Examples:

```
(lsh 4 1) => 10     (octal)
(lsh 14 -2) => 3
(lsh -1 1) => -2
```

rot SUBR 2 args

(rot *x y*) returns as a fixnum the 36-bit representation of *x*, rotated left *y* bits if *y* is positive, or rotated right $|y|$ bits if *y* is negative. *x* and *y* must be fixnums. The results are undefined if $|y| > 36$. Examples:

```
(rot 1 2) => 4
(rot -1 7) => -1
(rot 601234 36.) => 601234
(rot 1 -2) => 200000000000 (octal)
```

8 - Character Manipulation

8.1 - Character Objects

An atomic symbol with less than two characters in its pname is often called a "character object" and used to represent an ascii character. The atomic symbol with a zero-length pname represents the ascii null character, and the symbols with 1-character pnames represent the character which is their pname. Functions which take a character object as an argument usually also accept a string one character long or a fixnum equal to the ascii-code value for the character. Character objects are always interned on the obarray (see section 6.3), so they may be compared with the function eq.

`ascii` SUBR 1 arg

> `(ascii x)`, where x is a number, returns the character object for the ascii code x.
> Examples:
> `(ascii 101) => A`
> `(ascii 56) => /.`

`maknam` SUBR 1 arg

> `maknam` takes as its argument a list of characters and returns an uninterned atom whose pname is determined by the list of characters. The characters may be represented either as fixnums (ascii codes) or as character objects. Example:
>
> `(maknam '(a b 60 d)) => ab0d`

`implode` SUBR 1 arg

> `implode` is the same as `maknam` except that the resulting atom is interned.

explode SUBR 1 arg

(explode *x*) returns a list of characters, which are the characters that would have been typed out if (prin1 *x*) was done, including slashes for special characters but not including extra newlines inserted to prevent characters from running off the right margin. Each character is represented by a character object.
Example:

```
(explode '(+ 1/2 3)) => ( /( + /  // /1  /2 /  /3 /) )
  ;;Note the presence of slashified spaces in this list.
```

explodec SUBR 1 arg

(explodec *x*) returns a list of characters which are the characters that would have been typed out if (princ *x*) was done, not including extra newlines inserted to prevent characters from running off the right margin. Special characters are not slashified. Each character is represented by a character object.
Example:

```
(explodec '(+  1x 3)) => ( /( + /  /1 x /  /3 /) )
```

exploden SUBR 1 arg

(exploden *x*) returns a list of characters which are the characters that would have been typed out if (princ *x*) was done, not including extra newlines inserted to prevent lines characters from running off the right margin. Special characters are not slashified. Each character is represented by a number which is the ascii code for that character. cf. explodec. Example:

```
(exploden '(+ 1x 3)) => (50 53 40 61 170 40 63 51)
```

flatc SUBR 1 arg

flatc returns the length of its argument in characters, if it was printed out without slashifying special characters. (flatc *x*) is the same as (length (explodec *x*)).

flatsize SUBR 1 arg

flatsize returns the length of its argument in characters, if it was printed out with special characters slashified. (flatsize *x*) is the same as (length (explode *x*)).

`readlist` SUBR 1 arg

The argument to `readlist` is a list of characters. The characters may be represented either as fixnums (ascii codes) or as character objects. The characters in the list are assembled into an S-expression as if they had been typed into `read` (See chapter 13 for a description of `read`.) If macro characters are used, any calls to `read`, `readch`, `tyi`, or `tyipeek` in the macro character functions take their input from `readlists`'s argument rather than from an I/O device or a file.

Examples:

```
    (readlist '(a b c)) => abc
    (readlist '( /( p r 151 n t /  /' f o o /) ))
          => (print (quote foo))
```

Note the use of the slashified special characters left parenthesis, space, quote, right parenthesis in the argument to `readlist`.

8.2 - Functions on Strings

These character string functions only exist at present in the Multics implementation of MACLISP. A predicate to test if your implementation has these functions is

```
(status feature strings)
```

These functions all accept atomic symbols in place of strings as arguments; in this case the pname of the atomic symbol is used as the string. When the value of one of these functions is described as a string, it is always a string and never an atomic symbol.

`catenate` LSUBR 0 or more args

The arguments are character strings. The result is a string which is all the arguments concatenated together. Example:

```
(catenate "abc" "-" "bar") => "abc-bar"
```

`index` SUBR 2 args

`index` is like the PL/I builtin function index. The arguments are character strings. The position of the first occurrence of the second argument in the first is returned, or 0 if there is none. Examples:

```
(index "foobar" "ba") => 4
(index "foobar" "baz") => 0
(index "goobababa" "bab") => 4
```

`stringlength` SUBR 1 arg

The argument to `stringlength` must be a character string. The number of characters in it is returned. Examples:

```
(stringlength "foo") => 3
(stringlength "") => 0
```

`substr` LSUBR 2 or 3 args

This is like the PL/I substr builtin. `(substr` *x m n*`)` returns a string *n* characters long, which is a portion of the string *x* beginning with its *m*'th character and proceeding for *n* characters. *m* and *n* must be fixnums, *x* must be a string.

`(substr` *x m*`)` returns the portion of the string *x* beginning with its

m'th character and continuing until the end of the string.
Examples:

```
(substr "foobar" 3 2) => "ob"
(substr "resultmunger" 6) => "tmunger"
```

get_pname SUBR 1 arg

(get_pname x) returns the pname of x as a character string. x must be an atomic symbol.

make_atom SUBR 1 arg

make_atom returns an atomic symbol, uninterned, whose pname is given as a character string argument. Example:

```
(make_atom "foo") => foo     ;which is not eq to a
                             ;foo that is read in.
```

CtoI SUBR 1 arg

CtoI returns as a fixnum the ascii code for the first character of its argument, which must be a string.
Example: (CtoI "z") => 172

ItoC SUBR 1 arg

ItoC returns a string one character long, consisting of the character whose ascii code is the argument.
Example: (ItoC 101) => "A"

This page intentionally left blank.

9 - Functions Concerning Arrays

MACLISP provides arrays of any number of dimensions. The contents of the arrays can be any LISP objects. The different elements of an array need not be of the same type.

An array is implemented as a space in which to keep the contents of the array and a function to access it. The name of the function is the name of the array. The arguments to the function are the subscripts. LISP arrays are always 0-origin indexed, that is, the subscript values start at 0. The subscripts must be fixnums. The array-accessing function returns as its value the contents of the selected array cell and as a side-effect saves a pointer to this cell for the use of the store function (see below). The functional property of an array-accessing function is kept under the indicator array.

There is a special type of array called an "un garbage collected" array. If an object other than an atomic symbol is only referenced from an element of a normal array, that object will stay around, but if an object is only referenced by an element of an un garbage collected array then the object will be taken away by the garbage collector and the element of the un garbage collected array will be changed to something random. In other words, un garbage collected arrays do not protect the objects they contain.

Some implementations of MACLISP also have "number arrays," which can only hold either fixnums or flonums (but not both). They are more efficient for this purpose than regular arrays. The construction and use of number arrays is the same as described below, except that a special flag is specified when a number array is initially constructed. See the *array function described below.

Here is an example of a use of arrays:

```
        (array x t 500.)
        (array y t 500.)    ;define 2 arrays
        (do i 0 (1+ i) (= i (cadr (arraydims 'x)))
          (store (x i) (//$ (float i) 100.0))
          (store (y i) (sin (x i)))
         )        ;end loop
        (plot 'x 'y)        ;call a plotting routine
                            ;for example, the following:

    (defun plot (xarr yarr)
        (cursorpos 'C)       ;erase the display
        (do i 0 (1+ i) (= i (cadr (arraydims xarr)))
          (cursorpos
            (yarr i)
            (xarr i))        ;move plotting device to
          (princ '/.)))      ;position and put a dot there.
```

`*array` LSUBR 3 or more args

(`*array` *x y b1 b2 ... bn*) defines *x* to be an *n*-dimensional array. The first subscript may range from 0 to *b*1-1, the second from 0 to *b*2-1, etc. If *y* is `t` a normal array is created; if *y* is `nil` an "un garbage collected" array is created. If *y* is the atom `fixnum` or the atom `flonum`, then a number array would be created.

`array` FSUBR

(`array` *x y b1 b2 ... bn*) is like (`*array` *x y b1 b2 ... bn*) except that *x*, the name of the array, and *y*, the type of array, are not evaluated. The other arguments are evaluated.

`*rearray` LSUBR 1 or more args

`*rearray` is used to redefine the dimensions of an array.

(`*rearray` *x*) gets rid of the array *x*. *x* is evaluated - it must evaluate to an atomic symbol. The value is `t` if it was an array, `nil` if it was not.

(`*rearray x type dim1 dim2 ... dimn`) is like (`*array x type dim1 dim2 ... dimn`) except that the contents of the previously existing array named `x` are copied into the new array named x.

`store` FSUBR

The first argument to `store` must be a subscripted reference to an array. The second argument is evaluated and stored into the referenced cell of the array. `store` evaluates its second argument *before* its first argument.

Examples:

```
(store (data i j) (plus i j))

(store (sine-values (fix (*$ x 100.0)))
       (sin x))
```

`arraydims` SUBR 1 arg

(`arraydims` *x*), where *x* evaluates to an atomic symbol which is an array name, returns a list of the type and bounds of the array. Thus if A was defined by (`array A t 10 20`),

```
(arraydims 'A) => (t 10 20)
```

bltarray SUBR 2 args

bltarray is used to copy one array into another.

(bltarray *x y*) moves the contents of the array *x* into the contents of the array *y*. If *x* is bigger than *y*, the extra elements are ignored. If *x* is smaller than *y*, the rest of *y* is unchanged. *x* and *y* must evaluate to atomic symbols which have array properties.

fillarray SUBR 2 args

(fillarray *A L*) fills the array *A* with consecutive items from the list *L*. If the array is too short to contain all the items in the list, the extra items are ignored. If the list is too short to fill up the array, the last element of the list is used to fill each of the remaining slots in the array. fillarray could have been defined by:

```
(defun fillarray (a x)
    (do ((x x (cond ((cdr x))
                    (x)))
         (n 0 (1+ n))
         (hbound (cadr (arraydims a))))
        ((= n hbound))
      (store (a n) x)
     ))
```

An extension to the LISP definition is that fillarray will work with arrays of more than one dimension, filling the array in row-major order. fillarray returns its first argument.

listarray SUBR 1 arg

(listarray *array-name*) takes the elements of the array specified by *array-name* and returns them as the elements of a list. The length of the list is the size of the array and the elements are present in the list in the same order as they are stored in the array, starting with the zeroth element. If the array has more than one dimension row-major order is used.

This page intentionally left blank.

10 - "Mapping" Functions

Mapping is a type of iteration in which a function is successively applied to pieces of a list. There are several options for the way in which the pieces of the list are chosen and for what is done with the results returned by the applications of the function.

For example, mapcar operates on successive elements of the list. As it goes down the list, it calls the function giving it an element of the list as its one argument: first the car, then the cadr, then the caddr, etc. continuing until the end of the list is reached. The value returned by mapcar is a list of the results of the successive calls to the function.

An example of the use of mapcar would be mapcar'ing the function abs over the list (1 -2 -4.5 6.0e15 -4.2). The result is (1 2 4.5 6.0e15 4.2).

The form of a call to mapcar is

(mapcar *f* *x*)

where *f* is the function to be mapped and *x* is the list over which it is to be mapped. Thus the example given above would be written as

```
(mapcar 'abs
        '(1 -2 -4.5 6.0e15 -4.2))
```

This has been generalized to allow a form such as

(mapcar *f* *x1* *x2* ... *xn*)

In this case *f* must be a function of *n* arguments. mapcar will proceed down the lists *x1*, *x2*, ..., *xn* in parallel. The first argument to *f* will come from *x1*, the second from *x2*, etc. The iteration stops as soon as any of the lists becomes exhausted.

There are five other mapping functions besides mapcar. maplist is like mapcar except that the function is applied to the list and successive cdr's of that list rather than to successive elements of the list. map and mapc are like maplist and mapcar respectively except that the return value is the first of the lists being mapped over and the results of the function are ignored. mapcan and mapcon are like mapcar and maplist respectively except that they combine the results of the function using nconc instead of list.

Sometimes a do or a straight recursion is preferable to a map; however, the mapping functions should be used wherever they naturally apply because this increases the clarity of the code.

Often *f* will be a lambda-type function rather than the atomic-symbol name of a function. For example,

```
(mapcar '(lambda (x) (cons x something)) some-list)
```

The functional argument to a mapping function must be acceptable to apply: it cannot be a macro. A fexpr or an fsubr may be acceptable however the results will be bizarre. For instance, mapping set works better than mapping setq, and mapping cond is unlikely to be useful.

It is permissible (and often useful) to break out of a map by use of a go, return, or throw in a lambda-type function being mapped. This is a relaxation of the usual prohibition against "non-local" go's and return's. Consider this function which is similar to and, except that it works on a list.

```
(defun andl (x)
    (catch
       (progn
          (mapc (function (lambda (y)
                               (or y (throw nil the-answer)) ))
                x)
          t)
       the-answer))
```

Here is a table showing the relations between the six map functions.

```
                              applies function to

                          | successive  |  successive  |
                          |  sublists   |   elements   |
           ---------------+-------------+--------------+
               its own    |             |              |
               second     |    map      |     mapc     |
               argument   |             |              |
           ---------------+-------------+--------------+
              list of the |             |              |
returns        function   |   maplist   |    mapcar    |
               results    |             |              |
           ---------------+-------------+--------------+
             nconc of the |             |              |
               function   |   mapcon    |    mapcan    |
               results    |             |              |
           ---------------+-------------+--------------+
```

map LSUBR 2 or more args

The first argument to map is a function, and the remaining arguments are lists. map "goes down" the lists, applying the function to the lists each time. The value returned by map is its second argument. map stops as soon as one of the lists is exhausted. Example:

```
(map '(lambda (x y z) (print (list x y z)))
     '(1 2 3 4) '(a b c d e) '(+ - * |))
```

prints

```
((1 2 3 4) (a b c d e) (+ - * |))
((2 3 4) (b c d e) (- * |))
((3 4) (c d e) (* |))
((4) (d e) (|))
```

and returns (1 2 3 4).

mapc LSUBR 2 or more args

mapc is just like map except that the function is applied to successive elements of the lists rather than to the lists themselves. Thus the example given under map would print

```
(1 a +)
(2 b -)
(3 c *)
(4 d |)
```

and return (1 2 3 4)

mapcar LSUBR 2 or more args

mapcar is like mapc except that the return value is a list of the results of each application of the function. Thus the example given with mapc would return, not (1 2 3 4), but

```
((1 a +) (2 b -) (3 c *) (4 d |))
```

maplist LSUBR 2 or more args

maplist is like map except that the return value is a list of the results of each application of the function. Thus the example given with map would return, not (1 2 3 4), but

```
(((1 2 3 4) (a b c d e) (+ - * |)) ((2 3 4) (b c d e) (- * |)) ((3 4)
(c d e) (* |)) ((4) (d e) (|)))
```

mapcan LSUBR 2 or more args

mapcan is like mapcar except that the values returned by the function are nconc'ed together instead of being list'ed together. Thus the example would return
(1 a + 2 b - 3 c * 4 d |)

mapcon LSUBR 2 or more args

mapcon is like maplist except that the values returned by the function are nconc'ed together instead of being list'ed together. This can have disastrous effects on the arguments to mapcon if one is not careful. The example would return
((1 2 3 4) (a b c d e) (+ - * |) (2 3 4) (b c d e) (- * |)
(3 4) (c d e) (* |) (4) (d e) (|))

11 - Sorting Functions

Several functions are provided for sorting arrays and lists. These functions use algorithms which always terminate no matter what sorting predicate is used, provided only that the predicate always terminates. These sorts are not necessarily stable, that is equal items may not stay in their original order.

After sorting, the argument (be it list or array) is rearranged internally so as to be completely ordered. In the case of an array argument, this is accomplished by permuting the elements of the array, while in the list case, the list is reordered by rplacd's in the same manner as nreverse. Thus if the argument should not be clobbered, the user must sort a copy of the argument, obtainable by bltarray or append, as appropriate.

Should the comparison predicate cause an error, such as a wrong type argument error, the state of the list or array being sorted is undefined. However, if the error is corrected the sort will, of course, proceed correctly.

Both sort and sortcar handle the case in which their second argument is the function alphalessp in a more efficient manner than usual. This efficiency is primarily due to elimination of argument checks at comparison time.

sort SUBR 2 args

The first argument to sort is an array (or list), the second a predicate of two arguments. Note that a "number array" cannot be sorted. The predicate must be applicable to all the objects in the array or list. The predicate should take two arguments, and return non-nil if and only if the first argument is strictly less than the second (in some appropriate sense).

The sort function proceeds to sort the contents of the array or list under the ordering imposed by the predicate, and returns the array or list modified into sorted order, i.e. its modified first argument. Note that since sorting requires many comparisons, and thus many calls to the predicate, sorting will be much faster if the predicate is a compiled function rather than interpreted.

Example:

```
(defun mostcar (x)
    (cond ((atom x) x)
          ((mostcar (car x)))))

(sort 'fooarray
      (function (lambda (x y)
          (alphalessp (mostcar x) (mostcar y)))))
```

If fooarray contained these items before the sort:

```
(tokens (the lion sleeps tonight))
(carpenters (close to you))
((rolling stones) (brown sugar))
((beach boys) (i get around))
(beatles (i want to hold your hand))
```

then after the sort fooarray would contain:

```
((beach boys) (i get around))
(beatles (i want to hold your hand))
(carpenters (close to you))
((rolling stones) (brown sugar))
(tokens (the lion sleeps tonight))
```

sortcar SUBR 2 args

sortcar is exactly like sort, but the items in the array or list being sorted should all be non-atomic. sortcar takes the car of each item before handing two items to the predicate. Thus sortcar is to sort as mapcar is to maplist.

12 - Functions for Controlling the Interpreter

12.1 - The Top Level Function

When LISP is at its "top level," it continually evaluates the following form.

```
(errset (setq ^r nil ^q nil ... )  ;reset internal variables
        (mapc 'eval errlist)  ;(come in here on error)
        (setq * '*)
        (do nil (nil)         ;then do the following forever
          (setq * (cond ((status toplevel)
                         (eval (status toplevel)));user's
                        (t (print *)     ;system's,
                           (terpri)      ;print prev. result
                           (eval (read))) ;read and eval next.
                   ))))
```

which causes a "read-eval-print loop," i.e. each S-expression that is typed in gets evaluated and the value is printed, then the next S-expression is read. Notice that there is a place in the middle where the user can insert his own special form to be evaluated, using `(status toplevel)`. See the `sstatus` function (section 12.7).

When the LISP subsystem is entered, it is at top level. At this time, a * is typed out and LISP begins continually evaluating the top level form. LISP can return to top level from the middle of one of these evaluations when an error occurs. Generally errors do not immediately return to top level; rather they give the programmer a chance to find the cause of the error. However, if an error is not corrected it will eventually get to top level unless there is an `errset` in the way. When LISP returns to its top level, it again types a * and begins continuously evaluating the top level form.

`errlist` VARIABLE

The value of `errlist` is a list of forms which are evaluated when control returns to top level either because of an error or when an environment is initially started. This feature is used to provide self-starting LISP environments and to provide special error handling for subsystems written in LISP.

* VARIABLE

The value of * is the result of the last evaluation performed at top level or in a break loop. When the lisp environment is first entered and when control is returned to top level from an error, the value of * is the atomic symbol * itself.

12.2 - Break Points

break FSUBR

(break *tag pred*) evaluates *pred*, but not *tag*. If the value of *pred* is not nil, the state of the I/O system is saved, ";bkpt *tag*" is typed out, and control returns to the terminal. We say that a "break loop" has been entered. *tag* may be any object. It is used only as a message typed out to identify the break. It is not evaluated.

Forms may be typed in and evaluated as at top level. break does an errset so that errors cannot cause an abnormal return from the break.

If $p is typed in, break returns nil. This "$p" is <dollar> p <newline> in the Multics implementation, but <altmode> P <space> in the pdp-10 implementations.

If (return *x*) is typed in, break evaluates *x* and returns that value. When break returns, the state of the I/O system is restored.

break can be used to allow user intervention in a program when something unexpected happens. It is used in this way by the LISP error system.

(break *tag pred form*) is the same as (break *tag pred*) except that if $p is typed, the *form* is evaluated and used as the value of the break instead of nil.

12.3 - Control Characters

LISP can be directed to take certain actions by entering "control characters" from the terminal. The difference between control characters and normal input is that control characters take effect as soon as they are entered while normal input only takes effect when LISP asks for it, by use of functions such as read, or by being in the top level read-eval-print loop or in a break loop.

Control characters can be typed in from the terminal according to some procedure that depends on the implementation. A program can mimic the effects of the various control characters using the function ioc.

Although control characters are usually processed as soon as they are typed, they will be delayed if there is a garbage collection in progress or LISP is in (nointerrupt t) mode - see the nointerrupt function.

Entering Control Characters in ITS LISP

In the ITS implementation of MACLISP, control characters are entered by means of the "CTRL" key on the terminal. For example, CTRL/G is entered by holding down "CTRL" and striking the "G" key. Control characters echo as an uparrow or circumflex followed by the character. Numbers may not be used as control characters.

Entering Control Characters in DEC-10 LISP

Control characters may be entered in the same way as in ITS LISP if LISP is currently (read)'ing from the terminal. If a LISP program is actively running, it is necessary to first gain its attention by striking control-C. LISP rings the bell and types a ? and an up-arrow, prompting for the entry of a character to be treated as a control character. If you type a second control-C control will return to the monitor.

Entering Control Characters in Multics LISP

In the Multics implementation of MACLISP, one signals one's desire to enter a "control" character by hitting the "attention" key on the terminal. This is called "break," "interrupt," "attn", "quit," etc. on different terminals. If Multics is being accessed through the ARPA network, an "interrupt process" signal should be transmitted. This causes lisp to type out "CTRL/" After this has been typed, you may type one control character, which is a letter from the list in section 12.3.3 which will be interpreted to have its "control" meaning. The control character must be followed by a newline.

You may also enter a number, which will be interpreted in decimal. A user interrupt will be issued on the user interrupt channel indicated by the number you typed in. If there is no such interrupt, or the interrupt service function for that channel is nil, *no* warning will be issued. The argument passed to the interrupt service function is the atom `ioc`. The most useful interrupt channel numbers are 0, which is the same as CTRL/@; 1, which is the same as CTRL/h, and 2 which is the same as CTRL/a (exactly, including the changing of the value of the atom `^a`). Note that certain characters, notably @, may have special meaning to the Multics typewriter DIM and may have to preceded by a backslash.

Note: any input that has been typed in but has not yet been read by lisp when the attention key is pushed will be lost. Usually this is the current line of input.

It is also possible to enter "control" characters from an input character stream, which may have its source at the terminal or in an exec_com, without the use of the "attention" key. The desired control character is prefixed by a \036 character. If two of these prefix characters occur together, one \036 character is read and no "control" action is performed. Otherwise, the character following the \036 is processed as a control character, then reading continues. This method only works with the letter and special-symbol control characters, not with the number control characters.

NB: Control characters will be accepted in upper or lower case. All characters other than those with defined meanings are rejected with an error message. Only one control character may be entered at a time. When a "user interrupt" is caused, if the interrupt is not enabled nothing happens. If the interrupt is enabled, then a user-specified function is called. The interrupt may be enabled by using the function `sstatus`. *E.g.:* `(sstatus interrupt 2 'foo)` causes `foo` to be called with one argument when interrupt 2 occurs.

Example: (lines containing user input are preceded by >>>)

```
>>>     (defun loop (x) (loop (add1 x)))
        loop
>>>     (loop 0)
                   function runs for a long time,
>>>     <ATTN>     then user hits attention button.
>>>     CTRL/B     LISP types "CTRL/", user types "B"
>>>     ;bkpt ^b   system enters break loop
>>>     x          user looks at value of x
        4067
>>>     <ATTN>     user hits attention button again
>>>     CTRL/G     and returns to top level
        Quit
        *
```

12.3.1 - List of Control Characters

These are the control characters that have defined meanings

A makes the value of the atom ^a non-nil and causes user interrupt 2.

B causes user interrupt number 1, which (usually) enters the "bkpt ^b" breakpoint.

C sets the value of the atom ^d to nil, turning off garbage collector messages

D sets the value of the atom ^d to t, turning on garbage collector messages

G quits back to top level of lisp, undoing all bindings

Q sets the value of the atom ^q to t, enabling input from the source selected by the value of infile, or by use of the function uread.

R sets the value of the atom ^r to t, enabling output to the destinations selected by the value of outfiles, or by use of the uwrite function.

S sets the value of the atom ^q to nil, enabling input from the terminal

T sets the value of the atom ^r to nil, disabling output to the destinations that CTRL/r enables.

U causes the current call to (read) to be restarted from the beginning.

V sets the value of the atom ^w to nil, enabling output to the terminal

W sets the value of the atom ^w to t, disabling output to the terminal

X causes an error which can be caught by errset

Z On the pdp-10 returns to DDT. On Multics returns to Multics command level. (start re-enters lisp.) This control character is handled immediately even when LISP is garbage collecting or running in (nointerrupt t) mode, unlike most of the others.

@ causes user interrupt 0. Note that on Multics an escape must be used to type the @ sign.

\ causes user interrupt 14. Note that on Multics an escape must be used to type this backslash character.

] causes user interrupt 15.

^ causes user interrupt 16.

The following control characters only exist in the Multics implementation.

. does nothing, used to speed up a slow process by causing an interaction. This control character is handled immediately even when LISP is garbage collecting or running in `(nointerrupt t)` mode, unlike most of the others.

? asks the LISP subsystem what it is doing: running, waiting for input, collecting garbage, or running with user-interrupts masked off. This control character is handled immediately even when LISP is garbage collecting or running in `(nointerrupt t)` mode, unlike most of the others.

The following control characters only exist in pdp-10 implementations with the "moby I/O" capability.

F cause display slave to seize a display.

N turn on display.

O turn off display.

Y interrogate display slave.

The following control characters only work in the pdp-10 implementation.

K redisplay the current input. This allows you to get a clean copy of your input after rubouts have been used.

L erases the screen if the terminal is a display, then does a control K.

U if the terminal is a display, and in `(sstatus pagepause t)` mode, and the end of the screen has been reached, typing control u will tell lisp to continue typing out.

12.3.2 - Control-Character Functions

ioc FSUBR

The argument to ioc is processed as if it were a "control character" that had been typed in. Numbers are taken as a whole, pname atoms (atomic symbols) are processed character by character, except that nil causes an immediate return. Examples:

(ioc 1) causes user interrupt 1.
(ioc vt) switches output to the terminal.
(ioc q) switches input to a file.
(ioc g) quits back to the top level of lisp.

If ioc returns, its value is t.

iog FSUBR

iog first saves the values of the I/O switches ^q, ^r, and ^w. Then it processes its first argument the same as ioc. Next the remaining arguments to iog are evaluated, from left to right. The values of the variables ^q, ^r, and ^w are restored, and the value of the last argument is returned. Example:

```
(iog vt (princ "A Message."))
```

gets a message to the console no matter what the I/O system is doing. It evaluates to "A Message."

^a VARIABLE

When a CTRL/a is done, the value of the atom ^a is made non-nil (user interrupt 2 is also signalled.)

12.4 - Errors and User Interrupts

12.4.1 - The LISP Error System

The errors detected by the LISP subsystem are divided into two types: correctable and uncorrectable. The uncorrectable errors will be explained first since they are simpler.

An uncorrectable error is an error that causes the destruction of the evaluation in which it occurs. An example of an uncorrectable error is illegal format in a 'do'. When an uncorrectable error occurs, the first thing that happens is the printing of an error message. The error message goes to the terminal and nowhere else, no matter how the I/O switches and variables are set. The error message consists of some explanatory text and (sometimes) the object or form that caused the error.

After the error message has been printed, control is returned to the most recent error-catcher. There is an error-catcher at top level, and error-catchers are set up by the functions errset and break. All variable bindings between the error-catcher and the point where the error occurred are restored. Thus all variables are restored to the values they had at top level or at the time the errset was done, unless they were setq'ed without being bound.

What happens next depends on how the error-catcher was set up. At top level, (mapc 'eval errlist) is done, a * is typed, and the read-eval-print loop (or a user specified top level form) is re-entered. If an error returns to break, it simply re-enters its read-eval-print loop. In the Multics implementation the fact that break has caught an error is signalled by doing something to the terminal such as blinking a light, ringing a bell, or twiddling the typeball, depending on the type of terminal. If an error returns to errset, errset returns nil and evaluation proceeds.

The above description is slightly simplified. It is possible for a user interrupt to occur between the typing of the message and the unwinding of bindings and return of control to an error-catcher. This user interrupt is normally a break loop which allows the user to examine the values of variables before the bindings are restored, in hope of finding the cause of the error. If the error is going to return to top level, the *rset-trap user interrupt (number 19.) is signalled. In (*rset t) mode a break loop is entered, but in (*rset nil) mode the user interrupt is ignored by the system supplied handler. If the error is going to return to a break or an errset, the errset user interrupt (number 4) is signalled. The initial environment contains a null handler for this interrupt, but the user may supply a break loop or other handler.

Correctable errors are errors which may be corrected by user

intervention. If such an error is properly corrected, evaluation will proceed as if no error had occurred. If the option to correct the error is not exercised, this type of error will be handled the same as an uncorrectable error.

When a correctable error occurs, a user interrupt is signalled. See section 12.4.2 for user interrupt channel assignments for these errors. The initial environment contains handlers for these errors which print an error message similar to the message printed for an uncorrectable error and then enter a break loop.

The argument passed to the user interrupt handler is usually a list describing the error. See section 12.4.2 for details. If the user interrupt handler is `nil`, or if it returns a non-list, the error is treated like an uncorrectable error. But if the handler returns a list, the car of that list is used to correct the error in a way which depends on the particular error which occurred.

If the most recent error-catcher is an `errset` (or a `break`), correctable errors will be treated as uncorrectable errors unless there is a non-null handler for user interrupt 4, the `errset` interrupt. This is to prevent multiple confusing "nested" error breaks unless the user indicates that he is sophisticated by setting up a handler for the `errset` interrupt.

12.4.2 - User Interrupts

LISP provides a number of "user interrupts," which are a mechanism by which a user procedure may temporarily gain control when an exceptional condition happens. The exceptional conditions that use the user interrupt system include certain control characters, the alarmclock timers, the garbage collector, and many of the errors that are detected by the interpreter or by the system functions.

The user interrupts are divided up into several channels. Each channel is designated by a number. Each channel has associated with it a "service function." If the service function is `nil`, interrupts on that channel will be ignored. If the service function is not `nil`, it is a function which is called with one argument when the user-interrupt occurs. The nature of the argument depends on which channel the interrupt is on; usually it is an S-expression which can be used to localize the cause of the interrupt. Some user interrupts use the value returned by the service function to decide what to do about the cause of the interrupt.

The service function for user interrupt channel *n* can be obtained by `(status interrupt` *n*`)`. It can be set by `(sstatus interrupt` *n f*`)`. The initial values for the service functions of the various interrupts are provided by the system as break loops for some interrupt channels and `nil` for others.

Some user interrupt channels keep their service functions as the values of variables accessible to the user; this allows them to be lambda-bound. See section 12.4.3.

The interrupt channels with entries in the return value column of the table of user interrupts included in this section are user interrupts signalled by correctable error conditions. The argument to the service function is a description of the error in the form shown. If the service function returns `nil` (or any atom), the normal error procedure occurs -- control returns to the most recent `errset` or to top level if there was no `errset`. If the service function returns a list, it is interpreted as the form shown in the return value column. The car of the list is used to attempt recovery from the error. Note that interrupts 6 and 9. evaluate this before they use it. In the table, an apostrophe is used to indicate that the new value for user interrupt 6 will be evaluated unless you quote it. Often you would give a substitute atomic symbol which would then be evaluated to get the new value. If recovery is successful execution proceeds from the point where the error occurred. If recovery is unsuccessful another error is signalled.

Table of User Interrupt Channels

Chn Num	Symbol Whose Value is Fcn.	Reason for Interrupt	Argument passed to function	Return Value
0	-none-	ctrl @	nil	-
1	^b	ctrl b	nil	-
2	-none-	ctrl a	nil	-
3	alarmclock	a timer went off	time or runtime	-
4	errset	error caught by errset	nil	-
5	undf-fnctn	undefined function	(fcn)	(new-fcn)
6	unbnd-vrbl	undefined symbol	(symb)	('new-val)
7	wrng-type-arg	bad arg to a fcn.	(arg)	(new-arg)
8.	unseen-go-tag	go or throw error	(bad-tag)	(new-tag)
9.	wrng-no-args	wrong number of args	(form args-prop) or (form lambda-list)	(new-form)
10.	gc-lossage	out of memory	space-name	-
11.	fail-act	miscellaneous error	see below	see below
12.	pdl-overflow	infinite recursion	pdl-name	(t)
13.	gc-overflow	a space is filled	space-name	(t)
14.	-none-	ctrl \	nil	-
15.	-none-	ctrl]	nil	-
16.	-none-	ctrl ^	nil	-
18.	-none-	autoload	(fcn . property)	-
19.	*rset-trap	error return to top level	nil	-
20.	gc-daemon	garbage collection	see below	-

The `fail-act` interrupt, number 11., is used for a variety of miscellaneous error conditions. Here is a table giving the types of

arguments that may be passed to the fail-act service function. For each type the cause of the interrupt and the return value to correct the error are given.

(arrayindex (*name indices...*)) An out-of-bounds array access occurred on the array *name*. Sometimes it is not possible to determine the name of the array, in which case *name* will be ?. For example, this can happen if the array has been remob'ed. The return form is ((*name subs...*)) which will retry the access with the new subscripts *subs...* *name* is ignored.

(go return) go or return was used outside of a prog. The return value does not matter; this error is not actually correctable.

(arg (*n*)) The arg function was called with argument *n*, but this was not done inside a lexpr.

(setarg (*n value*)) The function setarg was called with arguments *n* and *value* and an error similar to the preceding occurred.

(ibase) The reader variable ibase had a bad value: not a fixnum or not between 2 and 36. It is reset to 8. before the user interrupt occurs. Returning (t) will cause the reader to continue reading.

(base) The printer variable base had a bad value: not a fixnum or not between 2 and 36. It is reset to 8. before the user interrupt occurs. Returning (t) will cause the printer to continue.

(infile *x*) The input file *x* is invalid for one reason or another. ^q has been reset to nil. If (t) is returned by the service function, ^q will be set back to t and the function that lost trying to input from *x* will proceed, taking input from infile.

(setq (nil)) You aren't allowed to change the value of nil.

(setq (t)) You aren't allowed to change the value of t.

(read-eof) The reader found an end of file in the middle of an object. Usually this indicates mismatched parentheses. If (t) is returned, it will go on reading the broken object from whatever input source is now selected.

(outfile *x*) The output file *x* is invalid for one reason or another. ^r is reset to nil before the user interrupt occurs. If (t) is returned, ^r will be set back to t and the function that lost trying to output to *x* will go on its way.

(filepos *x*) The filepos function was used on the file *x*, but this file is not equipped for random access. If ('*new-val*) is returned, filepos returns *new-val* to its caller without further ado.

(filepos *x n*) The filepos function was used in the form (filepos *x n*), but *n* is not a position inside the file *x*. If ('*new-val*) is returned,

filepos returns *new-val* to its caller without further ado.

(openi *x*), (openo *x*), (opena *x*), (rename *x y*), (deletef *x*) The file system complained in some way: e.g. file not found, incorrect access. The argument passed to the service function represents the form which was evaluated to cause the error, except that the values of the arguments are the values they have after being mergef'ed over the defaults, i.e. they are precise namelists. If ('*new-val*) is returned, the function that lost returns *new-val* as its value without further ado.

(zunderflow) A floating-point arithmetic underflow has occurred, i.e. a result was developed whose magnitude was too small to be represented by the machine. Setting the variable zunderflow to t would have prevented the fail-act from happening; instead the result would have been taken as 0.0. Returning a non-atomic value from the fail-act will cause this to happen.

(quotient 0) An attempt was made to divide a number by zero. If (sstatus divov t) has been evaluated, the fail-act would not have occurred. Instead the result of the division would have been taken as the numerator + 1. Returning a non-atomic value from the fail-act causes this to occur.

Here is an example of a user interrupt service function. This is the one supplied by the system for unbound variable errors when the user does not specify one.

```
(setq unbnd-vrbl
  '(lambda (args)
     (iog vt (errprint nil))
     ((lambda (readtable obarray)
          (break unbnd-vrbl t))
        (get 'readtable 'array)
        (get 'obarray 'array) )))
```

12.4.3 - User Interrupt Functions and Variables

alarmclock SUBR 2 args

alarmclock is a function for controlling timers. It can start and stop two seperate timers; one is a real-time timer (which counts seconds of elapsed time) and the other is a cpu-time timer (which counts microseconds of machine run time). The first argument to alarmclock indicates which timer is being referred to: it may be the atom time to indicate the real-time timer or the atom runtime to indicate the cpu-time timer.

The second argument to alarmclock controls what is done to the selected timer. If it is a positive (non-big) number the timer is started. Thus if n is a positive fixnum or flonum, evaluating (alarmclock 'time n) sets the real-time timer to go off in n seconds, and (alarmclock 'runtime n) sets the cpu-time timer to go off in n microseconds. If the timer was already running the old setting is lost. Thus at any given time each timer can only be running for one alarm, but the two timers can run simultaneously.

If the second argument to alarmclock is not a positive number, the timer is shut off, so (alarmclock *x* nil) or (alarmclock *x* -1) shuts off the *x* timer.

alarmclock returns t if it starts a timer, nil if it shuts it off.

When a timer goes off, user interrupt 3 occurs. The service function is run in (nointerrupt t) mode so that it can not be interrupted until it has done its thing. If it wants to allow interrupts, other timers, etc. it can evaluate (nointerrupt nil). In any case the status of the nointerrupt flag will be restored when the service function returns. The argument passed to the user interrupt service function is a list of one element, the atom time or the atom runtime, depending on the first argument in the call to alarmclock that set up the timer. See also the function nointerrupt.

alarmclock VARIABLE

The value of alarmclock is the service function for user interrupt number 3, which is the user interrupt signalled when a timer set up by the alarmclock function goes off.

nointerrupt SUBR 1 arg

(nointerrupt t) shuts off LISP interrupts. This prevents alarmclock timers from going off and prevents the use of control characters such as CTRL/g and CTRL/b. Any of these interrupts that occur are simply saved. (nointerrupt t) mode is used to protect critical code in large subsystems written in LISP. It is also used by the LISP subsystem itself to protect against interrupts in the garbage collector.

(nointerrupt nil) turns interrupts back on. Any interrupts which were saved will now get processed.

nointerrupt returns the previous state of the interrupt disable switch, t or nil. The normal, initial state is nil.

errset VARIABLE

The value of the atom errset is the service function for user interrupt number 4, which is signalled when an error is caught by an errset.

fail-act VARIABLE

The value of fail-act is the service function for user interrupt number 11., which is signalled when any of a large variety of miscellaneous error conditions occurs.

gc-daemon VARIABLE

The value of gc-daemon is the service function for user interrupt 20., which is signalled after each garbage collection.

gc-lossage VARIABLE

The value of gc-lossage is the service function for user interrupt number 10., which is signalled when there is no more memory. In the Multics implementation, there is always enough memory, so this user interrupt never occurs.

unbnd-vrbl VARIABLE

The value of unbnd-vrbl is the service function for user interrupt number 6, which is signalled when an attempt is made to evaluate an atomic symbol which does not have a value (an unbound variable.)

undf-fnctn VARIABLE

The value of undf-fnctn is the service function for user interrupt number 5, which is signalled when an attempt is made to call an undefined function.

unseen-go-tag VARIABLE

The value of unseen-go-tag is the service function for user interrupt 8., which is signalled when go or throw is used with a tag which does not exist in the current prog body or in any catch, respectively.

wrng-no-args VARIABLE

The value of wrng-no-args is the service function for user interrupt channel 9., which is signalled when a function is called with the wrong number of arguments.

wrng-type-arg VARIABLE

The value of wrng-type-arg is the service function for user interrupt number 7, which is signalled when an argument is passed to a system function which is not acceptable to that function.

*rset-trap VARIABLE

The value of *rset-trap is the service function for user interrupt 19., which is signalled when an error returns control to top level, just before the bindings are restored. By convention, the handler for this interrupt should not do anything unless the variable *rset is non-nil.

12.4.4 - Autoload

The "autoload" feature provides the ability for a function not present in the environment to be automatically loaded in from a file the first time it is called. When eval, apply, funcall, or the version of apply used by compiled LISP searches the property list of an atom looking for a functional property, if the first functional property found is under the indicator autoload, automatic loading will occur.

Automatic loading is performed by means of user-interrupt 18.; thus the user may assert any desired degree of control over it. When the autoload property is encountered, the user-interrupt 18. handler is called with one argument, which is a dotted pair whose car is the atomic symbol which is the function being autoload'ed, and whose cdr is the value of the autoload property. The system-supplied handler for user interrupt 18. could have been defined by:

```
(sstatus interrupt 18.
  '(lambda (x)
      (load (cdr x)) ))
```

Note: in the pdp-10 implementations the system autoload handler presently uses fasload rather than load because the load function requires the newio feature. This affects the form of an autoload property as described below.

When the interrupt handler returns, it had better have put a functional property on the property list of the function being autoloaded. If not, an undf-fnctn error will occur with a message such as

"function undefined after autoload"

Note that the functional property must be placed on the property list before the autoload property. This is normally the case.

Examples of setting up functions to be autoloaded:

In the Multics implementation:

```
(putprop 'foo ">udd>AutoProg>Library>foo-function" 'autoload)
```

In the ITS implementation:

```
(putprop 'foo '(foo fasl dsk me) 'autoload)
```

12.5 - Debugging

12.5.1 - Binding, Pdl Pointers, and the Evaluator

The MACLISP evaluator is conceptually based on a push down list (pdl), or stack, which holds bindings, evaluation frames, and sundry internal data. Bindings are values of atomic symbols which are saved when the symbols are used as lambda variables, prog variables, or do variables. Evaluation frames are constructed when a non-atomic form is evaluated or when apply is used. They correspond to function calls.

As the evaluator recursively evaluates a form, information is pushed onto the pdl and later popped off. When the `*rset` and nouuo flags are `t` this information is sufficiently detailed to be of use in debugging. (See the variables `*rset` and nouuo.) A position within the pdl may be named by means of a "pdl pointer," which is a negative fixnum whose value has meaning to the evaluator. `nil` is also accepted as a pdl pointer. It means the top of the stack, i.e. the most recent evaluation. Note that this is different from `nil` as an a-list pointer, which means the bottom of the stack or the outermost evaluation. 0 is also accepted as a pdl pointer; it designates the frame at the bottom of the stack. Pdl pointers may be used as arguments to several debugging functions described in the next section.

Several of the debugging functions described in the next section can be used to generate pdl pointers. Since the fixnum value of a pdl pointer has only internal meaning, generally a pdl pointer cannot be obtained from user input.

An important thing to note about pdl pointers is their limited scope of validity. If the information on the pdl which is named by a pdl pointer has been popped off since the pdl pointer was created, the pdl pointer no longer has valid meaning.

12.5.2 - Functions for Debugging

See also Chapter 15, on the trace package.

Note: the functions below which return "frames" are subject to change. The exact format of the list returned may vary from time to time and implementation to implementation, so try it on your implementation and see what it does. At some time in the future this will probably be stabilized.

*rset SUBR 1 arg

(*rset *x*) sets the *rset flag to nil if *x* is nil, to t if *x* is non-nil, and returns the value it set it to. If the *rset flag is t, extra information is kept by the interpreter to allow the debugging functions, such as baktrace and evalframe, to work.

baktrace LSUBR 0 to 2 args

baktrace displays the stack of pending function calls. It only works in (*rset t) mode. The first argument is a pdl pointer, as with evalframe. If it is omitted, nil is assumed, which means start from the top of the pdl. The second argument is the maximum number of lines to be typed; if it is omitted the entire stack is displayed.

errframe SUBR 1 arg

errframe returns a list describing an error which has been stacked up because of a user interrupt. The list has the form (pdlptr message alist), where pdlptr is a number which describes the location in the pdl of the error, message is a character string which can be printed out as a description of the error, and alist is a number which can be used as a second argument to eval or a third argument to apply to cause evaluation using the bindings in effect just before the error occurred.

The argument to errframe can be nil, which means to find the error at the top of the stack; i.e. the most recent error. It can also be a pdl ptr, in which case the stack is searched downward from the indicated position. Thus the second most recent error may be found by:

```
(errframe (car (errframe nil)))
```

The argument to errframe may also be a positive number, which is the negative of a pdl ptr. This means start from the position in the stack marked by the pdl ptr and search upwards.

If no error is found, errframe returns nil.

errprint SUBR 1 arg

errprint treats its argument the same as errframe. The message portion of the error frame is princ'ed. errprint returns t if a message was typed out and nil if no error frame was found.

evalframe SUBR 1 arg

The argument to evalframe is a pdl ptr, as with errframe. The pdl is searched for an evaluation of a function call, using the same rules about starting point and direction as errframe uses. evalframe always skips over any calls to itself that it finds in the pdl.

The value is a list (pdlptr form alist), where pdlptr is a pdl pointer to the evaluation in the stack, suitable for use as an argument to evalframe or errframe or baktrace, form is the form being evaluated or the name of the function being applied, and alist is an a-list pointer which can be used with eval to evaluate something in the binding context just before the evaluation found by evalframe.

evalframe returns nil if no evaluation can be found.

evalframe only works in (*rset t) mode.

freturn SUBR 2 args

(freturn *p* *x*) returns control to the evaluation designated by the pdl pointer *p*, and forces it to return *x*. This "non-local-goto" function can be used to do fancy recovery from errors.

The following functions only exist in the Multics implementation.

baktracel LSUBR 0 to 2 args

baktracel is the same as baktrace except that a-list ptrs suitable for use with eval and apply are displayed along with the function names.

baktrace2 LSUBR 0 to 2 args

baktrace2 is the same as baktrace1 except that pdl pointers, suitable for use with baktrace and evalframe, are displayed along with the function names and a-list pointers.

12.5.3 - An Example of Debugging in Maclisp

TO BE SUPPLIED

12.6 - Storage Management

In MACLISP storage for programs and data is automatically managed by the system. The casual user need not concern himself with storage management and need not read this section. However, the user who is curious about the implementation or who has to construct a subsystem on top of MACLISP may need to be concerned with how the internal storage management routines work and how to control their general functioning. In no case is it necessary to control the exact step by step operations of storage management, but a variety of functions are provided to set the general policy followed by the lisp storage management procedures.

12.6.1 - Garbage Collection

Garbage collection is the mechanism which LISP uses to control storage allocation. Whenever LISP feels that too much storage is being used, a garbage collection is initiated. The garbage collector traces through all the S-expressions which can be reached by car'ing and cdr'ing from atomic symbols' values and property lists, from forms and temporary results currently being used by the evaluator, from data used by compiled code, and from the saved values of bound variables. All the data which it finds in this way is "good" data, in that it is possible for it to be used again. Everything else is garbage, which can never again be used for anything because it cannot be accessed, so the storage used by it is reclaimed and reused.

gc FSUBR

(gc) causes a garbage collection and returns nil.

gctwa FSUBR

gctwa is used to control the garbage collection of "truly worthless atoms," which are atomic symbols which have no value and no properties, and which are not referenced by any list structure, other than the obarray (the current obarray if there is more than one).

(gctwa) causes truly worthless atoms to be removed on the next garbage collection.

(gctwa t) causes truly worthless atoms to be removed on each garbage collection from now on. Note: gctwa does not evaluate its argument.

(gctwa nil) causes this continual removal of truly worthless atoms to be shut off, but it does not affect whether the next garbage

collection removes twa's.

The value returned by gctwa is a fixnum which is 0 if no garbage collection of truly worthless atoms will be done, 1 if twa's are to be gc'ed on the next garbage collection, 10 if twa's are to be gc'ed on all garbage collections, or 11 if both. (These numbers are octal.)

^d SWITCH

If the value of ^d is non-nil, the garbage collector prints an informative message after each garbage collection.

See also the variables gc-daemon and gc-lossage.

12.6.2 - Spaces

In MACLISP the storage used for LISP objects is divided into several conceptual subdivisions, called *spaces*. Each space contains a different type of object. Allocation proceeds separately in the different spaces, but garbage collection of all spaces occurs together since an object in one space could contain a pointer to an object in any other space.

For example, in the "Bibop" version of the pdp-10 implementation, the spaces are as follows:

LIST conses (dotted pairs) and lists.
FIXNUM fixnums.
FLONUM flonums.
BIGNUM bignum headers. bignums also occupy fixnum and list space.
SYMBOL atomic symbols.
ARRAY "special array cells."
REGPDL the "regular" pushdown list.
SPECPDL the "special" pushdown list, used in binding.
FXPDL the fixnum pushdown list, used for temporary numeric values.
FLPDL the flonum pushdown list, used for temporary numeric values.
Binary Program Space used to hold arrays and compiled code.
pure LIST, pure FIXNUM, pure FLONUM, pure BIGNUM
See the Bibop manual; these spaces are not of general interest.

In the Multics implementation, the spaces are:

list conses (dotted pairs), lists, atomic symbols, bignums, and strings.
Static Storage arrays, files, and linkage to compiled code.
markedpdl a pushdown list of lisp objects.

unmarkedpdl a pushdown list of machine data, not lisp objects.

Note: in the Multics implementation there is no space for numbers because numbers are stored in such a way that they do not take up any extra room.

Associated with each space is information determining when an attempt to allocate in that space should cause a garbage collection. The idea is that one should allocate for quite a while in a space, and then decide that it is worth the trouble of doing an expensive garbage collection in order to prevent the space from using too many bits of actual storage.

The exact nature of this information varies with the space. In a pushdown list (pdl) space, all information must be stored contiguously so the only parameter of interest is how big the pdl is. This can be measured in three ways, so there are three parameters associated with a pdl:

pdlsize the number of words of valid data in the pdl at the moment.

pdlmax the size to which the pdl may grow before intervention is required. This is used to detect infinite recusion.

pdlroom the size beyond which the pdl may not grow no matter what.

A space such as a list space has three parameters, called the gcsize, gcmax, and gcmin. These are are in machine-dependent units of "words." The gcsize is the expected size of the space; as objects are allocated in the space it will grow without garbage collection until it reaches this size. When it gets above this size garbage colletion will occasionally be required, under control of the other two parameters.

The gcmax is the maximum size to which the space should grow; if it gets this big garbage collections will occur quite frequently in an attempt to prevent it from growing bigger.

The gcmin specifies the minimum amount of free space after a garbage collection. It may be either a fixnum, which specifies the number of words to be free, or a flonum which specifies the fraction of the space to be free. The exact interpretation of this depends on the implementation. In the pdp-10 implementation, which uses free storage lists, the gcmin is the number of words which must be on the free storage list after a garbage collction. If there are not this many, the space is grown, except if its size approaches gcmax it may not be grown by the full amount. In the Multics implementation, which uses a compacting garbage collector, the criterion for garbage collection is not when a free list is exhausted but when the space reaches a certain size. This size is the maximum of gcsize and the sum of the size after compactification plus gcmin (if it is a fixnum) or the size after compactification times 1/(1-gcmin) (if gcmin is a flonum.) The effect of this is to allow the same amount of allocation between garbage collections as there would be in the pdp-10 implementation with the same gcmin.

Note that these controls over the sizes of spaces are somewhat inexact,

since there is rounding. For instance, the Bibop version of the pdp-10 impementation allocates memory to spaces in units of 256. or 512. words. The Multics implementation allocates at least 16,384 words between garbage collections and presently controls the size of pushdown lists in units of 1024. words.

Some spaces, such as Binary Program Space in the pdp-10 implementation or Static space in the Multics implementation are not subject to detailed control by the user. The management of these spaces is entirely automatic. Generally these are spaces where the rate of allocation is fairly placid and most objects, once allocated, are used forever and never freed. Hence the exact policy used for storage management in these spaces is not too important.

12.6.3 - Storage Control Functions

`alloc` SUBR 1 arg

The `alloc` function is used to examine and set parameters of various spaces having to do with storage management. To set parameters, the argument to `alloc` should be a list containing an even number of elements. The first element of a pair is the name of a space, and the second is either a fixnum or a 3-list. A fixnum specifies the pdlsize (for a pdl space) or the gcsize (for other spaces.) A 3-list cannot be used with a pdl space. It specifies, from left to right, the gcsize, gcmax, and gcmin. `nil` means "don't change this parameter." Otherwise a fixnum must be supplied, except in the third element (the gcmin), where a flonum is acceptable.

An example of this use of `alloc`, in the pdp-10 Bibop implementation:

```
(alloc '(list (30000. 5000. 0.25)
        fixnum (4000. 7000. nil)
        regpdl 2000.))
```

or, in the Multics implementation:

```
(alloc '(list (30000. nil 0.3)
        markedpdl 5000.
        unmarkedpdl 5000.))
```

`alloc` may also be called with an argument of t, which causes it to return a list of all the spaces and their parameters. This list is in a form such that it could be given back to `alloc` at some later time to set the parameters back to what they are now.

The following functions are certain cases of the `status` and `sstatus` functions which are described in section 12.7.

`(status spcnames)`

Returns a list of the names of all the spaces available in the LISP being used. These are the spaces acceptable to the `alloc` function.

`(status spcsize` *space*`)`

Returns the actual, current size of *space*, in words. *space* is evaluated.

`(status pdlsize` *space*`)`

Returns the current number of words on a pdl.

`(status pdlroom` *space*`)`

Returns the "pdlroom" of a pdl, i.e. the maximum size to which it may ever grow.

`(status pdlmax` *space*`)`

Returns the current value of the "pdlmax" parameter of a pdl.

`(sstatus pdlmax` *space size*`)`

Sets the pdlmax parameter for the pdl *space* to *size*. Both arguments are evaluated.

12.6.4 - Dynamic Space and Pdl Expansion

There are several user interrupts generated by the storage management. See section 12.4 for a description of user interrupts. The `gc-daemon` interrupt (number 20.) occurs after each garbage collection. The argument passed to the gc-daemon interrupt handler is a list of items; each item has the form `(space before . after)`, where `space` is the name of a space, `before` indicates the number of cells free before the garbage collection, and `after`

indicates the number of cells free afterwards. (In the Multics implementation, where "free cells" is a meaningless concept, only the difference of these two numbers is significant. It represents the amount of compaction achieved.)

The gc-lossage interrupt (number 10.) occurs if the garbage collector tries to expand a space but fails because, for example, the operating system will not give it any more storage. The argument passed to the interrupt service function is the name of the space that lost.

The pdl-overflow interrupt (number 12.) is signalled when some pushdown list exceeds its pdlmax. The pdlmax is increased slightly so that the interrupt handler will have room to run. The argument passed to the interrupt function is the name of the pdl that overflowed. If the interrupt function uses too much pdl, this interrupt will occur again. If this happens enough times, the pdlmax will reach the pdlroom, there will be no room in the pdl to take a user interrupt, and an uncorrectable error will occur.

The interrupt function can decide to terminate the computation that overflowed the pdl, by doing an (ioc g) or a (throw), or it can increase the pdlmax by using alloc or (sstatus pdlmax) and then continue the computation by returning. Note that unlike most other user interrupts, if the pdl-overflow interrupt function returns nil (or the ";bkpt pdl-overflow" is $p'ed), the computation is continued as if the pdl overflow had not occurred.

The gc-overflow interrupt (number 13.) occurs when some space (other than a pdl) exceeds its gcmax. This gives the user a chance to decide whether the size of the space should be increased and the computation continued, or that something is wrong and the computation should be terminated. The argument passed to the interrupt handler is the name of the space that overflowed. The interrupt handling function will be able to run because the garbage collector makes sure that the space is sufficiently large before signalling the interrupt, even if this makes it become somewhat larger than its gcmax. This interrupt is similar to pdl-overflow; if the interrupt handler function returns at all, even if it returns nil, the interrupted computation proceeds. To terminate the computation an explicit (ioc g), (throw), or (error) must be done.

12.6.5 - Initial Allocation

The pdp-10 implementations of MACLISP run on a machine with a limited-size address space. Consequently the allocation of portions of this address space to different uses, such as LISP storage spaces, becomes important. This is particularly true of implementations without the "Bibop" feature, which do not take advantage of paging.

When LISP is first entered, it goes through a dialogue with the user known as "allocation." Normally the dialogue simply consists of the user

declining to specify anything, in which case LISP chooses suitable defaults. If a large problem is to be worked on, the defaults may be inappropriate and it may be necessary to explicitly allocate a larger amount of storage. It is also possible for the user's replies to come from a file.

If LISP is called with a command line from DDT, for example

:LISP INDEX LOADER COM:

it reads the indicated file in the same way that it would read .LISP. (INIT). See below.

On the other hand, if LISP is called without a command line, it identifies itself and asks

ALLOC?

Suitable responses are Y, N, and CTRL/Q. There are other obscure characters which can be used as replies to this question, but these three are sufficient. "N" means that you do not want to specify allocation. You will get the default. "Y" means that you wish to go through the following sequence of questions and answers.

LISP types out the names of various spaces and their sizes. The first one, "CORE", is special. It is not a space but the total amount of address space desired, and the size is in pages rather than words. After each question you may enter altmode, which terminates the dialogue and gives the remaining parameters default values, or space which goes on to the next question. Before your altmode or space you may put a number which is the size you want that space to be, instead of the number that was printed. Again, there are various other magic characters besides space and altmode.

If you reply with a control-Q, it means to read your .LISP. (INIT) file. The first form in the file should be a comment which is used to answer the questions. Note that supplying nonexistent space names in the comment doesn't hurt, so you can use the same comment for both Bibop and non-Bibop LISP. An example of the form of this comment is:

```
(comment fxs 4000 fixnum 5000 fls 2000
         symbol 4000 flonum 2000 bignum 1400)
```

12.7 - The Functions status and sstatus

status FSUBR

The `status` function is used to get the value of various system variables. Its first argument, not evaluated, is an atomic symbol indicating which of its many functions `status` should perform. The use of additional arguments depends on what the first argument is. Omitted arguments are assumed to be `nil`. The following "status functions" exist:

STATUS FUNCTIONS FOR THE I/O SYSTEM

Note: in the following, *c* represents an argument specifying a character. If *c* is non-atomic it is evaluated, and the value must be a fixnum which is the ascii code for a character. If *c* is atomic it is not evaluated, and it may be a fixnum or a character object.

`(status ioc c)` gives the state (`t` or `nil`) of the control-"c"-switch. For example, `(status ioc q)` is `t` if input is from a file, `nil` if input is from the terminal.

`(status uread)` returns a 4-list for the current `uread` input source, or `nil` if `uread` is not being done.

`(status uwrite)` returns the corresponding list for the current `uwrite` output destination.

`(status crunit)` returns a 2-list of the current unit; i.e. device and directory.

`(status crfile)` returns a 2-list giving the file names for the current file in the "uread" I/O system.

`(status tabsize)` returns the number of character positions assumed between tab stops.

`(status newline)` returns a fixnum which is the ascii code for the character which marks the end of a line of input. For example, `(= (setq ch (tyi)) (status newline))`

`(status charmode f)` returns the value of the character-mode switch for the file object *f*. If *f* is `nil` or omitted the terminal is assumed. If this switch is `t` (the normal case for the terminal) output is sent to the device as soon as it is generated. If the switch is `nil` (the normal case for files other than the terminal) output is held until a newline is typed, an error occurs, input is requested, or the buffer becomes full. This provides increased efficiency at the cost of not immediately seeing all output.

STATUS FUNCTIONS FOR THE READER

See section 13.6.2 for a description of how the data examined by these functions is used.
(status chtran *c*) gets the character translation table entry for the character *c*. This is the ascii code of a character substituted for *c* when it appears in a pname being read in.

(status syntax *c*) returns the 26 syntax bits for the character *c*, as a fixnum.

(status macro *c*) returns nil if *c* is not a macro character. If *c* is a macro character it returns a list of the macro character function and the type, which is nil for normal macros and s for splicing macros.

(status +) gets the value of the + switch (t or nil). This switch is normally nil. If it is t, atomic symbols more than one character long beginning with a + or a - are interpreted as numbers by the reader even if they contain letters. This allows the use of input bases greater than ten.

(status ttyread) returns the value of the ttyread switch, which is kept in the readtable. At present this is not used for anything in the Multics implementation. In the pdp-10 implementation it controls whether tty "force feed" characters are used.

STATUS FUNCTIONS FOR THE PRINTER

(status terpri) returns the value (t or nil) of the terpri switch, which is kept in the readtable. This switch is normally nil. If it is t, the output functions such as print and tyo will not output any extra newlines when lines longer than linel are typed out.

(status _) returns the value (t or nil) of the _ switch, which is kept in the readtable. If this switch is t, the _ format for fixnums with lots of trailing zeroes is not used.

(status abbreviate) returns the value of the abbreviation control. See section 13.7 for a description of the abbreviation control.

STATUS FUNCTION FOR THE GARBAGE COLLECTOR

(status gctime) returns the number of microseconds spent garbage-collecting.

ENVIRONMENT ENQUIRIES

(status date) returns a 3-list indicating the current date as (year-1900. month_number day)

(status daytime) returns a 3-list of the 24-hour time of day as (hour minute second).

(status time) is the same as (time), the number of seconds the system has been up.

(status runtime) is the same as (runtime), the number of microseconds of cpu time that has been used.

(status system *x*) returns a list of the system properties of the atomic symbol *x*, which is evaluated. This list may contain subr, fsubr, macro, or lsubr if *x* is a function, and value if this atomic symbol is a system variable.

(status uname) returns an atomic symbol whose pname is the user's login name. In the Multics implementation this is in the format User.Project; the dot will be slashified if print is used to display this.

(status udir) returns the name of the user's directory. In the ITS implementation this is the same as the user's name as returned by (status uname). In the Multics implementation this is the user's default working directory. In the DEC-10 implementation this is a list (proj prog).

(status lispversion) returns the version number of lisp.

(status jcl) returns the "job command line" from DDT in the ITS implementation. In the Multics implementation it returns the explodec'd second argument of the lisp command, or else nil if the lisp command did not have two arguments. If lisp was invoked by

lisp environment_name "foo bar"

then (status jcl) => (f o o / b a r)

The following status functions only exist in the Multics implementation.

(status paging) returns a list of the paging-device page faults and total page faults that have been incurred.

(status arg *n*) returns the *n*+1'th argument of the lisp command, as an interned atomic symbol. nil is returned if *n* is greater than the number of arguments on the lisp command.

MISCELLANEOUS STATUS FUNCTIONS

(status toplevel) returns the top-level form, which is continually evaluated when LISP is at its top level. If this is nil, a normal read-eval-print loop is used.

(status interrupt *n*) returns the service function for user interrupt channel *n*. *n* is evaluated.

(status pagepause) returns the value of the pagepause flag. See (sstatus pagepause) for a description of this flag.

(status uuolinks) returns a number which represents the number of available slots for linking between compiled functions.

(status divov) returns the state of the "divide overflow" switch. If this switch is nil an attempt to divide by zero causes an error. If the switch is t the result of a division by zero is the numerator plus 1.

(status features) returns a list of symbols representing the features implemented in the LISP being used. The following symbols may appear in this list:

bibop	pdp-10 big-bag-of-pages memory management scheme
lap	this LISP has a Lisp Assembly Program
sort	the sorting functions described in chapter 11 are present
edit	the edit function described in chapter 18 is present
fasload	the fasload facility described in chapter 14 is present
^f	the "moby I/O" facility is present
bignum	the arbitrary-precision arithmetic package is included in this LISP
strings	character strings and the functions on them described in chapter 8 are present
newio	the I/O functions described in chapter 13 are included in this LISP; if this feature is not present only some of those functions are available.
ml	this LISP is on the MathLab machine at MIT
ai	this LISP is on the AI machine at MIT
H6180	this LISP is on the H6180 Multics machine at MIT
its	this LISP is on some ITS system
Multics	this LISP is on some Multics system
dec10	this LISP is on some DEC TOPS-10 system; or on some TENEX system since the TENEX implementation runs under a TOPS-10 emulator.

(car (last (status features))) is generally considered to be an implementation name, such as its or dec10 or Multics. The main idea behind this status call is that an application package can be loaded into any MACLISP implementation and can decide what to do on the basis of the features it finds available.

Example:

```
(cond ((memq 'bignum (status features))
       (prog2 nil (eval (read)) (read)))  ;use first
      (t (read) (eval (read)) ))          ;use second

(defun factorial (n)          ;bignum version
    (cond ((zerop n) 1)
          ((times n (factorial (sub1 n))))
             ))

(defun factorial (n)          ;fixnum-only version
   (do () ((not (> n 13.)))  ;do until n ≤ 13.
          (error "argument too big - factorial"
                 n
                 'wrng-type-arg))
    (cond ((zerop n) 1)
          ((* n (factorial (1- n)))) ))
```

(status feature *foo*) is roughly equivalent to (memq '*foo* (status features)), i.e. it determines whether this LISP has the *foo*-feature. Note that *foo* is not evaluated.

(status status *foo*) returns t if *foo* is a valid status function. If it is not, nil is returned.

(status status) returns a list of valid status functions. The names are truncated to some implementation-dependent number of characters, such as 4 or 5.

(status sstatus *foo*) returns t if *foo* is a valid sstatus function. If it is not, nil is returned.

(status sstatus) returns a list of valid sstatus functions. As with (status status), the names are truncated to some implementation-dependent number of characters, such as 4 or 5.

sstatus FSUBR

sstatus is like status. It has a first argument which tells it what to do and how to interpret the rest of its arguments. sstatus is used to set various system variables. The following "sstatus functions" exist:

SSTATUS FUNCTIONS FOR THE I/O SYSTEM

(sstatus ioc *ccc*) is the same as (ioc *ccc*)

(sstatus uread --args--) is the same as (uread --args--)

`(sstatus uwrite --args--)` is the same as `(uwrite --args--)`

`(sstatus crunit --args--)` is the same as `(crunit --args--)`

`(sstatus charmode` *x f*`)` sets the character-mode switch of the file *f* (*f* may be `nil` to signify the terminal) to *x*, which may be `t` or `nil`. *x* and *f* are evaluated. *f* may be omitted, in which case the terminal is assumed.

SSTATUS FUNCTIONS FOR THE READER

See section 13.6.2 for a description of how the switches and tables set by these functions are used.

`(sstatus chtran` *c k*`)` sets *c*'s character translation to *k*. *k* follows the same rules as *c*, i.e. it may be a list which evaluates to a fixnum, or an unevaluated atom such as a fixnum or a character object. The value returned is *k* as a fixnum ascii code.

`(sstatus syntax` *c m*`)` sets *c*'s syntax bits to *m*. *m* is evaluated and returned.

Note that in the above 2 calls, if *c* is a macro character it is changed back to its standard syntax and chtran before the requested operation is performed. However, if in the standard readtable *c* is a macro (i.e. ' and ;), instead of being changed to its standard syntax and chtran its syntax is set to 502 (extended alphabetic) and its chtran is set to itself.

`(sstatus macro` *c f*`)` makes *c* a macro character which calls the function *f* with no arguments. *f* is evaluated. A fourth argument to `sstatus` may be supplied. It is not evaluated. If it is an atomic symbol whose pname begins with `s`, *c* is made a splicing macro. If *f* is `nil`, instead of *c* being made a macro-character, *c*'s macro abilities are taken away and *c* becomes an ordinary extended-alphabetic character.

`(sstatus +` *x*`)` sets the + switch to `t` or `nil` depending on *x*, which is evaluated. The new value of the + switch is returned.

`(sstatus ttyread` *x*`)` sets the ttyread switch to `t` or `nil` depending on *x*, which is evaluated. The new value of the switch is returned.

SSTATUS FUNCTIONS FOR THE PRINTER

`(sstatus terpri` *x*`)` sets the terpri switch.

`(sstatus abbreviate` *n*`)` sets the abbreviation control to *n*.
`(sstatus abbreviate nil)` turns off abbreviation.
`(sstatus abbreviate t)` turns on a maximal amount of abbreviation. See section 13.7 for a description of the abbreviation control.

SSTATUS FUNCTION FOR THE GARBAGE COLLECTOR

(sstatus gctime *n*) resets the gctime counter to *n* and returns the previous value of the gctime counter.

MISCELLANEOUS SSTATUS FUNCTIONS

(sstatus toplevel *x*) evaluates and returns *x* and sets the top level form to this value.

(sstatus uuolinks) causes all links between compiled functions to be "unsnapped." This should be done whenever (nouuo t) is done to insure that the interpreter always gets a chance to save debugging information on every function call.

(sstatus divov *x*) sets the "divide overflow" switch to *x*. nil means division by zero should cause an error, while t means the result of division by zero should be a quotient of the numerator plus 1 and a remainder of zero.

(sstatus interrupt *n f*) sets the service function for user interrupt channel *n* to *f*. The arguments are evaluated. *f* is returned.

(sstatus pagepause *x*) sets the pagepause switch to *x*, which may evaluate to t or nil. If the pagepause switch is t, and a display terminal is being used, LISP will stop when it gets to the end of the screen and wait for the user to hit control-U before it erases anything. (This feature is not presently available in the Multics implementation.)

*rset SWITCH

The value of *rset is the "*rset flag" manipulated by the status and sstatus functions. If *rset is t, the interpreter keeps extra information and makes extra checks to aid in debugging. *rset should never be setq'ed - always use (sstatus *rset t) or (sstatus *rset nil).

12.8 - Miscellaneous Functions

sysp SUBR 1 arg

The sysp predicate takes an atomic symbol as an argument. If the atomic symbol is the name of a system function (and has not been redefined), sysp returns the type of function (subr, lsubr, or fsubr). Otherwise sysp returns nil.

Examples:

```
(sysp 'foo) => nil        (presumably)

(sysp 'car) => subr

(sysp 'cond) => fsubr
```

12.8.1 - Time

`runtime` SUBR no args

(`runtime`) returns the number of microseconds of cpu time used so far by the process in which LISP is running. The difference between two values of (`runtime`) indicates the amount of computation that was done between the two calls to `runtime`.

`time` SUBR no args

(`time`) returns the time that the system has been up, in seconds. (As a flonum.)

`sleep` SUBR 1 arg

(`sleep` *n*) causes a real-time delay of *n* seconds, then returns *n*. *n* may be a fixnum or a flonum.

See also the `alarmclock` function, section 12.4.3.

12.8.2 - Getting into LISP

in the Multics implementation

The LISP subsystem is entered by means of the lisp command. If lisp is called with no arguments, a copy of the standard initial environment containing all the system functions and variables is made the current environment. If the lisp command is issued with an argument, the argument concatenated with ".sv.lisp" is the pathname of a saved environment (see the save function in section 12.8.3) which is copied into the current environment. Additional arguments to the lisp command in this case are actually arguments to the programs in the saved environment, which can retrieve them by using the (status arg *n*) function.

The LISP subsystem may also be entered through the lisp_compiler command, which is like calling lisp with an argument of the pathname of the saved environment containing the LISP compiler.

When the standard initial environment is entered, lisp checks for a segment named start_up.lisp in the user's home directory. If such a segment exists, it is read in, using the load function. This facility allows users to "customize" LISP.

When a saved environment is entered, (mapc 'eval errlist) is done. This feature can be used to make the saved environment self-starting.

in the ITS implementation

LISP may be entered by the ":LISP" or "LISP^K" command. The environment set up by this command is the standard initial environment. Lisp checks for a file named .LISP. (INIT) in the user's directory, or a file named UNAME .LISP. in the (INIT) directory if the user does not have a directory.

If it is found, it is read in and each form is evaluated. This allows a user to "customize" lisp. The .LISP. (INIT) file may begin with a comment in the form

```
(comment name number name number ...)
```

which is used to specify allocation. The names are the names of the various spaces, and the numbers are the sizes to be allocated. If there is no .LISP. (INIT) file, lisp asks the question ALLOC? A reply of n causes standard allocation, a reply of y allows allocation to be specified conversationally.

LISP may be entered by the command :LISP name1 name2 dev dir where dev and dir default to DSK and the user's directory. In this case the file dev:dir;name1 name2 is read in the same way as a .LISP. (INIT) file.

If a lisp containing some data other than the initial environment has been saved (see section 12.8.3) as TS NAME, the :NAME command will retrieve it. The first action performed by lisp is (mapc 'eval errlist), which allows the user programs to start up.

in the DEC-10 implementation

Type the monitor command, R LISP. LISP will ask the question "ALLOC?", and the possible answers are as described above for the ITS implementation of LISP. Similarly to the ITS implementation, if a lisp has been saved as NAME.SAV, the RUN NAME command will retrieve it and the first thing it will do when retrieved is (mapc 'eval errlist).

12.8.3 - Getting Out of LISP

Evaluating `(ioc z)` will cause LISP to temporarily release control. Control can be returned to LISP by an implementation defined method. There is also an implementation-defined way to leave LISP permanently, freeing any storage used by the current LISP environment or else saving it.

in the Multics implementation

`(ioc z)` will cause a QUIT. The start command may be used to re-enter the lisp subsystem. release may also be used, in which case LISP will clean itself up.

There are also the functions `quit` and `save`. `quit` causes the lisp subsystem to exit, throwing away the current environment. `save` causes the lisp subsystem to exit, saving the current environment in a specified file, whose name always ends in ".sv.lisp".

`quit` SUBR no args

`(quit)` causes the lisp subsystem to remove itself and return to its caller. The current environment is lost. (cf. `save`).

`save` FSUBR

`(save foo)` saves the current LISP environment in a file named foo.sv.lisp in the working directory. The argument is not evaluated. The command

```
lisp foo
```

can be used to retrieve this saved environment. All variable values, file objects, array contents, and function definitions (and other properties) are saved, but the contents of the push down lists, including previous values of bound variables, cannot be saved, so `save` should only be used from top level.

See also the function `cline` (described in section 12.8.4), which is used to get out of LISP, execute one Multics command line, and return into LISP.

(ioc z) will cause a return to DDT. $P may be used to re-enter LISP. (valret ':KILL) will cause LISP to exit, throwing away the current environment. The macdmp function, explained below, may be used to save the current environment.

macdmp LSUBR 0 or 1 args

If LISP is a top-level procedure (for instance if it is disowned), macdmp simply logs out. Otherwise it prepares the LISP for dumping as follows:

(1) if the display slave is open, it is closed.
(2) all bit tables, pdl areas, and free storage lists are zeroed out. (Remember, dumping with $Y uses a special compression technique on blocks of zeros.)
(3) if macdmp was given an argument, this is explodec'ed and valret'ed as with the valret function. Otherwise, macdmp returns to DDT with a .BREAK 16,100000.

Commonly one will write a setup routine for some LISP package like this:

```
(progn
      (terpri)
      (princ 'options:)
      ... read in options ...
      (terpri)
      (princ 'loading)
      ... load in files of functions ...
      (macdmp ':$ALL/ DONE/ -/ hooray!$/ ))
```

The LISP (with some new functions loaded) is now ready for dumping. Alternatively, one might write

```
(macdmp '$Y/ USER/;FOO/ BAR/ :$ALL/ DONE$/ )
```

which will do the dump and then print a message when done.

When such a dumped LISP is loaded and restarted, the effect is very similar to a ^G quit: LISP's top level is entered after evaluating all items on the errlist. Thus if before the macdmp is done the errlist is setq'ed to some list of initialization procedures, these procedures will be invoked when the dumped LISP is restarted.

in the DEC-10 implementation

Typing two control-C's causes control to be returned to the monitor. The CONTINUE command will return control to LISP. Running another program causes LISP to be thrown away.

The LISP may be saved for later use by evaluating (macdmp), which prepares for saving and then returns control to the monitor. The SAVE command may now be used. When the saved LISP is run at some later time, (mapc 'eval errlist) will be automatically performed. This allows the saved program to be self-starting.

12.8.4 - Sending Commands to the Operating System

in the Multics implementation

`cline` SUBR 1 arg

(`cline` *x*), where *x* is a character string, executes the Multics command *x* and returns `nil`.
Example:

```
(cline "who -long")
```

in the ITS implementation

`valret` LSUBR 0 or 1 args

(`valret`) is like (`ioc z`); that is, it does a .LOGOUT if LISP is a top level procedure, and otherwise valrets ":VK " to DDT.

(`valret` *x*) effectively performs an explodec on *x* (in practice *x* is some strange atomic symbol like `:PROCED/ :DISOWN/` , but it may be any S-expression). If the string of characters is one of "$^X.", ":KILL ", or ":KILL^M" then valret performs a "silent kill" by executing a .BREAK 16,20000; otherwise valret performs a .VALUE, giving the character string to DDT to evaluate as commands.

Examples:

```
(valret ':PROCED/ :DISOWN/ )
```
procedes and disowns the LISP.

```
(valret '/ :KILL/ :TECO/^M)
```
kills the LISP and starts up a TECO.

```
(valret '0$N)
```
causes DDT to print out the contents of all non-zero locations in LISP.

in the DEC-10 implementation

There is currently no way to do this in the DEC-10 implementation. However, (`valret`) will return control to the monitor so that a command may be manually typed. Then type CONTINUE to resume lisp.

13 - Input and Output

13.1 - Basic I/O

Input and output can be done in LISP in terms of S-expressions or in terms of characters. Operations may also be performed on certain devices, such as displays, robot arms, etc., in terms which are peculiar to the particular device, using the so-called "moby I/O" facility.

Initially we will discuss just I/O on the user's terminal.

S-expressions can be input by using the function `read`. `(read)` reads one S-expression, which is either a list enclosed in matching parentheses or an atom delimited by a special character such as a space or a parenthesis, which is saved and used to begin the next S-expression read. `read` returns the S-expression which it read, converting it from the external representation as characters to LISP internal form. See Chapter 2 and section 13.1.2.

`(readch)` reads in one character and returns it as a character object.

`(tyi)` reads in one character and returns a number which is the ascii code for the character.

`(print` *x*`)` prints out the S-expression *x* in a form which is readable by humans but which could also be read back into LISP if it was printed to a file rather than to the terminal. See section 13.2.3 for an explanation of how to do this.

The expression printed out is preceded by a newline and followed by a space. If special characters are used with other than their normal meanings, for example if a parenthesis appears in the pname of an atom, they are preceded by slashes so that the output could be read back in. Strings are enclosed in double quotes for the same reason.

`(prin1` *x*`)` is the same as `(print` *x*`)` except that the leading newline and trailing space are omitted. `prin1` can be used to print multiple items on a line, but spacing between the items must be provided for explicitly, for example by evaluating `(tyo 40)`.

`(princ` *x*`)` is like `(prin1` *x*`)` except that special characters are not slashified and strings are not quoted. This makes the output more pleasing in certain situations, however it cannot be read back into LISP.

`(terpri)` types out a newline.

Output of characters can be accomplished using either `tyo` or `princ`. `(tyo` *n*`)` outputs a character whose ascii code is given by the number *n*. `princ` may be used to output character objects.

As implied above, these functions can also be used to do I/O on devices other than the terminal. The ways to do this will be explained in section 13.2.3.

Note that what LISP does when it is at its "top level," that is when you first start talking to it, is first to call `read`, then to call `eval` on what was read, then to `print` the result and advance the terminal to a new line on which the next piece of input may be typed. This may be expressed as repeated evaluation of:

```
(prog2 (terpri)
       (print (eval (read))) )
```

13.2 - Files

I/O in LISP consists of communication between the LISP environment and sequences of characters called files, located in the external world. LISP refers to these files by using "file objects," which are special objects within the LISP environment which serve as representatives of, or symbols for, the files in the external world. Because there is a one-to-one correspondence between files and file objects, it is often convenient to confuse the two and call them both "file."

The LISP system includes functions which can manipulate files in various ways: A file may be "opened," that is a file object may be created and associated with a named file in the external world.

A file may be "closed," that is the association between the file-object and the external file may be broken and the file-object deleted.

The file-accessing information contained in a file-object may be examined or changed; for example, the line length of an output file may be adjusted.

The characters of information in the external file may be read or written.

The attributes of the external file, such as its name, may be changed.

In order to "open" a file, the external file and the file object must be named so that a connection may be established between them. The problem of naming file objects is solved trivially by making the rule that whenever a file object is created its name is decided by the system and returned as the value of the function that created it. File objects are then referred to in the same way as any S-expression. Note that the name of a file object does not have a printable form, so that if you want to manipulate the file object by typing from the terminal (rather than from a program), you must keep the file object as the value of an atomic symbol.

The naming of files in the outside world is more difficult because MACLISP has to operate with several different outside worlds, that is, under several different operating systems. It was thought undesirable to build too many assumptions about the operating system into the language, because that would restrict the transporting of programs between MACLISP implementations.

The assumptions that are built in are that the operating system provides named files located in named directories or on named devices, which may be accessed sequentially as streams of characters. The function `filepos` makes the additional assumption that simple random-access facilities are available. An interactive environment is also assumed. Some of the I/O functions assume that the names of files may be broken up into an arbitrary number of components, so that names take on a form such as "foo.bar.lisp" or "moby mess". However, it is possible for a MACLISP to operate with somewhat reduced effectiveness under an operating system which does not

satisfy all of these assumptions.

The user of a program or a subsystem written in LISP wants to be able to type in file names in the form customary in the particular operating system being used, and he wants to see them typed out in the same form. But if a program wants to do more with the file name supplied by the user than simply pass it on to the system I/O functions, it needs to have that name translated to a uniform internal format, in which the interesting components of the name are seperate atoms in a list, not buried inside a character string whose format is not even known to the program. To resolve this conflict, two forms for file names have been defined, and functions are provided to make the implementation-dependent translation from one form to the other. The forms of a file name are called the ***namelist*** and the ***namestring***.

The namestring is the implementation dependent form. Namestrings are represented as LISP character strings, however atomic symbols may also be used, in which case the pname of the atomic symbol is used as the character string. The contents of a namestring is just a sequence of characters which have meaning to the user and to the function `namelist`, which converts a namestring to a namelist. Namestrings should be read in using the `readstring` function and printed out using `princ`, so that no quotes will appear around them.

A namelist is a list whose car somehow specifies the device and/or directory which contains the file, and whose cdr specifies the name of the file. The exact way in which the car specifies the device/directory is implementation-dependent. It should not be of concern to programs. The cdr of a namelist is a list of names which are the components of the file name if the operating system uses multi-component file names. Each name is represented as an atomic symbol, which is "interned" so that it may be tested with the function eq.

An additional feature of namelists is the "star convention," by which a namelist may contain unspecified components, which are indicated by the atom *. Certain other constructions, explained in section 13.3, may also be used. The star convention allows a single namelist to specify a class of files, and allows programs to apply defaults to their file-name arguments in a straightforward fashion.

Some additional information about file objects has been collected here. It is in brief form and will be elaborated in later sections.

There is no way to input file objects to the reader, because they do not have pnames or anything of that sort, but for convenience in error messages and debugging the printer will print a file object as a sharp sign (#), followed by the namestring of the external file to which the file object is attached. # is the character which is used to indicate that an object of unknown type is being printed.

The information contained within a file object is here described briefly.

Namelist	the namelist for the external file of which the file object is a representative.
Eoffn	a function which is applied when the end of an input file is reached.
Endpagefn	a function which is applied when the end of a page is reached on an output file.
Linel	the number of characters per line on an output file.
Charpos	the horizontal position on the line, where 0 is the left margin.
Chrct	the number of character positions remaining on the current line of an output file,
Pagel	the number of lines per page.
Linenum	the number of the current line, with 0 being the top of the page.
Pagenum	the number of the current page, with the first page being 0.
Filepos	the position within the file of the character currently being accessed. (Not necessarily meaningful for all kinds of files.)
Other	internal information used by the LISP I/O functions in transactions with the operating system.

Note that as a special case nil is considered to be a file object which represents the terminal. This is in addition to nil's other identities as an atomic symbol and as a list.

13.2.1 - Naming Files

Some examples may help clarify the connection between namelists and namestrings.

In the Multics system, files are stored in a tree structure of directories. A file's name consists of a sequence of names seperated by the ">" character. The last name is the name of the file, and preceding names are the names of directories in a path from the "root" directory down through the tree to the file in question. Each name may consist of several components seperated by periods. Thus a typical namestring in the Multics implementation of MACLISP would be

```
">udd>AutoProg>Hacker>hacks>my.new.hack"
```

The corresponding namelist is:

```
(>udd>AutoProg>Hacker>hacks my new hack)
```

In addition, the star convention for namelists may also be represented in namestring form. Some examples of the correspondence are:

```
(* foo) == "foo"              - omitted components are *
(* foo * bar) == "foo.*.bar"
(frotz foo . bar) == "frotz>foo.**.bar"
(* foo . *) == "foo.**"
```

Multics LISP can also use "streams" for files. Streams are a sequential-I/O entity in Multics. For example, input from and output to the terminal are performed by means of streams. In LISP the convention has been defined that a "$" character at the beginning of a namestring distinguishes the name of a stream from the name of a file stored in the directory hierarchy. Thus the namestring

```
"$user_input"
```

indicates the stream used for input from the terminal. The corresponding namelist is:

```
(stream user_input)
```

In the ITS (pdp-10) system, files are stored in directories which are kept on devices. Directories may not be kept within directories, so there is no tree structure. Each file-name has exactly two components. Thus a file whose name has first component foo and second component bar, located in directory comlap on device ml, would have the namestring:

```
ml:comlap;foo bar
```

As a namelist this would be represented:

```
((ml . comlap) foo bar)
```

If the device and directory were omitted, the namelist would be:

```
(* foo bar)
```

If only one component of the name were specified, the second would be *.

In the DEC-10 implementation, namestrings take the usual

```
dev:name.ext[proj,prog]
```

form, and the corresponding namelist is

```
((dev proj prog) name ext)
```

and *'s are substituted for omitted components in the same way as for the ITS version described above.

`namelist` SUBR 1 arg

`namelist` converts its argument to a namelist. Omitted or * components in the argument produce *'s in the result.

The argument to `namelist` can also be a file object or `nil`. Giving `namelist` a file object causes it to return the namelist of that file object. Giving it `nil` causes it to return the default namelist.

`namestring` SUBR 1 arg

`namestring` converts its argument from a namelist to a namestring. It is the opposite of `namelist`.

The argument to `namestring` can also be a file object or `nil`. Giving `namestring` a file object causes it to convert that file object's namelist to a namestring and return it. Giving `namestring nil` causes it to convert the default namelist to a namestring and return it.

`shortnamestring` SUBR 1 arg

`shortnamestring` is just like `namestring` except that there is no mention of directory or device in the resulting string. Example:

```
(shortnamestring '(abc d e)) => "d.e"
```

`defaultf` SUBR 1 arg

`(defaultf` *x*`)`, where *x* is a namelist or a namestring, sets the default namelist to *x*. The default namelist is used to fill out any *'s in the argument to `openi`, `openo`, etc.

In Multics MACLISP, the default namelist is initially set to:

```
(working-directory . *)
```

when LISP is first entered.

In ITS MACLISP, the default namelist is initially set to:

```
((dsk udir) . *)
```

when LISP is first entered. `udir` is what `(status udir)` returns, namely the user's directory. In DEC-10 MACLISP the default namelist is set to

```
((dsk proj prog) . *)
```

when LISP is first entered.

Note: to obtain the default namelist, use `(namelist nil)`.

13.2.2 - Opening and Closing

`openi` SUBR 1 arg

This function is used to create a file object and associate it with a file in the external world. The argument is a namelist or namestring which specifies the file to be opened. The return value is the file object which was created.

`openi` first creates a file object and initializes its endpagefn (for an output file) or eoffn (for an input file) to the default. The linel and pagel are set to appropriate values for the type of device on which the file resides. The charpos, linenum, and pagenum are set to zero. The namelist is set by merging the argument to `openi` and the default namelist. See the description of the `mergef` function, in section 13.3, for the full details. Basically what happens is that components of the file name not specified by the argument to `openi` are obtained from the default namelist. `openi` now negotiates with the operating system to obtain the file. A `fail-act` correctable error occurs if this does not succeed.

The file created by `openi` can be used for input.

Example:

```
(inpush (openi "input_file_1"))
```

`openo` SUBR 1 arg

`openo` is like `openi` except that the file object created is used for output. Any pre-existing file of the same name is over-written.
Example:

```
(setq outfiles (list (openo "output.data")))
```

The following function only exists in the Multics implementation, at present.

opena SUBR 1 arg

opena is just like openo except that if there is a pre-existing file with the same name, the output is appended to the end of the file, where openo would erase the old contents of the file and begin outputting at the beginning of the file. Note that the pagenum, linenum, and charpos are set to zero, not the number of pages, lines, etc. actually present in the file.

close SUBR 1 arg

(close *x*), where *x* is a file, closes *x* and returns t. If *x* is already closed nothing happens, otherwise the file system is directed to return *x* to a quiescent state.

For a description of the way in which the argument to openi, openo, or opena has the defaults applied to it, see section 13.3.

13.2.3 - Specifying the Source or Destination for I/O

When an I/O function is called, the source (file) from which it is to take its input or the destinations (files) on which it is to put its output may be specified directly as an argument to the function, or they may be specified by default. The default input source and output destinations are specified by several system variables, which may be setq'ed or lambda-bound by the user. They are described below.

infile is the default input source, if the switch ^q is t. If ^q is nil the terminal is used as the default input source.

outfiles is a list of default output destinations. Output is sent to all of these files if the ^r switch is t. Output also goes to the terminal unless the ^w switch is t.

Note that in the values of infile and outfiles nil means the terminal, and anything other than nil must be a file object.

infile VARIABLE

The value of infile is a file object which is the default input source if ^q is non-nil. infile can also be nil which specifies that input will be from the terminal even if ^q is not nil. The initial value of infile is nil.

^q SWITCH

If the value of ^q is non-nil, the default input source is the value of the atom infile. If ^q is nil, the default input source is nil, i.e. the terminal.

instack VARIABLE

The value of instack is a list of pushed-down values of infile. It is managed by the function inpush. The initial value is nil.

outfiles VARIABLE

The value of outfiles is a list of file objects which are output destinations if ^r is not nil. Elements of the list outfiles may be either file objects created by openo or opena, or nil meaning output to the terminal. Note that output goes to the terminal anyway if ^w is nil, so it is possible to get double characters this way.

^r SWITCH

If the value of ^r is non-nil, the default output destinations include the files in the list which is the value of the atom outfiles.

^w SWITCH

If the value of ^w is non-nil, the default output destinations do not include the terminal. (Unless ^r is on and nil is a member of the outfiles list.)

Now the basic I/O functions can be explained in full detail:

`read` LSUBR 0 to 2 args

This is the S-expression input function.

`(read)` reads an S-expression from the default input source.

`(read` *f*`)`, where *f* is a file or `nil` meaning the terminal, reads an S-expression from *f*. During the reading, `infile` and `^q` are bound so that evaluation of `(read)` within a macro-character function will read from the correct input source.

`(read` *x*`)`, where *x* is not a file, not `nil`, and not `t`, passes *x* as an argument to the end-of-file function of the input source if the end of the file is reached. Usually this means that `read` will return *x* if there are no more S-expressions in the file.

`(read t)` suppresses the calling of the end-of-file function if the end of the file is reached. Instead, `read` just returns `t`.

`(read` *x f*`)` or `(read` *f x*`)` specifies the end-of-file value *x* and selects the input source *f*.

`readch` LSUBR 0 to 2 args

`readch` reads in one character and returns a character object. The arguments are the same as for `read`.

`readline` LSUBR 0 to 2 args

`readline` reads in a line of text, strips off the newline character or characters at the end, and returns it in the form of a character string. The arguments are the same as for `read`. The main use for `readline` is reading in file names typed by the user at his terminal in response to a question.

`tyi` LSUBR 0 to 2 args

`tyi` inputs one character and returns a fixnum which is the ascii code for that character. The arguments are the same as for `read`.

(tyipeak <as below> <2 args as for read>)

tyipeek LSUBR 0 or 1 arg

(tyipeek) is like (tyi) except that the character is not eaten; it is still in the input stream where the next call to an input function will find it. Thus (= (tyipeek) (tyi)) is (almost) always t. If the end of the file is reached, tyipeek returns 3, the ascii code for "end of text." The end of file function is not called, and the file is not closed.

(tyipeek *n*), where *n* is a fixnum less than 200 octal, skips over characters of input until one is reached with an ascii code of *n*. That character is not eaten.

(tyipeek *n*), where *n* is a fixnum ≥ 1000 octal, skips over characters of input until one is reached whose syntax bits from the readtable, logically anded with (lsh *n* -9.), are nonzero.

(tyipeek t) skips over characters of input until the beginning of an S-expression is reached. Splicing macro characters, such as ";" comments, are not considered to begin an object. If one is encountered, its associated function is called as usual (so that the text of the comment can be gobbled up or whatever) and tyipeek continues scanning characters.

prinl LSUBR 1 or 2 args

(prinl *x*) outputs *x* to the current output destination(s), in a form suitable for reading back in.

(prinl *x f*) outputs *x* on the file *f*, or the terminal if *f* is nil.

print LSUBR 1 or 2 args

print is like prinl except that the output is preceded by a newline and followed by a space. This is the output function most often used.

(print *x*) prints *x* to the default output destinations.

(print *x f*) prints *x* to the file *f*, or to the terminal if *f* is nil.

princ LSUBR 1 or 2 args

princ is like prinl except that special characters are not slashified and strings are not quoted.

(princ *x*) outputs *x* to the current output destination(s).

(princ *x f*) outputs *x* to the file *f*, or the terminal if *f* is nil.

tyo LSUBR 1 or 2 args

(tyo *n*) types out the character whose ascii code is *n* on the current output destination(s).

(tyo *n f*) types out the character whose ascii code is *n* on the file *f* or on the terminal if f is nil. tyo returns its first argument.

terpri LSUBR 0 or 1 arg

(terpri) sends a newline to the current output destination(s).

(terpri *f*) sends a newline to *f*, where *f* may be an output file or nil meaning the terminal.

inpush SUBR 1 arg

(inpush *f*), where *f* is a file object open for input or nil to specify the terminal, pushes the current input source onto the input stack and selects *f* as the current input source. This is like

```
(setq instack (cons infile instack)) (setq infile f)
```

f is returned.

(inpush 0) just returns infile.

(inpush -1) pops a new input source off of the input stack:

```
(setq infile (car instack)
      instack (cdr instack))
```

except that in the case where instack is nil, i.e. empty, inpush leaves instack nil and makes infile nil, which means the terminal.

(inpush -*n*) does (inpush -1) *n* times.

(inpush 1) does (inpush (inpush 0)), (inpush +*n*) does that *n* times.

The result of inpush is the newly selected input source. If inpush causes infile to be set to nil, ^q is set to nil since the terminal has become the input source.

13.2.4 - Handling End of File

Calls to the input functions read, readch, readline, and tyi specify an argument called the "eofval." If this argument is omitted nil is assumed. If the end of the input file is reached during the execution of the function, the eofval argument is used by the following procedure:

Each input file object has an end-of-file handler, its eoffn. When an end of file occurs while input is being taken from this file, the eoffn is examined. (Eof on the terminal cannot occur.) If the eoffn is nil, then the following default action is taken: If eofval on the call to read was not supplied, then the input file is closed and read continues taking characters from a new input file popped off the input stack. If the input stack is empty, (setq ^q nil) is done and read continues reading from the terminal. If an eofval was supplied on the call to read, then read immediately returns it. The input file is not closed.

This is not strictly true in the case where the input function is read (or readline) and it is in the middle of an object (or a line). In this case, rather than allowing the object to cross files, a fail-act error occurs. The argument passed to the user interrupt service function is the list (read-eof). If the interrupt service function returns an atom (such as nil), read signals an error; but if it returns a list, read goes on reading from the new input source as if there had not been any end-of-file.

If the eoffn for the input file is not nil, then it is a function and it is applied with two arguments. The first argument is the file object that eof'ed. The second argument is the eofval on the call to read, or, if an eofval was not supplied, nil. If the eoffn returns nil, the file is closed and reading continues from the input source popped off the input stack. The above prohibition of objects crossing eofs applies. If the eoffn returns t, reading continues from whatever input source was made the current default one by the eoffn. If the eoffn returns something other than t or nil, then read immediately returns whatever the eoffn returned, and the file is not closed unless the eoffn closes it.

`eoffn` LSUBR 1 or 2 args

(`eoffn` *x*), where *x* is an input file, gets *x*'s end-of-file function. The end-of-file function is called if the end of the file is reached during input.

(`eoffn nil`) gets the default end-of-file function.

(`eoffn` *x f*) sets *x*'s end-of-file function to *f*.

(`eoffn nil` *f*) sets the default end-of-file function to *f*.

f may be `nil`, which means that no end-of-file function is to be used.

13.3 - Applying Defaults to File Names

The I/O system provides a mechanism for applying defaults which programs can use and which is used when a file object is created by the open functions.

A default namelist, eoffn, and endpagefn are remembered for initialization of file objects when they are created by the `openi`, `openo`, or `opena` function. These defaults may be examined and modified by use of the `eoffn`, `endpagefn`, `namelist`, and `defaultf` functions. Passing `nil` instead of a file object indicates that the defaults are being referred to.

There is also a system of defaults for file names (actually for namelists), which is based on the use of namelists containing the special atom *.

If one of the elements in a namelist is the atom *, it indicates that that component is not specified. Thus the namelist

```
(* foo bar)
```

specifies a file named foo.bar but it is not said in what directory or on what device it exists. Similarly a namelist like

```
(dir foo *)
```

indicates a file in directory dir, with a two component name of which the first component is foo, but the second component is not specified.

A namelist may also be dotted, that is, it may end with an atom rather than with `nil`. If it ends with an atom other than *, i.e. if it looks like

```
(devdir name1 name2 name3 . foo)
```

it specifies a file whose name begins with the components name1.name2.name3, ends with the component foo, and may also have any number of components in between. For example, ignoring directories, in a system such as Multics where the namestring consists of the file-name components from the namelist concatenated together with periods, the namelist

```
(-- moe larry . mung)
```

could specify all of the following file names:

```
moe.larry.mung
moe.larry.curly.mung
moe.larry.foo.bar.blech.mung
```

and so on. This form of namelist can be used to apply what is sometimes

called a "default extension."

A namelist can also have a dotted star, that is it can be in the form

```
(devdir -names-  .  *)
```

This specifies a file whose name begins with the components -names-, and may have zero or more components following those. Thus

```
(-- mung bung  .  *)
```

means any of these file names:

```
mung.bung mung.bung.lung mung.bung.foo.goo.zoo
```

The process of applying defaults to file names consists of "merging" two (or more) namelists into a single namelist, where one of the namelists is a user-supplied file specifier and the other is a set of defaults. For this purpose, the function `mergef` is supplied. `Mergef` is also used by `openi` and `openo` when they combine their argument with the default namelist to get the namelist of the file being opened.

`mergef` LSUBR 2 or more args

`mergef` is used for applying defaults to file specifiers, which may be namelists or namestrings.

(`mergef` *x y*) returns a namelist obtained by selecting components from *x* and *y*, which are converted to namelists. Where a component of *x* is *, the corresponding component of *y* is selected. It is an error if *y* is not that long. When a component of *x* is not *, it is selected and the corresponding component of *y* is skipped. If *y* ends with a dotted atom other than a * this atom is added to the end of the namelist if it is not already there. The same applies if *x* ends with a dotted atom. If *x* ends with a dotted *, the rest of *y* is copied over.

(`mergef` *x* `nil`) strips off the last component of *x*.

(`mergef` *w x y z*)
  is equivalent to
(`mergef` (`mergef` (`mergef` *w x*) *y*) *z*).

13.4 - Requests to the Operating System

13.4.1 - Manipulating the Terminal

tty VARIABLE

The value of the atom tty is initially set to a number describing the type of terminal being used. The values presently defined are:

0	normal terminal with no special capabilities
1,2	datapoint
3	Imlac
4	ARDS
5	pdp-11 TV

Except in the ITS implementation, tty will generally always be 0.

cursorpos LSUBR 0 to 2 args

The cursorpos function is used to manipulate the cursor on those display terminals which are similar to the Datapoint in that they show exactly one character at each position on the screen and can change each of these characters separately.

With no arguments it returns the dotted pair (line . column), where line is the line number, starting from 0 at the top of the screen, and column is the column position, starting from 0 at the left edge of the screen. If the terminal being used is not a display terminal with this type of cursor, nil is returned instead.

With two arguments, (cursorpos *line column*) moves the display cursor to the indicated position and returns t if it was successful, or nil if the terminal was incapable of doing this. Either of the two arguments may be nil, indicating that that coordinate should not be altered.

With one argument, cursorpos executes a number of special control operations. The argument must be a character object chosen from the following list:

F	move one space to the right
B	move one space to the left
D	move down by one line
U	move up by one line
C	clear the screen
T	go to top left corner of screen
Z	go to bottom left corner of screen
E	erase contents of screen after current point
]	erase from current point to end of line

X delete character to left

listen SUBR no args (listen <tty>) — also considers buffered chars within LISP

(listen) returns a fixnum which is non-zero if there is any input that has been typed in on the terminal but has not yet been read. In the ITS implementation it also waits for all terminal output and cursor motion to be completed.

13.4.2 - File System Operations

deletef SUBR 1 arg

(deletef *x*), where *x* is a namelist, a namestring, or a file object, deletes the file specified by *x*. The return value is the namelist of the file actually deleted, i.e. *x* mergef'ed over the defaults. Examples:

In the Multics implementation,
```
(deletef "foo.bar") => (>udd>proj>user>junk foo bar)
```

In the ITS implementation,
```
(deletef "foo bar") => ((ml loser) foo bar)
```

rename SUBR 2 args

(rename *x* *y*), where *x* and *y* are namelists, namestrings, or file objects, renames the file specified by (mergef *x* (names nil)) to the name specified by (mergef *y* *x* (names nil)). The directory part of *y* is ignored; a file may not be renamed onto a different device or directory. The return value is the namelist of the new name of the file.
Examples:

In the Multics implementation,
```
(rename "foo.baz" "*.bar") => (>udd>Bar>Foo foo bar)
```
and renames >udd>Bar>Foo>foo.baz to foo.bar.

In the ITS implementation,
```
(rename "foo baz" "* bar") => ((ml loser) foo bar)
```
and renames ml:loser;foo baz to foo bar.

`allfiles` SUBR 1 arg

(`allfiles` *x*), where *x* is a namelist, returns a list of namelists which are precise; i.e. they do not contain any stars or dotted parts. These are the namelists for all the files in the file system which match the namelist *x*. Whatever search rules are customary in the particular operating system are used.

`allfiles` with a precise namelist as an argument can be used as a predicate to determine whether or not a file exists.

The argument to `allfiles` may also be a namestring or a file object.

`clear-input` SUBR 1

(`clear-input` *x*), where *x* is a file or `nil` meaning the terminal, causes any input that has been received from the device but has not yet been read to be thrown away, if that makes sense for the particular device involved.

`force-output` SUBR 1

(`force-output` *x*), where *x* is a file or `nil` meaning the terminal, causes any buffered output to be immediately sent to the file, if that makes sense for the particular device involved.

13.4.3 - Random Access to Files

`filepos` LSUBR 1 or 2 args

(`filepos` *x*), where *x* is a file object open for input, returns the current character position within the file as a fixnum. The beginning of the file is 0.

(`filepos` *x* *n*), where *x* is a file object open for input and *n* is a non-negative fixnum, resets the character position of the file to position specified by *n*. It is an error if this position does not lie within the file or if the file is not randomly accessible. *n* is returned.

13.5 - The Old "Uread" I/O System

The functions uread, uwrite, ufile, ukill, and crunit are part of an older LISP I/O system. They are retained for compatibility. Various "status" functions are also part of this older I/O system. (See section 12.7)

These five functions name files in a different way from the other I/O functions. A file is named by a 4-list,

```
(name1 name2 dev dir)
```

Name1 and name2 together make up the "filename," dev is the "device," and dir is the "directory." In the ITS implementation of MACLISP, these go together to make up the ITS file name:

```
DEV: DIR; NAME1 NAME2
```

In the DEC-10 implementation, dev is the device name, name1 is the file name, name2 is the extension, and dir is a list of two fixnums, the project number and the programmer number. Thus the 4-list

```
(name1 ext dev (proj prog))
```

represents the file

```
dev:name1.ext[proj,prog]
```

In the Multics implementation, dev is ignored and dir is the directory pathname. The entry-name is name1.name2. Thus the Multics filename is:

```
dir>name1.name2
```

These five functions maintain their own set of defaults, which are updated every time one of these functions is called, so that the defaults correspond to the last file that was used. The defaults may be examined with (status crfile), which returns the first two elements of the default 4-list, and (status crunit) which returns the last two.

It is not necessary to specify all four parts of the 4-list when one of these five functions is used. If the list contains less than four elements, the elements at the end which were dropped are supplied from the defaults.

These functions are fsubrs which do not evaluate their arguments. They may be applied to a 4-list, e.g.

```
(apply 'uread (cons filename '(stuff dsk macsym)))
```

or they may be called with the four-list as four arguments, which is convenient when calling them from top level. e.g.:

```
(uread foo bar dsk crock)
```

uread FSUBR

This function selects an input file. The argument list is a 4-list as described above. The specified file is made the default input source. Note that the ^q switch must be turned on before input will be automatically taken from this file.

uwrite FSUBR

uwrite opens an output file. When done with this file, ufile must be used to close it and give it a name. The arguments are the last two elements of a 4-list, specifying the device and directory on which the file is to be written. The first two parts of the 4-list are not specified until the file is ufile'd.

ufile FSUBR

(ufile *name1* *name2*) closes the uwrite output file and gives it the name *name1.name2*. The arguments are not evaluated. (ufile) takes *name1* and *name2* from the defaults.

crunit FSUBR

(crunit) returns the current device and directory.

(crunit *dev* *dir*) sets the device and directory and returns it. The arguments are not evaluated. Example:

```
(crunit) => (dsk >udd>Bar>Foo>subdirectory)
```

ukill FSUBR

(ukill -args-), where -args- are as for uread, deletes the specified file.

uread VARIABLE

The value of uread is a file object being used for input initiated by the uread function, or nil if no file is currently being uread.

`uwrite` VARIABLE

The value of `uwrite` is a file object being used for output initiated by the `uwrite` function, or `nil` if no file is currently being uwritten.

There are also some status/sstatus functions associated with these. These are `(status crunit)`, `(status crfile)`, `(status uread)`, and `(status uwrite)`.

13.6 - Advanced Use of the Reader

13.6.1 - The Obarray

obarray VARIABLE & ARRAY

The value of obarray is an array which is a table of known atomic symbols - when an atomic symbol is read in it is "interned" on this obarray, that is made eq to any atomic symbols with the same pname that were previously read in. If an atomic symbol, such as one created by (gensym), is not in this table an atomic symbol read in with the same pname will not be the same - there will be two separate copies.

The obarray may be manipulated by the functions remob and intern. A new obarray may be created by using the makoblist function. The atom obarray may be setq'ed or lambda-bound to different obarrays at different times, which allows multiple sets of atomic symbols to be kept seperate - you can have different atomic symbols with the same pname interned on different obarrays at the same time.

Note that the value of obarray is not an atomic symbol which names an array, but the array-object itself, as obtained by (get 'name 'array).

The array property of obarray is the same array as its initial value.

Example of the use of multiple obarrays:

```
(setq private-obarray
      (get (makoblist 'private-obarray) 'array))
          ;make another obarray.

((lambda (obarray) (read)) private-obarray)
          ;read using atoms on private obarray
          ;instead of regular one
```

makoblist SUBR 1 arg

(makoblist nil) returns a list of lists of atomic symbols which is a representation of the current obarray.

(makoblist *a*) gives the atom *a* an array property of a copy of the current obarray, i.e. it makes *a* an obarray which is a copy of the current one. The value returned is the argument; in this case, *a*.

See also the functions remob and intern.

13.6.2 - The Readtable

The readtable is a special table used to control the reader. It has an entry for each character, containing 26. syntax bits and a "chtran" character translation code. The "chtran" allows characters to be translated when they are put into pnames that are being read in. The syntax bits are used by the reader to determine the significance and syntactic meaning of each character it encounters. A table of the meanings of the bits follows. Initially the syntax bits are set to give the standard LISP meanings for all the characters, but the user can change them to make the read function usable as a lexical analyzer for a wide variety of input formats or languages. It is also possible to have several readtables containing different syntax and to switch from one to the other by binding the atom readtable.

Table of Syntax Bits

Octal Value	Meaning
1	alphabetic: A-Z and a-z
2	"extended alphabetic" - used for characters which are not letters, but which are to be treated the same as letters. E.g. ":" in the standard syntax table.
4	digit: 0-9
40	This is the "alternate meaning" bit. When added in it alters the meaning of other bits. For example, 10 is plus sign but 10+40 is minus sign.
10	plus sign.
50	minus sign.
20	fixed-point-scaling number modifier character. In the standard readtable "^" has this syntax. An example of use is 13^2, which is the same as 1300.
60	fixed point left-shift number modifier. In the standard readtable "_" has this syntax. An example of its use is 7_5, which is 7 shifted left 5 bits, or 340 octal.
100	Indicates that the character should be "slashified" by print,

prinl, and explode if it appears in a pname but is not the first character in that pname.

200	Decimal point. Embedded in a number it indicates floating point. At the end of a number it indicates that the number is to be interpreted in decimal regardless of the value of ibase. Note that the decimal point need not be the same as the dotted-pair dot.
400	The character will be "slashified" by print, prinl, and explode if it is the first character in a pname. Thus special characters which need to be slashified usually have 500 = 100+400 in their syntax bits.
1000	Indicates that this character is the "rubout" character. (in the pdp-10 implementations only.)
2000	The slashifier character, which is used as an escape. It makes special characters like space and parentheses look like letters. Normally "/" has this syntax. If this character appears in input to read or readlist, the following character is taken to have a syntax of 000000002 and its chtran is not used. This allows it to be used in a pname even if it is a special character.
4000	indicates a macro character. This bit should not be set explicitly, only by using (setsyntax c 'macro f) or (setsyntax c 'splicing f). When this character is seen in input to read or readlist, an associated function is called and the value returned by the function is assumed to have been read.
4040	same as 4000 except the macro is "splicing." That is, the associated function returns not an object to be inserted in the list being read, but a list to be spliced (nconc'ed) into the list being read. Splicing macros at top level (not inside a list) have their values ignored by the reader. The same applies to splicing macros that return nil, as this is the empty list.
10000	Right parenthesis.
10040	Right super-parenthesis. A super parenthesis cancels out all left parentheses back to the beginning of the object or to a left super-parenthesis.
20000	The dotted-pair dot.
40000	Left parenthesis.
40040	Left super-parenthesis. The chtran of a left super-parenthesis must be set to the ascii code for the corresponding right super-parenthesis so that the reader can check for proper matching of super parentheses.
100000	A blank, i.e. a character which delimits an atom or a number but

is otherwise ignored. In the standard read table, space, tab, comma, and newline have this syntax.

600000 "Single-character object." This character begins, ends, and is an atomic symbol. The difference between a single character object and a letter is that a single character object need not be delimited by spaces or any other special chcracters. If ":" had this syntax then "(:::)" would read as a list of length three, unlike "(aaa)" which reads as a list of length 1. In the standard readtable no characters have this syntax.

400000 A character with this syntax bit turned on ends pnames and numbers. In the standard readtable all the special characters such as space, parentheses, and dot have this syntax.

1000000 This character is the exponent introducer (e) for floating-point numbers.

1000040 This character is the string quote, which begins and ends character strings (usually a " sign.)

2000000 This character causes vertical motion (newline, newpage.)

4000000 In the pdp-10 implementation, this bit indicates a "force feed" character. This is a character which causes LISP to "wake up" and read typed input. Initially such characters as space and right parenthesis have this syntax. In the Multics implementation newline is always a forcefeed character and no other character can be a force feed.

The syntax for most characters is a combination of several of these bits. Here is a table of the syntax codes assigned to various characters in the standard readtable:

worthless control characters
000400500
backspace 000000002 (to allow underlining in pnames)
space, tab, newline, newpage (the "white space" characters)
000500500 (002500500 for nl and np)

"	001400540
'	000404500
(	000440500
)	000410500
+	000000410
-	000000450
,	000500500
.	000420700
/	000402500
0-9	000000404
;	000404540
A-Z	000000001
E	001000001

^	000000022
_	000000062
a-z	000000001
e	001000001

The remaining special characters have 000002 syntax.

setsyntax SUBR 3 args

(setsyntax *c s x*) adjusts the syntax of the character *c* in the readtable. *c* can be a fixnum which is the ascii code for a character, or it can be a character object or a string one character long. *s* can be nil, meaning don't change the syntax of *c*, or a fixnum, meaning set the syntax bits for *c* to that fixnum, or the atom single, meaning set the syntax of *c* to the syntax for single-character objects, or the atom macro, meaning set the syntax of *c* to be a macro character in which case *x* is the function for the macro, or the atom splicing which is like macro except it makes a splicing macro, or a character object or string in which case the standard initial syntax for this character is used. *x* can be a fixnum to which *c*'s chtran should be set, or a character object or one-character string to which *c*'s chtran should be set, or nil meaning don't change *c*'s chtran, or a function if the second argument was splicing or macro.

If *c* is a macro character, it is changed back to its standard syntax and chtran before anything else is done, unless it is a macro in the standard readtable, in which case its syntax is set to 502 (extended alphabetic) and its chtran is set to itself.

setsyntax always returns t.

readtable VARIABLE & ARRAY

The value of readtable is an array which contains tables used by the reader to determine the meaning of input characters. This array may be manipulated using the functions setsyntax, status, and sstatus. Multiple readtables may be constructed by using the makreadtable function, and the atom readtable may be lambda-bound or setq'ed to one or another of these readtables.

Note that the value of readtable is not an atomic symbol which names an array, but the array-object itself, which is the result of (get 'x 'array) if x names an array.

The array property of readtable is the same as its initial value, which is the read table with the standard meanings for all characters.

`makreadtable`   SUBR 1 arg

(makreadtable *a*) gives the atom *a* an array property which is a copy of the current readtable, a table which is used by the reader to determine the syntactic properties of characters.

(makreadtable nil) is the same except that the array property is hung on a (gensym)'ed atom.

(makreadtable t) is like (makreadtable nil) except that a copy of the initial readtable is used rather than a copy of the current readtable. The initial readtable has the standard meanings for all the characters, e.g. (and) delimit lists, ' is used for quoting, etc.

makreadtable returns the atomic symbol on which it has hung the array property. To switch to this readtable, evaluate

```
(setq readtable (get (makreadtable whatever) 'array))
```

See also the functions status and sstatus.

13.7 - Control of Printer Formatting

This section describes how the functions print, prin1, princ, explode, exploden, and explodec convert LISP objects to strings of characters so that they can be printed out in readable form.

Atomic symbols are represented by their "pnames," which are the strings of characters by which they were originally typed in. When special characters appear in a pname, they are sometimes "slashified," i.e. preceded by a "/" to remove their special meaning. See the descriptions of the individual functions to see which slashify and which don't.

Strings are printed out as the characters they contain. Those functions which slashify put quotes (") around strings to distinguish them from atomic symbols and to indicate that any special characters, such as space or period, they contain are not to be considered to have their special meanings.

Flonums are printed out in decimal radix, with an embedded decimal point. If the magnitude of the number is outside of a certain range, a trailing exponent delimited by an "e" is printed.

Fixnums are printed in the radix specified by the variable base. Negative fixnums have a preceding "-" sign. If base is ten and the variable *nopoint is nil, which it is unless changed by the user, fixnums will be printed with a trailing decimal point. If the base is greater than ten, letters will be used as digits.

Fixnums with many trailing zeros are made more legible by use of the "_", or left-shift, operator, unless (sstatus _ nil) is done. The "_" character is used because on certain formerly-used terminals it was printed as a leftward arrow. The use of the "_" can be described by example: in octal radix, "12345000000" would be printed instead as "12345_18.".

Bignums are printed in much the same format as fixnums.

*nopoint SWITCH

If the value of *nopoint is nil, trailing decimal points are printed when numbers are printed out in base ten. This allows these numbers to be read back in correctly even if ibase is not ten. If *nopoint is non-nil, the trailing decimal points are suppressed. The initial value of *nopoint is nil.

base VARIABLE

The value of base is a number which is the radix in which numbers are to be printed. The initial value of base is 8.

Lists or "conses" are represented using the parentheses notation that has been used throughout this manual. A dotted pair whose cdr is nil is printed as a list but other dotted pairs are printed using the dot. Thus (cons 'a nil) prints as (a), while (cons 'a 'b) prints as (a . b). (list 'a 'b 'c) prints as (a b c) rather than (a . (b . (c . nil))).

Other types, such as arrays, subrs, and files, have no proper printed representation. They are printed as a "#" sign followed by some string that has internal meaning, such as an octal number.

Most files have a limit on the number of characters that may be printed on a single line when output is done to these files. For example, on the user terminal the maximum number of characters per line is determined by the width of the platen or display screen. This limit is called the file's "linel." When a file-object is created, its linel is set to a value appropriate for the device on which the file resides.

Each file also has another number associated with it, called its "charpos." The charpos is the horizontal position, starting at 0 at the left margin. If no backspaces or tabs are used it is the number of characters that have been printed so far on the current line. As characters are sent to the file, charpos increases for printing characters or is adjusted appropriately for format effectors such as carriage return or backspace. When charpos exceeds linel, an automatic newline is provided by the output functions, such as print, to ensure that the line being sent to the file is not longer than the file's linel. This feature can be turned off by use of the (sstatus terpri) function. Note that the linel is not an absolute limit since some implementations will not break atoms across lines, so that a particularly long atom near the right margin could result in a line longer than linel.

Some other attributes of a file are its linenum, pagel, and pagenum. The linenum is the line number, starting with 0 at the top of a page. The pagel is the number of lines per page. The pagenum is the page number, starting with zero at the beginning of the file.

The following functions can be used to examine or modify the attributes of files:

linel LSUBR 1 or 2 args

(linel *f*), where *f* is an output file, gets the number of characters per line on *f*. Lines output to *f* that exceed this length get an extra newline inserted at the next break between atoms. If the linel = 0, this feature is suppressed.

(linel nil) gets the linel of the terminal.

(linel *f* *n*) sets the file *f*'s linel to the fixnum *n*.

(linel nil *n*) sets the terminal's linel to the fixnum *n*.

`charpos` LSUBR 1 or 2 args

(charpos *f*), where *f* is a file or nil meaning the terminal, returns the current character position of *f*, with 0 being the left margin.

(charpos *f n*) sets the charpos of *f* to the fixnum *n*. This does not move a cursor or anything of that sort.

`chrct` LSUBR 1 or 2 args

This is an older, now obsolete, version of charpos.

(chrct *x*), where *x* is an output file or nil meaning the terminal, returns the number of character positions left on the line being output to *x*.

(chrct *x y*), where *x* is a file and *y* is a fixnum, sets *x*'s chrct to *y* and returns *y*.

`pagel` LSUBR 1 or 2 args

(pagel *f*), where *f* is a file or nil meaning the terminal, returns the number of lines per page of the file *f*.

(pagel *f n*) sets the number of lines per page of the file *f* to the fixnum *n*.

`linenum` LSUBR 1 or 2 args

(linenum *f*), where *f* is a file or nil meaning the terminal, returns the current line number of *f*, with 0 being the top of the page.

(linenum *f n*) sets the line number of *f* to the fixnum *n*. Note that this does not causes any physical motion, it simply changes the number.

`pagenum` LSUBR 1 or 2 args

(pagenum *f*), where *f* is a file or nil meaning the terminal, returns the current page number of *f*, which is 0 when the file is first opened.

(pagenum *f n*) sets the page number of the file *f* to the fixnum *n*.

The printing of large lists can be limited by use of the variables

prinlevel and prinlength. If these variables are nil, they have no effect, but if they are set to fixnum values they take effect as follows: prinlevel specifies the maximum depth of nested parentheses that will be printed. If this depth of nesting is exceeded, a sharp sign (#) will be printed and the list structure below that depth will be omitted. prinlength specifies the maximum number of list elements (atoms or sub-lists) that will be printed in any one list. If more than this number need to be printed, the excess will be omitted and 3 dots (...) will be printed to indicate the omission. These features operate under the control of some abbreviation control bits set by (sstatus abbreviate). (sstatus abbreviate 1) enables it for print, prinl, etc. when they output to files. (sstatus abbreviate 2) enables it for flatsize, flatc, explode, etc. (sstatus abbreviate 3) enables it for both. (sstatus abbreviate t) enables it for everything (currently the same as 3. (sstatus abbreviate nil) or (sstatus abbreviate 0) turns it off. Note that abbreviation is always in effect for the terminal. The only way to turn it off is to setq prinlevel and prinlength to nil.

prinlevel VARIABLE

prinlevel can be set to the maximum number of nested lists that will be printed before the printer will give up and just put a "#". If it is nil, which it is initially, any number of nested lists can be printed. Otherwise, the value of prinlevel must be a fixnum. The effect of prinlevel is under the control of (sstatus abbreviate).

prinlength VARIABLE

prinlength can be set to the maximum number of elements of a list that will be printed before the printer will give up and just put "...". If it is nil, which it is initially, any length list can be printed. Otherwise, the value of prinlength must be a fixnum. The effect of prinlength is under the control of (sstatus abbreviate).

endpagefn LSUBR 1 or 2 args

(endpagefn *f*), where *f* is an output file, returns the end of page function of *f*. This is a function which is invoked whenever the file is advanced to a new page. nil is returned if *f* has no endpagefn.

(endpagefn *f* *x*) sets the endpagefn of the file *f* to the functional form *x*. If *x* is nil *f*'s endpagefn is removed.

Note that if *f* is nil it means set the endpagefn of the terminal, which may or not be implemented depending on the implementation and the type of terminal.

13.8 - Input Format Expected by (read)

This section describes the forms of input which will be accepted by the read function and converted to LISP objects.

A string of digits, with an optional leading sign and trailing decimal point, is read as a fixnum, unless it is too large to fit in a fixnum, in which case it is read as a bignum. If a trailing decimal point is included, the number is converted in decimal radix. Otherwise the variable ibase specifies the radix. Initially it is eight. If ibase is greater than ten, and the number begins with a leading "+" or "-" sign, and (status +) is not nil, upper- or lower-case letters may be used as digits, with "a" being 10., "b" 11., etc. If (status +) is nil, the standard initial setting, this feature is turned off and what looked like a number with letters as digits would be read as an atomic symbol.

ibase VARIABLE

The value of ibase is a number which is the radix in which numbers will be read. The initial value of ibase is 8.

"Fixed point number modifier" characters may be used in fixnums or bignums. In the standard readtable these characters are "_" and "^". "mmm_nn" causes the number mmm to be shifted left nn bits. Note that nn is interpreted in the ibase radix unless a trailing decimal point is placed on it. If mmm is to be read in decimal, a trailing decimal point may be placed just before the "_". "mmm^nn" is read in as mmm followed by nn zeros, i.e. as mmm multiplied by the nn'th power of the input radix. nn must have a trailing decimal point if it is not to be interpreted in the ibase radix.

A string of digits with a leading or embedded decimal point, and/or an exponent introduced by "e" or "E", is read as a flonum. The number and the exponent may be optionally signed. The number and the exponent are interpreted in decimal radix regardless of ibase. The number is generally converted to binary and rounded to the equivalent of 6 to 8 decimal digits of precision, depending on the implementation.

A string of letters, numbers, and "extended alphabetic" characters represents an atomic symbol whose pname is the string, provided that it does not look like a number (e.g. all numeric characters.) A special character, such as a parenthesis, a period, or a space, may be included in an atomic symbol by preceding it with a slash. A slash itself is represented by two slashes. A string of digits may be made to represent an atomic symbol rather than a number by preceding one of the digits with a slash, to make it into an alphabetic character.

A parenthesized sequence of items such as atoms or parenthesized lists

is read in as a list. The items may be seperated by spaces or commas. If a dot appears, it must be between the last two items, and a dotted pair or a list ending in an atom other than nil is created. Where there would be ambiguity between a dotted-pair or a decimal point, the decimal-point interpretation will be chosen, so dotted-pair dots should be surrounded by spaces. Thus (1.2) is different from (1 . 2).

A sequence of characters enclosed in quotes (such as "foo") is read as a character string. If a quote is to be included in the string, two quotes must be written.

The readtable may be set up so that certain characters are "single character objects." These characters read in as atomic symbols whose pname is the single character, without the benefit of delimiting spaces or commas. In the standard readtable there are no single character objects.

Characters may be defined as macro characters. (See the setsyntax function.) When these characters are encountered by the reader, a special action defined by a LISP function is performed, unless, of course, the character is slashified. Two macro characters included in the standard readtable are ' and ;. 'x is equivalent to (quote x), similarly '(a b) is equivalent to (quote (a b)). The ; is used to introduce a comment: the semicolon and the rest of the line to the right of the semicolon are skipped over.

Note that the specific characters used in all the constructions defined above are only the initial default characters for these constructions. Any other characters may be substituted by changing the readtable. See the setsyntax function.

13.9 - "Moby I/O"

This section describes how to use some of the peculiar I/O devices present on the MIT A.I. Lab pdp-10.

The Display Slave

The Display Slave runs on the pdp-6, if it is available, otherwise on the pdp-10. It displays pictures on the 340 display under control of commands sent to it by certain LISP functions.

The following conventions are used in the descriptions of these functions:

x,y are assumed to be fixnum arguments to line drawing, point inserting, and other such functions. They represent coordinates.

n is a fixnum argument whose meaning is described under the particular functions which use it.

item is the numerical index or name of some display slave item, returned by `discreate`.

bright each item has a brightness level associated with it, ranging between 1 and 8. The default is 8.

scale each item has a scale, or magnification, factor associated with it. The scale factor ranges between 1 and 4. The default is 1. 2 doubles the length of drawn lines and the size of text; 3 quadruples; and 4 multiplies by 8. Text looks much nicer (on the 340) if drawn with a magnification of 2.

flag is an indicator, which if `nil` specifies that a given action is to be undone, or if non-`nil` specifies that the given action is to be done.

bsl is either `nil` indicating no change, or is a list (bright scale) specifying new values of these parameters for a given action.

All numbers in this section are octal unless followed by a decimal point, in which case they are decimal.

The display slave maintains a number of items, whose names are indicated by 'item' above. Each item has associated with it variables determining the brightness, scale, and visibility of points and lines in the item. Like the LOGO Turtle, we think of the item as having a "pen" which can be "down" so that a line is visible when the turtle is requested to move from one place to another, or "up" so that no visible mark is made.

For the functions which affect brightness, scale, or the penup status, 0 generally means no change. Functions which take an optional 'bsl' argument, namely disapoint, discuss, and disaline, will treat it as a

temporary setting for these values, and upon exit will restore them to their values prior to the call. The optional penup argument to disaline is similarly treated as temporary.

Arguments that are intended to specify coordinates on the screen for the functions disaline, disapoint, and discuss are interpreted in one of four ways depending on the slave variable "astate," which can be set by disini. "astate" is *not* a lisp variable.

0 relative mode - the point specified is in relation to the "home" of the item on which the function is acting.

1 absolute mode - x and y are interpreted as direct screen coordinates, modulo 1024., with (0,0) at the lower left-hand corner.

2 incremental mode - the coordinates specified are relative to the current position of the pen of the item which is being acted upon.

3 polar mode - like incremental, but the x and y arguments, which must be flonums instead of fixnums, are considered as the radius and angle respectively in a polar coordinate system centered on the current pen position, with zero degrees being horizontal and to the right.

NB: functions like discreate, dislocate, and dismotion, which specify an item's home, always take the coordinates of the home in absolute mode.

To emphasize that the interpretation of *x* and *y* is controlled by astate, we will write astate(*x*,*y*) to mean the point specified by *x* and *y*.

`discreate` LSUBR 0 or 2 args

(`discreate` *x y*) creates a new display item with home at (*x*,*y*) on the screen. (`discreate`) creates one with home at (0,0). `discreate` returns the item number of the newly-created item, by which it may be referred to in later calls.

`disini` LSUBR 0 or 1 args

`disini` seizes and initializes the slave. If the user already has the slave (this is not the first `disini`), it is re-initialized. astate is set to the argument if there is one, provided it is 0, 1, 2, or 3. Otherwise astate is not changed. Initially astate is 0. The previous value of astate is returned.

`display` SUBR 2 args

(`display` *item flag*) makes the *item* visible or not depending on whether *flag* is `t` or `nil`.

`disflush` LSUBR 0 or more args

(`disflush`) flushes the slave. (`disflush` *item item* ... *item*) flushes the indicated items, i.e. deletes them from the slave's memory.

`dislocate` SUBR 3 args

(`dislocate` *item x y*) moves the *item*'s home to (x,y).

`disblink` SUBR 2 args

(`disblink` *item flag*) makes *item* blink if *flag* is non-`nil`, stop blinking if *flag* is `nil`.

`discopy` SUBR 1 arg

(`discopy` *item*) makes a copy of *item*, and returns the new item, which has its home at the same location.

`dismark` SUBR 2 args

(`dismark` *item n*) if $n = 0$, removes marker from *item*. If $n < 0$, inserts standard marker on *item*. If $n > 0$, inserts display-item *n* as marker on *item*.

`discribe` SUBR 1 arg

(`discribe` *item*) returns a list of the parameters of *item*:

(home-x, home-y, pen-pos-x, pen-pos-y, bright, scale, penup, marker)

`dischange` SUBR 3 args

(`dischange` ***item bright scale***) adds ***bright*** and ***scale*** to the corresponding variables of ***item***.

`dislink` SUBR 3 args

(`dislink` ***item***-1 ***item***-2 ***flag***) links or unlinks ***item***-1 to ***item***-2 (links if ***flag*** non-`nil`, unlinks if ***flag*** `nil`.) ***item***-2 is the "inferior" of ***item***-1, and will be `dislocated`, `dischanged`, `displayed`, and `disblinked` whenever ***item***-1 is.

`dislist` LSUBR 0 or 1 args

(`dislist`) returns a list of all items "on display," that is made visible by (`display` ***item*** `t`).
(`dislist` ***item***) returns a list of all inferiors of ***item***.

`diset` SUBR 3 args

(`diset` ***item n bsl***) sets the values for penup, brightness, and scale for the item. If ***n*** is -1, put pen down. If ***n*** is +1, lift pen. If ***n*** is 0, leave pen alone. Set bright and scale from ***bsl***, as described above.

`disaline` LSUBR 3 to 5 args

(`disaline` ***item x y bsl n***) sets penup, bright, and scale from ***bsl*** and ***n*** as in `diset`, then moves pen position to astate(x,y), leaving a visible line if the pen is down, and restore the item's penup, bright, and scale.
The forms (`disaline` ***item x y***), (`disaline` ***item x y n***), and (`disaline` ***item x y bsl***) are also allowed. The unspecified parameters are left unchanged.

`disapoint` LSUBR 2 or 3 args

(`disapoint` ***x y bsl***) displays a point at astate(x,y) as part of ***item***. ***bsl*** is interpreted as by `disaline`, as a temporary setting of bright and scale. It may be omitted.

discuss LSUBR 4 or 5 args

(discuss *item x y text bsl*) processes *bsl* as disapoint does, then inserts the characters of *text*, as if princ'ed, into the *item* beginning at the point astate(x,y). *bsl* may be omitted.

dismotion SUBR 4 args

(dismotion *item x y speed*) causes *item* to be slowly dislocate'ed so that its home eventually becomes (x,y). If either x or y is negative the *item* is placed under control of spacewar console 1. The button returns control to the tty. [???] The argument *speed* is an inverse measure of the speed at which the *item* will move. *speed* = 0 is the maximum.

disgorge SUBR 1 arg

(disgorge *item*) creates a (gensym) atom, gives it an array property, and fills the array with the internal display code of *item*.

disgobble SUBR 1 arg

(disgobble *array-name*) takes the array named array-name and generates a display item from the internal display code in the array.

Examples:

A subroutine to draw a light box with a medium point inside it at the center of the screen. A description of the item is returned.

```
((lambda (oastate b)
   (disaline b -100 -100 1)   ;go to lower left corner of box
   (diset b 0 (list 3 boxscale));set scale from global variable
                              ;set bright but don't change penup
   (disaline b 0 200)         ;draw box in incremental mode
   (disaline b 200 0)
   (disaline b 0 -200)
   (disaline b -200 0)
   (disini 0)                 ;go to relative mode to
   (disapoint b 0 0 '(6 0))   ;draw the point
   (disini oastate)           ;restore astate
   (discribe b))              ;return value.
 (disini 2)                   ;enter with astate 2
 (discreate 1000 1000))       ;and b set to this item.
```

To add some text on top of the box, assuming astate = 0 and that b is the item as above:

```
(discuss b -200 207 "here is the box - see the box" '(6 2))
```

To move the box b right 100 units:

```
(setq foo (discribe b))
(setq foo (list (car foo) (cadr foo)))
(dislocate b (+ 100 (car foo)) (cadr foo))
```

To put a cross where the pen is now, and some text where it used to be before it was moved:

```
(dismark b -1)
(discuss b (caddr foo) (cadddr foo) "turtle slept here"))
```

To brighten the box and point (but text is already brightest, so it does not change):

```
(dischange b 2 0)
```

To get rid of the box:

```
(disflush b)
```

To get rid of the slave:

```
(disflush)
```

The display slave is also available in the Multics implementation of MACLISP, in a somewhat different form. The Multics Graphics System is used so the "display" can be any device. Normally it is the terminal, but it can be directed elsewhere by attaching the stream "graphic_output," before using the display slave.

The Multics display slave is not part of the initial LISP environment. It must be loaded in. (As of this writing, 21 January 1974, the Multics display slave is actually not yet available. However it is expected to be finished soon.)

The Multics display slave does not implement brightness or scale. Blinking is simulated by the use of dotted lines. `dismotion` is not implemented. The graphic data does not actually appear until `(disgo)` is evaluated. `(diserase)` should be called if an item is changed or removed, rather than simply added, so that the display will be redrawn from the beginning the next time `(disgo)` is used. The Multics display slave accepts both fixnums and flonums as coordinates.

The display slave consists of a set of LISP functions which are autoloaded in when `disini` is first called.

Arms, Hands, and Eyes

TO BE SUPPLIED

14 - Compilation

LISP programs can be compiled into machine code. This representation of a program is more compact than the interpreted list-structure representation, and it can be executed much more quickly. However, a price must be paid for these benefits. It is not as easy to intervene in the execution of compiled programs as it is with interpreted programs. Thus most LISP programs should not be compiled until after they have been debugged.

In addition, not all LISP programs can be compiled. There are certain things which can be done with the interpreter that cannot be effectively compiled. These include indiscriminate use of the functions `eval` and `apply`, especially with pdl-pointer arguments; "nonlocal" use of the `go` and `return` functions; functions which modify themselves. Also there are a number of functions which detect illegal arguments when they are called interpretively but not when a call to them is compiled - therefore erroneous compiled programs can damage the LISP environment and can cause strange errors to occur - be forewarned. However, most "normal" programs are compilable.

14.1 - Peculiarities of the Compiler

Some operations are compiled in such a way that they will behave somewhat differently than they did when they were interpreted. It is sometimes necessary to make a "declaration" in order to obtain the desired behavior. This is explained in section 14.2.

14.1.1 - Variables

In the interpreter "variables" are implemented as atomic symbols which possess shallow-bound value cells. The continual manipulation of value cells would decrease the efficiency of compiled code, so the compiler defines two types of variables: "special variables" and "local variables." Special variables are identical to variables in the interpreter.

Local variables are more like the variables in commonly-used algebraic programming languages such as Algol or PL/I. A local variable has no associated atomic symbol, thus it can only be referred to from the function that possesses it. The compiler creates local variables for prog-variables, do-variables, and lambda-variables, unless directed otherwise. The compiled code stores local variables in machine registers or in locations within a stack.

The principal difference between local variables and special variables is in the way a binding of a variable is compiled. (A binding has to be compiled when a prog-, do-, or lambda-expression is compiled, and for the entry to a function which has lambda-variables to be bound to its arguments.) If the variable to be bound has been declared to be special, the binding is compiled as code to imitate the way the interpreter binds variables: the value of the atomic symbol is saved and a new value is stored into its value cell. If the variable to be bound has not been declared special, the binding is compiled as the *declaration* of a new local variable. Code is generated to store the value to which the variable is to be bound into the register or stack-location assigned to the new local variable.

Although a special variable is associated with an atomic symbol which is the name of the variable, the name of a local variable appears only in the input file - in compiled code there is no connection between local variables and atomic symbols. Because this is so, a local variable in one function may not be used as a "free variable" in another function since there is no way for the location of the variable to be communicated between the two functions.

When the usage of a variable in a program to be compiled does not conform to these rules, i.e. it is somewhere used as a "free variable," the variable must be declared special. There are two common cases in which this occurs. One is where a "global" variable is being used, i.e. a variable which is setq'ed by many functions but is never bound. The other

is where two functions cooperate, one binding a variable and then calling the other one which uses that variable as a free variable.

14.1.2 - In-line Coding

Another difference between the compiler and the interpreter is "in-line coding," also called "open coding." When a form such as (and (foo x) (bar)) is evaluated by the interpreter, the built-in function and is called and it performs the desired operation. But to compile this form as a call to the function and with list-structure arguments derived from (foo x) and (bar) would negate much of the advantage of compiling. Instead the compiler recognizes and as part of the LISP language and compiles machine code to carry out the intent of (and (foo x) (bar)) without actually calling an and function. This code might look like:

```
pick up value of variable x
call function foo
is the result nil?
if yes, the value of the and is nil
if no, call the function bar
the result of the and is what bar returned.
```

This "in-line coding" is done for all "special forms" (cond, prog, and, errset, setq, etc.), thus compiled code will usually not call any of the built in fsubrs.

Another difference between the compiler and the interpreter has to do with arithmetic operations. Most computers on which MACLISP is implemented have special instructions for performing all the common arithmetic operations. The MACLISP compiler contains a "number compiler" feature which allows the LISP arithmetic functions to be "in-line coded" using these instructions.

A problem arises here because of the generality of the MACLISP arithmetic functions, such as plus, which are equally at home with fixnums, flonums, and bignums. Most present-day computers are not this versatile in their arithmetic instructions, which would preclude open-coding of plus. There are two ways out of this problem: one is to use the special purpose functions which only work with one kind of number. For example, if you are using plus but actually you are only working with fixnums, use + instead. The compiler can compile (+ a b c) to use the machine's fixnum-addition instruction. The second solution is to write (plus a b c) but tell the compiler that the values of the variables a, b, and c can never be anything but fixnums. This is done by means of the "number declarations" which are described in section 14.2.

Another problem that can arise in in connection with the in-line coding of arithmetic operations is that the LISP representation of numbers and the machine representation of numbers may not be the same. Of course, this depends on the particular implementation. If these two representations are

different, the compiler would store variables which were local and declared to be numeric-only in the machine form rather than the LISP form. This could result in compilation of poor code which frequently converts number representations and in various other problems. Compilers which have this problem provide a (closed t) declaration which inhibits open coding of arithmetic operations.

14.1.3 - Function Calling

Another property of compiled code that should be understood is the way functions are called. In the interpreter function calling consists of searching the property list of the called function for a functional property (if it is an atomic symbol) and then recursively evaluating the body of the function if it is an expr, or transferring control to the function if it is a subr. In compiled code function calling is designed according to the belief that most of the functions called by compiled code will be machine executable, i.e. "subrs:" other compiled functions, or builtin functions, and only infrequently will compiled code call an interpreted function. Therefore a calling mechanism is used which provides for efficient transfer between machine-executable functions without constant searching of property lists. This mechanism is called the "uuo link" mechanism for historical reasons.

When a compiled function is first loaded into the environment, it has a uuo link for each function it will call. This uuo link contains information proclaiming that it is "unsnapped" and giving the name of the function to be called, which is an atomic symbol. The first time a call is made through such a uuo link, the fact that it is "unsnapped" is recognized and a special linking routine is entered. This routine searches the property list of the function to be called, looking for a functional property in just the same way as the interpreter would. If the function turns out to be an expr, or is undefined, the interpreter is used to apply the function and the result is given back to the compiled code. The link is left "unsnapped" so that every time this function is called the interpreter will be invoked to interpret its definition.

If the function being called is machine executable (a subr), the link is "snapped." Exactly what this means is implementation dependent but the effect is that from now on whenever a call is made through this uuo link control will be transferred directly to the called function using the subroutine-calling instruction of the machine, and neither the linking routine nor the interpreter will be called.

There is a flag which can be set so that links will not be snapped even if they go to a function which is machine executable. This flag is the value of the atomic symbol nouuo. There is also a function, (sstatus uuolinks), which unsnaps all the links in the environment. These facilities are used in circumstances such as when a compiled function is redefined.

In the pdp-10 implementation a uuo link is implemented as an instruction which is executed when a call is to be made through the link. An "unsnapped" link consists of a special instruction, "uuo", which causes the LISP linking routine in the interpreter to be called. The address field of the uuo points to the atomic symbol which names the function to be called. The operation code and accumulator fields indicate the type of call and number of arguments. When the link is snapped the uuo instruction is replaced with a "pushj" instruction, which is the machine instruction for calling subroutines.

In the Multics implementation, a uuo link is implemented as a pointer. To call through this link a "tspbp" instruction indirect through the pointer is used. An unsnapped link points at the linking subroutine and various fields in the pointer, left unused by the machine, indicate the type of call, number of arguments, and the atomic symbol which names the function. When the link is snapped the pointer is changed to point at the first instruction of the called function.

Before a function can be used it must be made known in the LISP environment. Interpreted functions are made known simply by putting a functional property on the property list of the atomic symbol which names the function. This is usually done using the built in function defun. Compiled functions must be made known by a more complex mechanism known as "loading," because of the complexity of the support mechanisms needed to make compiled functions execute efficiently. In some dialects of LISP the compiler automatically makes the compiled functions known, but in MACLISP the compiler creates a file in the file system of the host operating system, and this file has to be loaded before the compiled function can be called. In the pdp-10 implementation this file is called a "fasl file." In the Multics implementation it is called an "object segment." Loading is described in detail in section 14.3.

14.1.4 - Input to the Compiler

The input to the compiler consists of an ascii file containing a number of S-expressions. The format of this file is such that it could be read into a LISP environment using a function such as load or uread, and then the functions defined in this file would be executed interpretively.

When a file is compiled, the compiler reads successive S-expressions from the file and processes them. Each is classified as a function definition, a declare, or a "random form" according to what type of object it is and according to its car if it is a list.

A function definition is a form whose car is one of the atoms defun, defprop. When the compiler encounters a function definition if it defines a macro the macro is defined for use at compile time. If it defines an expr or a fexpr, the compiler translates the definition from LISP to machine code and outputs it into the "fasl file" or "object segment" which is the output from the compiler. If it defines some other property, it is

treated as a random form.

A random form is anything read from the input file that is not a function definition or a declare. It is simply copied into the output file of the compiler in such a way that when that file is loaded it will be executed.

A declare is a form whose car is the atom `declare`. It is ignored by the interpreter because there is an fsubr called declare in MACLISP which does nothing.

Note that if a form is read from the input file and its car has been defined as a macro, the compiler will apply the macro and then process the result as if it had been read from the input file.

14.1.5 - Functions Connected with the Compiler

`declare` FSUBR

In the interpreter, `declare` is like comment. In the compiler, the arguments are evaluated at compile time. This is used to make declarations, to gobble up input needed only in the interpreter, or to print messages at compile time. Examples:

```
(declare (special x y) (*fexpr f00))

(declare (read)) (needed-only-in-the-interpreter)

(declare (iog vt (princ "Now compiling fubar")))
```

`%include` FSUBR

(`%include` *name*) is used to cause an "include file" to be included in the file being read. This works in both the compiler and the interpreter. *name* may be a string or an atomic symbol. The include-file search rules are used.

Note: this function presently exists only in the Multics implementation.

nouuo SWITCH

If the nouuo switch is on, function calls made by compiled functions to compiled functions or system functions are forced to go through the interpreter each time. This aids in debugging. If the nouuo switch is off, the normal case, compiled calls can be made to go directly, which is much faster.

nouuo SUBR 1 arg

(nouuo t) sets the noouo switch.

(nouuo nil) turns off the noouo switch. (this is the initial state.)

nouuo returns t or nil according to whether it turned the nouuo switch on or off.

purcopy SUBR 1 arg

This function is of use only in the "bibop" implementation of pdp-10 lisp, which is presently under development on ITS. It has the effect of making a copy of its argument in pure free storage, so that constant list structure may be placed on sharable pages. This is primarily of use in the creation of large sharable systems like macsyma. On other implementations purcopy simply returns its argument.

14.2 - Declarations

It is often necessary to supply information to the compiler in order to compile a function beyond the definition of the function with `defun`, which is all that the interpreter needs in order to interpret the function. This information can be supplied through declares.

A declare is a list whose first element is the atom `declare` and whose remaining elements are forms called "declarations." The compiler processes a declare by evaluating each of the declarations, at compile time. Usually the declarations call on one of the declaration functions which the compiler provides. These are described below. However it is permissible for a declaration to be any evaluable form, and it is permissible for a declaration to read from the input file by using the `read` function. This may be used to prevent the compiler from seeing certain portions of the input which are only needed when a program is run interpretively. Prefixing a form in the input file with `(declare (eval (read)))` would cause it to be evaluated at compile time if the file was compiled or at read-in time if the file was interpreted. Arbitrarily complex compile-time processing may be achieved by the combination of declarations and macros.

The remainder of this section describes the declaration functions provided by the compiler. Note that if a declaration function described below is of the form `(foo t)`, its effect can be reversed by using the form `(foo nil)`.

`(special` *var1 var2* ... `)`
Declares *var1*, *var2*, etc. to be special variables.

`(unspecial` *var1 var2* ... `)`
Declares *var1*, *var2*, etc. to be local variables.

`(*expr` *fcn1 fcn2* ... `)`
Declares that *fcn1*, *fcn2*, etc. are `expr`- or `subr`-type functions that will be called. This declaration is generally supplied by default by the compiler but in some peculiar circumstances it is required to avoid error messages.

`(*lexpr` *fcn1 fcn2* ... `)`
Declares *fcn1*, *fcn2*, etc. to be `lexpr`- or `lsubr`-type functions that will be called. This declaration is required for non-builtin functions unless the functions are defined in the file being compiled and are not referenced by any functions that are defined before they are.

(*fexpr *fcn1* *fcn2* ...)
Declares *fcn1*, *fcn2*, etc. to be fexpr- or fsubr-type functions that will be called. This declaration is required for non-builtin functions unless the functions are defined in the file being compiled and are not referenced by any functions that are defined before they are.

(**array *arr1* *arr2* ...)
Declares *arr1*, *arr2*, etc. to be arrays that will be referred to. See the note under *expr.

(fixnum *var1* *var2* ...)
Declares *var1*, *var2*, etc. to be variables whose values will always be fixnums.

(fixnum (*fcn* *type1* *type2* ...) ...)
Declares *fcn* to be a function which always returns a fixnum result. Also the types of the arguments may be declared as *type1*, *type2*, etc. An argument type may be fixnum, meaning the argument must be a fixnum, flonum, meaning the argument must be a flonum, or notype, meaning the argument may be of any type.

The two types of fixnum declarations may be intermixed, for example (fixnum x (f00 fixnum) y).

(flonum *var1* *var2* ... (*fcn* *type1* ...) ...)
Is the same as the fixnum declaration except the variables or function-results are declared to always be flonums.

(notype *var1* *var2* ... (*fcn* *type1* ...) ...)
Is the same as the fixnum declaration except the variables or function-results are declared not to be of any specific type.

(fixsw t)
Causes the compiler to assume that all arithmetic is to be done with fixnums exclusively, except that obviously functions such as +$ and cos will still use flonums.

(fixsw nil)
Turns off the above.

(flosw t)
Causes the compiler to assume that all arithmetic is to be done with flonums exclusively, except that obviously functions such as + and tyo will still use fixnums.

`(flosw nil)`
Turns off the above.

`fixsw` and `flosw` are variables so `(setq fixsw t)` is an equivalent declaration to `(fixsw t)`.

`(setq special t)`
Causes all variables to be special.

`(setq nfunvars t)`
Causes the compiler to disallow functional variables.

`(macros t)`
Causes macros to be defined at run time as well as at compile time.

`(macros nil)`
Causes macros to be defined only at compile time. This is the default case.

`(genprefix` *foo*`)`
Causes auxiliary functions generated by the compiler (for such things as lambda-expressions passed as arguments) to be named `foo`*n*, where *n* is a number incremented by 1 each time such a function is generated. The genprefix declaration is used when several separately compiled files are to be loaded together, in order to avoid name clashes.

`(array*` *(type arr*1 *n*1 *arr*2 *n*2 ...) ...)
Is used to declare arrays *arr*1, *arr*2, etc. *type* may be fixnum, flonum, or notype; it indicates what type of objects will be contained in the arrays. *n*1, *n*2, etc. are the number of dimensions in *arr*1, *arr*2, etc. respectively.

`(arith` (*type*1 *fcn*1 *fcn*2 ...) (*type*2 *fcn*21 *fcn*22 ...) ...)
Is used to declare a general arithmetic function such as `plus` to be replaced by a one-type arithmetic function such as +. *fcn*1, *fcn*2, etc. are the functions to be replaced. *type*1, etc. is the type of function to replace them with: `fixnum` means replace them with the corresponding fixnum-only functions, e.g. replace `plus` by +. `flonum` means replace them with the corresponding flonum-only functions, e.g. replace `plus` by +$. notype means turn off a previous `arith` declaration for these functions.

The following declarations are useful only in the pdp-10 implementation; however, the Multics implementation will accept them and ignore those which are irrelevant.

`(mapex t)`
In the pdp-10 implementation, causes all map-type functions to be open-coded as `do` loops. (This is always done in the Multics implementation.) The resulting code is somewhat larger than otherwise, but also somewhat faster.

`(noargs t)`
Causes the compiler not to output information as to the number of arguments each function compiled takes; this provides some saving of memory space in some pdp-10 implementations.

`(messioc` *chars*`)`
Causes an `(ioc` *chars*`)` to be done just before printing out each error message. In this way one may direct error messages to the LAP file instead of to the terminal on the pdp-10.

`(muzzled t)`
Prevents the pdp-10 fast-arithmetic compiler from printing out a message every time closed compilation of arithmetic is forced.

`(symbols t)`
Causes the pdp-10 lisp compiler to output LAP directives so that the LAP assembler will attempt to pass assembly symbols to DDT for debugging purposes.

14.3 - Running Compiled Functions

After a file of functions has been compiled, those functions can be loaded into an environment and then called. They can be loaded either by using the load or fasload functions described below, or by using the autoload feature described in section 12.4.4.

The following function is at present available only in the Multics implementation.

load SUBR 1 arg

(load *x*), where *x* is a file specification acceptable by openi, i.e. a namestring or a namelist, causes the specified file to be loaded into the environment. The file may be either a source file or a compiled file (called a "fasl" file in the ITS implementation and an object segment in the Multics implementation.) load determines which type of file it is and acts accordingly. A source file is loaded by openi'ing and inpush'ing it. A read-eval loop is then executed until the end of the file is reached. An object file is loaded by reading it, defining functions as directed by specifications inserted in the file by the compiler.

fasload FSUBR

fasload takes the same arguments as uread. It causes a file of compiled functions, called a "fasl" file in some implementations, to be loaded in. Example:

```
(fasload foo fasl dsk macsym)
```

The following function only exists in the Multics implementation.

defsubr LSUBR 3 to 7 args

defsubr is the function used to define new machine code functions. It defines various types of functions, depending on its arguments. The way to define a subr written in PL/I is

```
(defsubr "segname" "entryname" nargs)
```

which defines segname$entryname as a subr expecting nargs arguments. The value returned is a pointer which can be putprop'ed under the subr property or the fsubr property. The way to define an lsubr written in

PL/I is

```
(defsubr "segname" "entryname" nargs2*1000+nargs1 -2)
```

which defines segname$entryname as an lsubr allowing from nargs1 to nargs2 arguments. The 1000 is octal. The value returned should be putprop'ed under the lsubr property.

Examples:

```
(putprop 'mysubr (defsubr "myfuns" "mysubr" 1) 'subr)
(putprop 'myfsubr (defsubr "myfuns" "myfsubr" 0) 'fsubr)
(putprop 'mylsubr
      (defsubr "myfuns" "mylsubr" 2001 -2) 'lsubr)
```

A function defined in this way receives its arguments and returns its value on the marked pdl, which may be accessed through the external static pointer

lisp_static_vars_$stack_ptr

See section 14.6 for details on how to access the arguments, and on the internal format of LISP data. lisp_static_vars_$nil and lisp_static_vars_$t_atom are fixed bin(71) external static; they contain nil and t.

14.4 - Running the Compiler

in the Multics implementation

The compiler is invoked by the lisp_compiler command to Multics. This command can be abbreviated lcp. The arguments to the command are the pathname of the input file and options. The compiler appends ".lisp" to the given pathname unless it is preceded by the -pathname or -pn option. The output object segment is created in the working directory with a name which is the first component of the name of the input file. For example, the command

lcp dir>foo.bar

reads the file "dir>foo.bar.lisp" and produces an object segment named "foo" in the working directory.

Usually no options need be supplied, since there are defaults. The options available are:

-pathname -pn -p
: Causes the following argument to be taken as the exact pathname of the input file, even if it begins with a minus sign. ".lisp" will not be appended.

-eval
: Causes the following argument to be evaluated by LISP. For example,
lisp_compiler foo -eval "(special x y z)"

-time -times -tm
: As each function is compiled, its name and the time taken to compile it will be typed out.

-total_time -total -tt
: At the end of the compilation, print metering information.

-nowarn -nw
: Suppresses the typing of warning messages. Error messages of a severity greater than "warning" will still be typed.

-macros -mc
: Equivalent to the (macros t) declaration: Causes macro definitions to be retained at run time.

-all_special
: Causes all variables to be made special. Equivalent to the (setq special t) declaration.

-genprefix -gnp -gp
Takes the following argument as the prefix for names of auxiliary functions automatically generated by the compiler. Equivalent to the genprefix declaration.

-check -ck
Causes only the first pass of the compiler to be run. The input file is checked for errors but no code is generated and no object segment is produced.

-ioc
If the following argument is *x*, (ioc *x*) is evaluated.

-list -ls
Causes a listing file to be created in the working directory, containing copy of the source file and an assembly language listing, with commentary, of the generated code. If the object segment is named "name", the listing file will be named "name.list".

-no_compile -ncp
Causes the compiler not to attempt to compile the file. Instead the input file is simply treated as being composed entirely of random forms. It is digested into a form which can be processed quickly by the load function.

in the ITS pdp-10 implementation

There are presently two versions of the LISP compiler on the ITS pdp-10 systems. COMPLR is the standard compiler, NCOMPLR the fast-arithmetic compiler; one must use the latter to produce open-compiled arithmetic code, but when this is not required the standard compiler is faster. (Note that it does not hurt to include fast-arithmetic declarations in files compiled by COMPLR; any irrelevant declarations are simply ignored.) Except for this one difference the use of the two compilers is identical, and the rest of this section applies to both unless otherwise specified. The word COMPLR will similarly mean "either COMPLR or NCOMPLR".

When you say COMPLR∧K the compiler will announce itself, print an underscore or backarrow, and accept a command line, which should be of the standard form

<output file> _ <input file> (switches)

The file specifications should be standard ITS file names, e.g. DEV:DIRNAM;FNAME1 FNAME2. If it is necessary to get a "funny" character such as _ into the file name, it may be quoted with a slash.

The compiler normally processes a file of LISP functions and produces a so-called "LAP file", containing S-expressions denoting pdp-10 machine-language instructions, suitable for use with LAP (the Lisp Assembly Program). However, one may direct the compiler instead to produce a binary

object file, called a "FASL FILE", suitable for use with the fasload function or the autoload feature. A third option is to process a previously generated file of LAP code to produce a FASL file. This is especially useful in the case where special-purpose functions have been hand-coded in LAP.

If one specifies only an input file name, say FOO BAR, then by default the name of a generated LAP file will be FOO LAP, and of a FASL file, FOO FASL.

The various modes of operation of the compiler may be controlled by specifying various switches, which are single letters, inside parentheses at the end of the command line. A switch may be turned off by preceding the switch letter with a minus sign. Extraneous or invalid switches are ignored. Initially all switches are off (the use of minus sign described above is provided in case the compiler is used for several files in succession). Valid switches to the compiler are:

A Assemble only. The specified input file contains LAP code which is to be made into a binary FASL file.

D Disown. Causes the compiler to disown itself after it has started running. This is the safest way to disown a COMPLR, because the compiler will know that it can't try to get any information from DDT.

F Fasl. Accept a file of LISP functions, produce a LAP file, and then assemble the LAP file into a FASL file. This is probably the most useful mode.

K Kill LAP file. Delete the LAP file after assembly. Usually used in conjunction with the F switch.

M Macros. Equivalent to (declare (macros t)). Causes macro definitions to be defined at run time as well as at compile time.

N No args properties. Equivalent to (declare (noargs t)). Normally the compiler outputs information in the LAP code as to how many arguments each function requires, so that args properties may be created on the appropriate atomic symbols at load time. In some implementations these properties occupy a significant amount of list space; thus it may be desirable to eliminate these properties.

S Special. Equivalent to (declare (setq special t)). Causes all variables to be considered special.

T Tty notes. Causes the compiler to print a note on the user's terminal as each function is compiled or assembled. This switch is normally off so that a COMPLR may be proceeded and allowed to run without the TTY. In any case error message will be printed out on the terminal.

U Unfasl comments. Useful only in conjunction with the F or A switch. Causes the assembler to output comment messages into a file whose second file name is UNFASL. (Actually, this file is always created, and error comments will be directed into this file also if messioc so specifies; but the file is immediately deleted if it contains nothing significant.) These comment messages describe the size of each function assembled, and give other random information also.

V no functional Variables. Equivalent to (declare (nfunvars t)).

W muzzled (i.e. Whisper). Equivalent to (declare (muzzled t)). Prevents the fast-arithmetic compiler from printing out a message when closed compilation of arithmetic is forced.

X map eXpand. Equivalent to (declare (mapex t)). Causes all map-type functions to be open-coded as if they were do loops. The resulting code is somewhat larger, but also somewhat faster.

Z Zymbols. Equivalent to (declare (symbols t)). Causes the compiler to output a special directive in the LAP code so that the LAP assembler will attempt to pass assembly symbols to DDT for debugging purposes. Primarily of use to machine language hackers.

COMPLR will accept a "Job Command Line" if desired; simply type

:COMPLR <command line><cr>

In this mode COMPLR will automatically proceed itself and run without the TTY, and kill itself when done.

It may be desirable to execute some LISP functions in the compiler before actually compiling a file. Typing ctrl/G will cause the compiler to announce itself and then type an asterisk; you will then be at lisp's top level. To make the compiler accept a command line, say (maklap). One useful command for debugging and snooping around is cl; (cl foo) will compile the function foo, which should be defined in the compiler's lisp environment, and print LAP code onto whatever device(s) are open for output.

14.5 - LAP on the pdp-10

The lisp compiler for the pdp-10 implementation does not output binary object files directly; rather, it outputs a series of S-expressions denoting the machine-language instructions of the compiled function. There are two programs which accept such S-expressions and convert them to binary machine language, called `lap` and `faslap`. (Historical note: the word "lap" dates back to 7090 LISP, and is derived from the phrase "Lisp Assembly Program".) `lap` is an in-core assembler; it reads in the S-expressions (hereafter referred to as "lap code") and deposits the resulting binary instructions in the binary program space of the current lisp environment. `faslap`, on the other hand, takes a file of lap code and produces a binary file suitable for use with `fasload`. `faslap` is normally part of the pdp-10 lisp compiler. Both assemblers will accept the same lap code, except for certain peculiar conditions. This section will describe the `lap` function and lap code; differences between `lap` and `faslap` will be treated in a special section.

14.5.1 - The LAP Function

`lap` is an fsubr which is executed primarily for its side effect - loading in a binary program. It accepts a series of S-expressions similar in form to a program written in MIDAS or MACRO-10. It is not intended to be a fancy assembler: it does not have conditional assembly, macros, or complex literal generation features. It does, however, contain sufficient power to load the output of the compiler, plus enough extra features that simple machine-language functions may be hand-coded in it. (See the section on conventions for writing lap code by hand later in this chapter.) The major operational differences between `lap` and MIDAS or MACRO-10 are that (1) `lap` is one-pass, while the others take two, (2) `lap` uses the function `read` to input lap code, while the others are more efficient, and (3) `lap` assembles directly into the lisp environment, while the others produce a binary object file (note that `faslap` differs only in the first two respects).

The `lap` function is an fsubr which expects to get two atomic symbols as arguments; the first is the name of the function to be assembled, and the second is the type (i.e. the property under which it is to be stored on the property list.) Thus

```
(lap quux subr)
```

would assemble a subr called `quux`. When invoked, `lap` repeatedly calls the `read` function, operating on the S-expressions thus obtained, until a `nil` is encountered, at which time the assembly ends. Some messages may be printed out as this happens. If the assembly completed successfully, the variable `bporg` is updated to reflect the new size of binary program space, and the appropriate property is placed on the property list of the specified atomic symbol.

Normally, lap does not reside in the initial lisp system, though the initial system does contain several specialized functions for use by lap. Instead, lap has an autoload property of (lap fasl com) on ITS and (lap fasl sys) on the 10/50 system. Thus, if one simply reads in a file of lap code lap will load automatically and assemble the functions.

Here is an example of some lap code which corresponds roughly to the lisp function memq:

```
    (LAP FUNNY-MEMQ SUBR)
    (ARGS FUNNY-MEMQ (NIL . 2))
MEMBEG  (JUMPE B MEMEND)    ;result nil if arg 2 nil
        (HLRZ T 0 B)        ;else look at car of arg 2
        (CAIN T 0 A)
        (JRST 0 MEMEND)     ;win if same as arg 1
        (HRRZ B 0 B)        ;else take cdr of arg 2
        (JRST 0 MEMBEG)     ; and try again
MEMEND  (MOVEI A 0 B)       ;return arg 2
        (POPJ P)            ;exit from function
    NIL
```

Note that this is greatly different in style from 7090 lap, which took the entire program as one argument, and a symbol table as the other. The drawback with the 7090 method is that the entire program must be read in before it is assembled; this can require prohibitively large amounts of memory, especially for the lap output from compilation of a large lisp function. The method used by pdp-10 lap is much more reasonable in practical situations (e.g. reading in lap code from a file).

lap does not use an a-list for its symbol table, either. Rather, the value of the symbol is stored on the property list under the sym property. Thus (defprop ztesch 43 sym) would make the symbol ztesch known to lap, with the value 43. lap has a number of symbols initially defined, including the names of all the accumulators, and the addresses of some useful routines internal to lisp. It also uses the function getmidasop on a symbol if the symbol is otherwise undefined to determine whether it is a pdp-10 instruction (the getmidasop function contains a concise table of all pdp-10 instructions and most monitor calls, as well as names of UUO's used internally by lisp). In this way lap can recognize all standard instruction mnemonics without defining 400 or more sym properties. If lisp's symbol table has been loaded into DDT, then lap will ask DDT about the values of symbols as a last resort. In this manner hand-coded lap code may refer to any location internal to lisp (with an appropriate amount of caution, of course).

When lap terminates, it returns as its value a list of the new value of bporg and the entry point(s) of the function defined (hand-coded functions may have more than one entry point). If any symbols were undefined or multiply defined, they will be printed out first. It is generally a good idea to let lap terminate naturally, rather than quitting out of it, since it hacks the lisp environment in various peculiar ways.

14.5.2 - Valid LAP Code Forms

lap acts on the S-expressions it reads as follows:

nil

Terminate assembly and return. Any literals generated are assembled into memory at the end of the function, temporary symbol definitions are flushed, and (gctwa t) is evaluated. The nil should be followed by a space, because carriage return is not always an atom separator.

<atomic symbol>

Assign to the atomic symbol (non-nil, of course) a temporary sym property equal to the address of the next word to be assembled into. If (symbols t) is in effect lap will pass the value of this symbol to DDT if lisp's symbol table has been loaded. Thus one uses atomic symbols as location tags.

(defsym <atom1> <value1> ... <atomn> <valuen>)

For each i define <atomi> with a sym property of (eval <valuei>). No binary words are generated, and these symbols are not passed to DDT. Note that this performs a lisp evaluation, not a lap evaluation!

(entry <name> <type>) or (entry <name>)

Defines the atomic symbol <name> to be a function of type <type> with entry point at the current location. If <type> is not specified, it defaults to the second argument to lap. No binary words are generated. This form is not used by the output of the lisp compiler, and is provided only as an aid to writers of hand-coded lap code. It allows several functions to share symbols and storage areas.

(args <atom> <args-prop>)

If the assembly terminates successfully, then <atom> gets <args-prop> as its args property. <atom> must be an entry point of the function being assembled. No binary words are generated. The pdp-10 lisp compiler will output this declaration in each function compiled unless (declare (noargs t)) or the N switch is given. faslap requires that each args declaration follow the corresponding entry point.

(comment ...)

This form is totally ignored. Of course, since the lisp reader is used to input lap code, semicolon comments may be used as well.

(eval <form1> ... <formn>)
Applies the function eval to each form in turn. No binary words are generated. This is useful for such things as (eval (setq ibase 23.)) or whatever.

(symbols <t-or-nil>)
Controls the lap feature which tries to pass symbolic location names to DDT. Furthermore, if the symbols pseudo-operation occurs anywhere within a given lap function, the names and locations of the entry points of that function will also be passed to DDT. No binary words are generated. Note that there is possibility for confusion here because lap will accept as tags atomic symbols with names of any length, while MIDAS and DDT truncate tags to six characters. Thus quuxbar and quuxbaz are two different symbols to lap, but will interfere with each other when passed to DDT. When symbols are passed to DDT, the names are truncated to six characters, and any non-squoze character (a character other than A-Z, 0-9, ., $, or %) is assumed to be a dot. Since dots must be slashified to be read into lisp, the standard convention is to use * (the lisp compiler uses this convention). Thus one would write (JSP D *LCALL) instead of (JSP D /.LCALL).

(block <fixnum>)
Assembles a block <fixnum> words long, containing zeros.

(ascii <S-expression>)
Explode's the <S-expression> and assembles the characters obtained into successive words, five per word, in ASCII code.

(sixbit <S-expression>)
Similar to ascii, but characters are assembled six per word in sixbit code (ascii characters between 40 and 137 are represented as 0 to 77).

(squoze <atom>) or (squoze <fixnum> <atom>)
(ITS only) Produces a word of squoze code, which is a left-justified radix-50 code, from the first six characters of the print name of <atom>. If <fixnum> is present, then it is divided by 4 and the low four bits of the quoutient are added into the high four bits of the squoze value (the MIDAS convention).

<any other list>
Assembles a single word, which is assumed to be an instruction of some kind. First, if the list contains the atomic symbol @, it is deleted from the list and saved. Then the first four components of the list are processed in order. These must all be lap "syllables" (described below). If the length of the list is less then four, missing elements are assumed to be zero. The four elements of the list are assumed to

be, in order, the operation code, accumulator, address, and index fields of a pdp-10 instruction. These are evaluated by the lap syllable evaluator to obtain four numbers, which are then added together after being modified as follows:

opcode	no change
accumulator	shift left 23. places
address	clear left half
index	swap halves

Finally, if the atom @ had been present, the octal number 20000000 is logically or'ed into the result, thus turning on the indirection bit. Note that neither the accumulator nor the index field is truncated to four bits. This has many useful applications; see, for example, the description of the specbind routine below.

There is a fairly strong similarity between code written in lap and equivalent code written in MIDAS or MACRO-10. The essential difference is that lap processes assembly fields in order from left to right in order to determine which field is which. One pitfall to avoid is writing such instructions as (JRST FOO) or (SETZM FOO) when one intends rather (JRST 0 FOO) or (SETZM 0 FOO). Another difference to remember is that lap uses the lisp reader to input lap code; thus one must remember to put spaces around an @, and that one cannot write (JRST 0 FOO+3) unless FOO+3 really is a tag! (For arithmetic operations within assembly fields, see the description of lap syllables below.) If it is desired to make lap code look more like the standard assembly languages, one may use the fact that comma is like a space to lisp, and that extra parentheses don't hurt, and write (MOVE A,TABLE(10)) instead of (MOVE A TABLE 10).

14.5.3 - LAP Syllables

Each of the four components of assembly words are evaluated by the lap evaluator to produce numeric quantities; these are then combined to form an assembly word. Note that @ is treated specially and is not a component. Forms to be evaluated by the lap evaluator are called lap syllables. Valid forms for lap syllables are as follows:

<number>

Fixnums evaluate to themselves, and may be operated upon by lap arithmetic operations. Flonums also evaluate to themselves, but arithmetic operations on them will not work. Flonums should not be used in the address field, because the left halves will be truncated off; they should be used only in the opcode or index fields (the latter is useful for writing (FADRI 7 0 3.0) or something like that).

nil
: same as (quote nil).

*
: Evaluates to the address of the word into which the current instruction will be assembled. Equivalent to . in MIDAS and MACRO-10 (however, see the note below about literals).

<atomic symbol>
: Any atomic symbol other than @, *, and nil evaluates to its assembly symbol value. That is, if the symbol has a sym property, then it is the value of that property; otherwise, the value returned by the getmidasop function if non-nil; otherwise, the value which DDT assigns to it. An error occurs if no value can be found for a symbol.

(quote <s-expression>)
: Protects <S-expression> from garbage collection, and evaluates to the address of the S-expression. Thus (MOVEI A '(A B)) puts the address of the S-expression (A B) into accumulator a. Warning: faslap permits this syllable only in the address field.

(function <s-expression>)
: Same as (quote <s-expression>), but emphasizes that <S-expression> is a function. Thus one might write (CALL 2 (FUNCTION CONS)).

(special <atom>)
: Evaluates to the address of the value cell of the atomic symbol <atom>. If <atom> does not have a value cell, one is created for it first. Thus, for example,

```
(MOVE A (SPECIAL QUUX))
(MOVEM B (SPECIAL ZTESCH))
```

Accomplishes the equivalent of (SETQ ZTESCH QUUX). faslap permits this syllable only in the address field.

(array <atom>)
: Evaluates to the address of the sar0 of the array which is on the property list of <atom>. If <atom> is not yet an array, a dummy array property is created. This is of use for open-coded array accessing. (At present this feature is not yet implemented.)

(ascii <s-expression>)
Evaluates to a 36-bit quantity consisting of the ascii representation of the first five explodec'd characters of <S-expression>. Note that the ascii pseudo-op may generate several binary words as a lap form, but only a single-word quantity as a syllable.

(sixbit <S-expression>)
Like ascii, but uses the first six characters and produces a sixbit representation quantity.

(squoze <atom>) or (squoze <fixnum> <atom>)
(ITS only) Similar to the same form as a lap form: produces a word of squoze code as its value.

(+ <lapsyl> <lapsyl> ... <lapsyl>)
Adds together the values of the lap syllables. (Thus note that lap syllables are defined recursively.) This allows one to write such things as (JRST 0 (+ FOO 3)).

(- <lapsyl>)
Evaluates to the negative of the value of <lapsyl>.

(- <lapsyl> <lapsyl> ... <lapsyl>)
Subtracts the values of all the lap syllables after the first from the value of the first.

(<lapsyl> <lapsyl> ... <lapsyl>)
Same as (+ <lapsyl> <lapsyl> ... <lapsyl>).

(<lapsyl>)
Evaluates to the value of <lapsyl>. It most definitely does not evaluate to the swapped-halves value of <lapsyl>, as some might think! When one writes (MOVE A,FOO(B)), the value of b gets swapped because it is in the index field, and not because it is in parentheses.

(% <lap assembly word>)
Generates a literal; i.e. the cdr of the list is saved and assembled at the end of the function. The value of the syllable is the address of this remotely generated word. <lap assembly word> must be an instruction, or one of the ascii, sixbit, or block pseudo-ops. (The block pseudo-op is relatively useless here.) Thus, for example, (MOVEI T (% SIXBIT LONG-MESSAGE!)) is perfectly valid. There are some restrictions on the use of literals: they cannot be nested, i.e. a literal may not contain in its assembly word another literal. Also, * in a literal refers to the location of the literal, not of the

referencing instruction. Thus (JUMPE A (% AOJA T (+ * 1))) will not do what you might expect from using MIDAS. Finally, faslap permits literals only in the address field.

14.5.4 - Functions and Variables Used by lap and faslap

The functions described in this section are available in the initial pdp-10 lisp system primarily for the benefit of lap and faslap. Some of them may be of use to the user, however. Those which are probably not of use to the user are flagged with a ***.

6bit| *** SUBR 1 arg

The argument is explodec'ed, the first six characters assembled into one word in sixbit code, and the result returned as a fixnum.

ascii| *** SUBR 1 arg

The argument is explodec'ed, the first five characters are assembled into one word in ascii code, and the result returned as a fixnum.

sqoz| *** SUBR 1 arg

The argument should be a list of one or two items. The last item should be an atomic symbol, and the first, if present, should be a fixnum. The first six characters of the atom's print name are converted to ITS-style (left-justified) squoze code, with non-squoze characters assumed to be dots. If the fixnum is present, then it is divided by four, and the low four bits of the result become the high four bits of the squoze code. The squoze code is then returned as a fixnum.

pagebporg SUBR no args

Causes the variable bporg to be adjusted upwards so as to lie on a page boundary. This is principally useful on ITS in conjunction with the function purify. pagebporg returns the new value of bporg.

purify SUBR 3 args

The first two arguments to purify should be fixnums, and delimit a range of memory within the lisp system. The third argument is a flag. If it is nil, then the pages covered by the specified range of memory are made impure, i.e. writable. If it is t, then the pages are made pure, i.e. read-only and sharable. If it is bporg, then the pages are

also made pure, but in addition some work is done to make sure that no uuo on those pages may ever be "clobbered". This option should always be used if the pages involved contain binary code loaded by lap or fasload. Presently purify does nothing in the dec-10 implementation; it is intended primarily for producing systems built on lisp, such as Macsyma, in such a way that pure pages can be shared between users. Example: the following sequence of commands might be used to produce a sharable system on ITS:

```
(SETQ LOPAGE (PAGEBPORG))       ;save low page address
(SETQ PURE T)                   ;specifies pure code
(FASLOAD FUNNY FASL)            ;load up system
(FASLOAD WEIRD FASL)
(UREAD SOME LAP)
    ...
(SETQ ERRLIST '((TERPRI)        ;stuff for system startup
                (PRINC 'WELCOME/ TO/ SUPERSYSTEM/!)
                (TERPRI)))
(SSTATUS TOPLEVEL               ;set up top level for system
         (FUNCTION TOP-HANDLER))
(SETQ HIPAGE (PAGEBPORG))       ;save high page address
(PURIFY LOPAGE (1- HIPAGE) 'BPORG)      ;purify pages
(MACDMP ':PDUMP/ SYS:TS/ SUPER/^M)  ;tell DDTto dump
```

getddtsym   SUBR 1 arg

The argument to getddtsym is explodec'ed and the first six characters converted to squoze code, as with the sqoz| function. This code is then given to DDT to determine its value. This value is returned as a fixnum, if there is a value; if there is no value, or if there is but the symbol table has not been loaded into DDT, then nil is returned. Presently this function always returns nil in the dec-10 implementation. Note that this function works as well for system symbols, which are of course known to DDT; thus (GETDDTSYM '*RSNAM) will return 16.

getmidasop   SUBR 2 args

This function is similar to getddtsym, but works only for standard pdp-10 operation codes such as HLRZ and JFFO, for names of UUO's used by lisp, and on dec-10 for names of monitor calls such as LOOKUP. The pdp-10 lisp system contains its own symbol table in a compact format for these symbols; it permits their values to be determined without having to consult DDT. This is of course of great value to lap and faslap. Example: (GETMIDASOP 'JRST) would return 254000000000.

putddtsym SUBR 2 args

This is the inverse of getddtsym. The first argument is used to obtain squoze code, and the second should be a fixnum. The function attempts to tell DDT that the symbol has the specified value. It returns t if it succeeded, and otherwise nil. In the dec-10 implementation it always returns nil.

putddtsym| *** SUBR 2 args

This is a special entry to putddtsym for use by lap only.

lapsetup| *** SUBR 2 args

This function is used for initialization of lap, and should not be used by the user.

gwd| *** SUBR 1 arg

This function assembles instructions for lap; it resides in the initial lisp system for efficiency. For use by lap only.

rpatch| *** SUBR 3 args

This function primarily handles the problem of forward references during the one-pass lap assembly by modifying previously assembled words when a symbol is eventually defined. For use only by lap.

faslapsetup| *** LSUBR 0 or 1 args

Used for initializing faslap.

smallnump SUBR 1 arg

This function determines whether a given fixnum lies in the range of fixnums which are "uniquized" for efficiency reasons by the pdp-10 lisp system. This aids lap somewhat in determining whether a number needs protection from the garbage collector.

gcprotect *** SUBR 1 arg

The argument to gcprotect is "uniquized" (i.e. "interned") on a special array internal to the pdp-10 lisp system, and the unique copy returned. This array is the one whose size is controlled by (sstatus losef <fixnum>). This is of use to lap for protecting from garbage collection S-expressions which are referred to by lap code, e.g. via

the quote syllable.

gcrelease *** SUBR 1 arg

This is the inverse to gcprotect. If it is used, extreme caution must be exercised!

Some system variables are also required by lap and fasload; these are described below.

bporg VARIABLE

The value of bporg should always be a fixnum, whose value is the address of the first unused word of binary program space. This value generally should not be altered by the user, but only examined. bporg is updated whenever binary code is loaded by lap or fasload.

bpend *** VARIABLE

This variable should also always have a fixnum as its value; this indicates the first address above the last available word of binary program space. This is updated by many internal lisp routines, such as the garbage collector, the array allocator, and lap and fasload.

pure VARIABLE

This variable, initially nil, should be made non-nil by the user before loading binary code which is to be made pure. It signals lap and fasload to be circumspect about any UUO's in the code, because pure UUO's cannot be clobbered to be PUSHJ's or JRST's. lap solves this problem by clobbering the UUO immediately if the referenced function is already defined and is itself a subr rather than an expr; otherwise the UUO is made permanently unclobberable (i.e. CALL is converted to CALLF, etc.).

fasload is somewhat more clever: it too tries to clobber each UUO immediately, but if it can't it puts the address of the UUO on a list called purclobrl, which is checked at the end of each call to fasload, and each UUO on the list is clobbered at that time, if the appropriate function had been loaded by that call to fasload. If the function never does get defined, then purify will also check purclobrl and convert each UUO to its permanently unclobberable form.

If pure has a fixnum as its value, then fasload (but not lap) behaves somewhat differently. If the value of pure (which must be between 1 and 8 or so) is, say, 3, then fasload calls pagebporg, and then reserves 6=2*3 pages of binary program space, unless a previous call

to fasload has already reserved them (i.e. they are reserved only once). Thus fasload has two sets of 3 pages to work with; we shall call the first set "area 1" and the second set "area 2". Now whenever fasload has to load a clobberable UUO, it does not place it in the code being loaded, but rather hashes it and places it in area 1 if it was not there already; a copy is placed in the same relative position in area 2. Then an XCT instruction pointing to the UUO in area 1 is placed in the binary code. When all loading has been done, area 2 may be purified, but area 1 may not.

Now when running the code, the UUO's pointed to by the XCT's may be clobbered (the pdp-10 lisp UUO handler is clever about XCT), and the code will run faster the second time around because the XCT's will point to PUSHJ's. However, if (sstatus uuolinks) is called, then area 2 is copied back into area 1, effectively unclobbering all the UUO's. Naturally, an area large enough to contain all the UUO's should be reserved; (status uuolinks) (q.v.) yields information relevant to this.

Thus the example given under the purify function above might be modified as follows:

```
(SETQ LOPAGE (PAGEBPORG))       ;save low page address
(SETQ PURE 3)                   ;specifies pure code
(SETQ LOPAGE (+ LOPAGE 6000))   ;allow for area 1
(FASLOAD FUNNY FASL)            ;load up system
(FASLOAD WEIRD FASL)
(UREAD SOME LAP)
     ...
(SETQ ERRLIST '((TERPRI)        ;stuff for system startup
                (PRINC 'WELCOME/ TO/ SUPERSYSTEM/!)
                (TERPRI)))
(SSTATUS TOPLEVEL               ;set up top level for system
         (FUNCTION TOP-HANDLER))
(SETQ HIPAGE (PAGEBPORG))       ;save high page address
(PURIFY LOPAGE (1- HIPAGE) 'BPORG)     ;purify pages
(MACDMP ':PDUMP/ SYS:TS/ SUPER/^M)     ;tell DDTto dump
```

*pure   *** VARIABLE

This variable is relevant only to the "bibop" implementation on ITS, and controls certain kinds of automatic purification of S-expressions and atomic symbols.

purclobrl   *** VARIABLE

Used by fasload to keep track of UUO's which are potentially but not immediately clobberable.

gwd|, lapsetup|, rpatch| *** VARIABLES

The values of these variables are used for internal communications within lap and should not be disturbed by the user.

14.5.5 - Differences Between lap and faslap

Much effort has been made to keep lap and faslap compatible. There are of necessity, however, some differences. One is that lap reads in a function only once, whereas faslap presently reads a file of functions twice through; if funny things such as macro characters are happening during reading this may cause problems. A related problem is that faslap reads the lap code at assembly time, and not at load time, which means ,that read macro characters and obarray hackery will not happen at fasload time. faslap and fasload cooperate in a scheme to gain speed by calling the function intern only once on each atomic symbol needed by a file of functions; faslap creates a table of such symbols and passes them when encountered into the binary file. This means that switching obarrays in the middle of a fasload file will probably lose.

There are also some internal differences due to the different modes of operation. As an in-core assembler, lap does not need to worry about questions relating to relocatability. faslap, however, does not know where in memory a binary file will be loaded, and thus must produce relocatable binary code. This implies that faslap must distinguish between relocatable and absolute symbols. This is done by using non-numeric sym properties for relocatable symbols; the user who hand-codes lap code and expects to look at sym properties at assembly time should be aware of this.

faslap furthermore does not know into what version of lisp the binary file will be loaded. This poses a problem because compiled code needs to refer to routines and locations internal to lisp, such as FLOAT1 and ERSETUP. This is solved by the so-called globalsym convention; these labels, which for lap have numeric sym properties, in faslap have non-numeric sym properties, and direct faslap to output directions to fasload to find the correct value of a symbol for the lisp being loaded into. For most purposes such symbols should be treated as a funny kind of relocatable symbol.

faslap imposes some restrictions on the use of certain constructs. Multiple and negative relocatability is not permitted. Relocatable symbols, the quote, function, special, and sar0 constructs, and literals are permitted only in the address field of an instruction.

14.5.6 - Conventions for Functions in Lisp

This section briefly describes some of the internal conventions of pdp-10 lisp, and contains enough information for a person who knows pdp-10 machine language to understand the output of the compiler, and possibly to write simple lap functions for use with lisp. However, the information within this section is subject to change. Whenever any location within lisp is referred to symbolically in this section, that symbol is predefined to lap and may be used by any lap program even if DDT does not have lisp's symbols loaded.

The names of the accumulators and their uses are, briefly:

0	nil	atom header of the atomic symbol nil
1	A	first argument to a function; value of function
2	B	second argument
3	C	third argument
4	AR1	fourth argument
5	AR2A	fifth argument
6	T	negative of the number of args to an lsubr; temp
7	TT	super-temporary; value from numeric function
10	D	semi-temporary; arithmetic
11	R	semi-temporary; arithmetic
12	F	semi-temporary; arithmetic
13	FREEAC	unused, except saved/used/restored by gc
14	P	regular pushdown list (pdl) pointer
15	FLP	flonum pdl pointer
16	FXP	fixnum pdl pointer
17	SP	special (variable bindings) pdl pointer

In general, S-expressions should be manipulated in the five argument accumulators; the contents of these are protected by the garbage collector. Random arithmetic should not be done in them; this might accidentally generate the address of something the garbage collector should not protect. Arguments to subrs are passed through these five accumulators, and the value of a function is returned in accumulator A. The single argument to an fsubr is likewise passed through accumulator A.

It is generally assumed that when an argument is passed or a value returned through these five accumulators that that the left half will be zero, while the right half will contain a pointer to an S-expression. Much code depends on the left half being zero; in particular, tests for nil (which is the zero pointer) use JUMPE instructions, which require that the left half be zero so that the test of the right half will be valid. In general, then, instructions like HRRZ and HLRZ should be used to fetch items into these accumulators.

S-expressions are represented in such a way that if a pointer to a dotted pair is in, say, accumulator A, then

```
(HLRZ B 0 A)
```

will get, as a pointer, the car of the S-expression and put it in accumulator B, and

```
(HRRZ B 0 A)
```

will get the cdr. If the S-expression whose address is in A is a fixnum or flonum, then

```
(MOVE TT 0 A)
```

will get the machine representation of the number and put it in accumulator TT.

Accumulators T through F may be used as scratch registers, in general. When an lsubr is called, however, the negative of the number of arguments is passed through accumulator T. Many useful internal routines are called by JSP T,FOO, and the argument or value is commonly passed through TT. Functions compiled by the fast-arithmetic compiler return their values in TT. TT is also used in connection with array accessing.

FREEAC is presently unused by the lisp system, except for the garbage collector, which, however, saves and restores it. This fact should not be taken as permanent; it is mentioned primarily because it can be useful for debugging purposes.

The lisp system uses no fewer than four pushdown lists, or stacks. The regular and special pdis, whose pointers are in P and SP, are marked from by the garbage collector; thus an S-expression is "safe" from gc is pushed on either of these pdls. (Only the right half of each pdl slot is marked from; the laft half may contain garbage.) The special pdl is used to hold variable bindings, and its contents are highly structured. The user should not use SP except through the routines SPECBIND and UNBIND, described below. P may be used for any purpose, provided that totally random things are not put into the right halves of pdl slots (the same restriction as for argument accumulators). The fixnum and flonum pdls are used primarily by compiled code produced by the fast-arithmetic compiler, and their contents are not affected by gc in any way. If it is desired to save random quantities on a stack, the fixnum pdl should be used if possible.

The standard function calling convention in pdp-10 lisp requires that functions be effectively called via a (PUSHJ P <function>) and exit via (POPJ P). The arguments to subrs and fsubrs are as described above. Lsubrs take their arguments on the regular pdl (where they are safe from gc), and T has minus the number of arguments. The return address is also on the pdl, under the arguments. This usually requires code of this sort:

```
        (PUSH P (% 0 0 G0475))
        (PUSH P A)
        (PUSH P '(funny list))
        (MOVNI T 2)
        (JRST 0 FOO-LSUBR)
G0475   --- lsubr returns to here ---
```

That is, the return address must be pushed ahead of time. It is the responsibility of the called lsubr to remove its arguments from the pdl and return with a POPJ.

Interfacing between compiled code and the interpreter is accomplished via a large set of UUO instructions. All of them work in the same fashion: the effective address must be the address of an S-expression which is the function to be invoked. The arguments to this function are passed in the manner described above, and the accumulator field describes which argument passing convention has been used (hopefully the same as that required by the called function): 0-5 means a call to a subr with that many arguments, 16 means a call to an lsubr, and 17 means a call to an fsubr. Thus the function CONS might be called with the UUO (CALL 2 (FUNCTION CONS)).

There are several variants on this basic UUO type. One variant is the JRST vs. PUSHJ mode; sometimes instead of writing a PUSHJ to a function one wants to write a JRST for efficiency. To see why, consider that

```
(PUSHJ P FOO)
(POPJ P)
```

is in effect equivalent to

```
(JRST 0 FOO).
```

This kind of UUO is also useful for calling lsubrs (see the example above).

A second variant is the "clobberable" vs. the "unclobberable" UUO. If certain conditions are met, it is possible for the UUO handler to replace the invoking UUO by the equivalent PUSHJ or JRST, so that next time the same code is used it will call the desired function directly. In some cases, however, it is not desirable for the UUO to be so clobbered, for example if the function to be invoked is an argument in an accumulator, and is to be invoked via something like (CALL 1 0 A). A UUO may therefore specify that it may never be clobbered. A third option is used by code compiled by the fast arithmetic compiler. It is undesirable for a function which returns a number to do a "number cons" in order to return the number as an S-expression if the number will only be converted back to a machine number and used in more open-coded arithmetic. (It is undesirable because number consing, like ordinary consing, eventually causes garbage collection, an expensive process.) Thus a UUO may specify that it wants only a machine number as a result; this is to be returned in accumulator TT, rather than a lisp number in A. The mnemonics for all these UUOs are summarized here:

	clobberable		unclobberable	
	PUSHJ	JRST	PUSHJ	JRST
standard result	CALL	JCALL	CALLF	JCALLF
numeric result	NCALL	NJCALL	NCALLF	NJCALF

Thus the example of an lsubr call above would actually be written:

```
          (PUSH P (% 0 0 G0475))
          (PUSH P A)
          (PUSH P '(FUNNY LIST))
          (MOVNI T 2)
          (JCALL 16 (FUNCTION FOO-LSUBR))
G0475
```

Functions produced by the fast arithmetic compiler follow a convention so that NCALLs will work properly: If a function is to be NCALL'ed, and returns a fixnum, the first instruction of the function should be (PUSH P (% 0 0 FIX1)); if it returns a flonum, the first instruction should be (PUSH P (% 0 0 FLOAT1)). (For a description of the FIX1 and FLOAT1 routines, see below.) If the function is NCALL'ed, the function is entered at the second instruction, i.e. after the PUSH. The appropriate machine number is returned in accumulator TT, as expected by the caller. If, on the other hand, the function is simply CALL'ed, then it is entered at the normal entry point, and the address of FIX1 or FLOAT1 goes on the stack. When the function exits, it will transfer to FIX1 or FLOAT1, which will convert the machine number to a lisp number and then return to the original caller.

Some other UUO's besides the CALL UUO's are useful to compiled code and hand-coded lap. The STRT (STRing Typeout) UUO is quite useful for printing out constant strings of characters. The effective address of the STRT UUO must be the first of several words of sixbit characters. Several characters in the string have special significance:

^ Complement the 100 bit of the character before printing it. (This occurs after 40 has been added to convert it to ascii.) Thus ^M in the sixbit string causes a carriage return to be printed. Similarly, ^4 is a lower case t.

! Terminate typeout.

Quote the next character. This is used to get #, ^, and ! into a string.

Thus, for example, to print the message "YOU LOSE!" in lap code, preceded and followed by a carriage return, say

```
          (STRT 0 (SIXBIT /^MYOU/ LOSE#!/^M!))
```

(The slashes are necessary because lap will read this using the lisp reader!)

The LERR (Lisp ERRor) UUO takes a string like the ones STRT takes, and signals an uncorrectable error, with the string as the error message. Because the error is uncorrectable, control never returns to after the LERR; it is like a JRST to the error handler.

The LER3 UUO is similar to LERR, but also takes an S-expression in accumulator A; this expression should be followed by the string which

constitute the error message.

The ERINT UUO is used to signal correctable errors. It too takes a string argument and an S-expression in A. The accumulator field of the ERINT UUO indicates the type of error:

```
0       undef-fnctn
1       unbnd-vrbl
2       wrng-type-arg
3       unseen-go-tag
4       wrng-no-args
5       gc-lossage (ordinarily used only by gc)
6       fail-act
```

The S-expression becomes the argument to the error interrupt handler for the given type of error (in the case of types 0 to 3, the error handler automatically applies the function ncons to this object before passing it as the argument). If the handler returns a corrected value (e.g. the user in a standard error break used the return function) then this new value is passed back in A and control returns to the instruction after the ERINT.

A typical piece of lap code to use this might be:

```
(LAP FOO SUBR)
        (PUSH P A)
TEST    (JSP T FXNV2)         ;get numeric value in d
        (TRNE D 3)            ;want a multiple of 4
        (JRST 0 LOSE)
        . . .
        (POPJ P)
LOSE    (EXCH A B)            ;get bad arg in a
        (ERINT 2 (% SIXBIT NOT A MULTIPLE OF 4))
        (EXCH A B)            ;switch back again
        (JRST 0 TEST)         ;go try again
NIL
```

UUO's never change the values in any accumulators except ERINT, which may return a new value in A, and the various CALL UUO's, which may clobber everything if they have to invoke eval to link to an interpreted function. CALL UUO's save all accumulators when linking from one compiled or handcoded function to another. This implies that the called function will get whatever was placed in accumulators T through F as well as A through AR2A. It does not imply, however, that any accumulators will have been preserved by the time the called function has returned to the caller.

14.5.7 - Internal Routines for use by LAP Code

Compiled code requires a certain set of support routines. The names and addresses of these routines are predefined to lap. It should not be assumed that a given routine saves any accumulators unless it is

specifically described as doing so. They are briefly described here:

(JSP T SPECBIND)

This routine handles the binding of special variables. The call is followed by one or more specifications of the form (<type> <where> (special <atom>)), where <type> is either 7_41 or 0. The value of the atomic symbol <atom>, which is in the word pointed to by the effective address of the argument, is saved on the special pdl, and a new value is placed in the value cell, as specified by <type> and <where>. If both <type> and <where> are zero, the new value is nil. If <type> is zero, then <where> is the number of an accumulator containing the new value. If <type> is 7_41, then the new value is in the regular pdl slot addressed by subtracting <where> from the current contents of accumulator P; <where> may be any number less than 2000 octal. (This is a case where not truncating the accumulator field of a lap instruction to four bits is very useful.) Any number of specifications may follow the call to SPECBIND; the end of the call is determined by the fact that a valid pdp-10 instruction within lisp cannot be zero in the first nine bits or ones in the first three. All the values pushed in a single call form a single bind block; this fact is used by the UNBIND routine. SPECBIND destroys the contents of accumulator R.

(JSP T (SPECBIND -1))

This is an alternate entry to SPECBIND, which has the additional effect of passing all new values through the routine PDLNMK (see below) before placing them in the value cells. It is used by code compiled by the fast-arithmetic compiler.

(PUSHJ P UNBIND)

Pops one bind block off the special pdl, thus restoring the old values of the atoms whose values were formerly saved. Example: the following lisp code and lap code are roughly equivalent:

```
((LAMBDA (SPECVAR) (ZORCH)) 'BARF)

        (MOVEI B (QUOTE BARF))
        (JSP T SPECBIND)
        (0 B (SPECIAL SPECVAR))
        (CALL 0 (FUNCTION ZORCH))
        (PUSHJ P UNBIND)
```

UNBIND does not destroy any accumulators.

(JSP T PDLNMK)

"Pdl number make". This routine examines the S-expression in accumulator A, and if it is a pdl number it replaces it with a freshly number-consed copy. Used by code produced by the fast-arithmetic compiler. Does not destroy any other accumulators, even TT.

(JRST 0 PDLNKJ)
Equivalent to
```
(JSP T PDLNMK)
(POPJ P)
```

(JSP T FXCONS)
Takes a machine fixnum in accumulator TT and returns an equivalent S-expression number in accumulator A. The value in TT is not preserved. No other accumulators are disturbed. Another name for FXCONS is FIX1A; they are entirely equivalent. Note that lisp fixnums are represented in such a way that the address in A will point to a word containing what was in TT.

(JSP T FLCONS)
Similar to FXCONS, but takes a floating-point machine number in TT, and returns a lisp flonum in A.

(JSP T FXNV1)
Verifies that the S-expression in accumulator A is a fixnum; if it is not, a correctable .wrng-type-arg error is signaled. If it does contain a fixnum, or if the error break eventually returns a fixnum, then it returns with the equivalent machine fixnum in accumulator TT. This routine is useful primarily for the error checking; if it is already known that A contains a lisp fixnum, the instruction (MOVE TT 0 A) serves just as well. Such knowledge, for example, can be derived from declarations by the fast-arithmetic compiler.

(JSP T FXNV2)
(JSP T FXNV3)
(JSP T FXNV4)
Similar to FXNV1, but take arguments and return machine fixnums in different accumulators:
```
FXNV2    B   -> D
FXNV3    C   -> R
FXNV4    AR1 -> F
```

There is no FXNV5 - you must move an argument in AR2A into some other accumulator first.

(JSP T IFIX)
Takes a machine flonum in TT and converts it to a (truncated) machine fixnum, returned in TT. Destroys accumulator D.

(JSP T IFLOAT)
Takes a machine fixnum in TT and converts it to a machine flonum, returned in TT. Does not destroy any other accumulators.

(JRST 0 FIX1)
(JRST 0 FIX2)
(JRST 0 FLOAT1)
(JRST 0 FLOAT2)
These are convenient exits to the following code internal to the lisp system:

```
FIX2    (JSP T IFIX)
FIX1    (JSP T FXCONS)
        (POPJ P)
FLOAT2  (JSP T IFLOAT)
FLOAT1  (JSP T FLCONS)
        (POPJ P)
```

(JSP T FLTSKP)
Verifies that the S-expression in A is a fixnum or flonum; if it is not, a wrng-type-arg error is signaled. If it is, then the machine number is returned in accumulator TT; moreover, the return skips if it is a flonum. Example: here is a simplified version of the sub1 function which does not accept bignums:

```
(LAP SUB1NOBIG SUBR)
(ARGS SUB1NOBIG (NIL . 1))
        (JSP T FLTSKP)
        (SOJA TT FIX1)
        (FSBRI TT 0 1.0)
        (JRST 0 FLOAT1)
NIL
```

(JSP T (NPUSH -<n>)) This routine pushes <n> nils onto the regular pdl; i.e. it is equivalent to writing (PUSH P (214 0 0 NIL)) <n> times. <n> must be between 1 and 20 octal. Note the minus sign in the above: to push 4 nils one writes (JSP T (NPUSH -4)). This routine is used greatly by compiled code to create pdl slots for local variables.

(JSP T (0PUSH -<n>))
Similar to NPUSH, but pushes zeros onto the fixnum pdl. <n> must be between 1 and 10 octal. Used by code produced by the fast-arithmetic compiler.

(JSP T (0*0PUSH -<n>))

Similar to NPUSH, but pushes zeros onto the flonum pdl. <n> must be between 1 and 10 octal. Used by code produced by the fast-arithmetic compiler.

(JSP D *LCALL)

This routine is called by user lsubrs produced by the lisp compiler. It accounts for the number of arguments, and saves some information so that the arg and setarg functions can find the arguments. After the user lsubr has been executed it takes care of popping the arguments off the pdl and returning to the caller.

(PUSHJ P IOGBND)

Used by compiled code to perform the iog function. Equivalent to the code

```
(JSP T SPECBIND)
(0 0 (SPECIAL ^W))
(0 0 (SPECIAL ^Q))
(0 0 (SPECIAL ^R))
(0 0 (SPECIAL ^B))
(0 0 (SPECIAL ^N))
```

(JSP T (*MAP -<n>))

Used by compiled code to call the various mapping functions in the common case where there are two arguments. The function should be in B, and the list in A. (This is backwards from the standard order!) <n> determines which mapping function as follows:

1	maplist	3	map	5	mapcon
2	mapcar	4	mapc	6	mapcan

(JSP T *SET)

Used for compiling calls to the function set. Accumulator A should have the value (second argument to set), while AR1 should have the atomic symbol which is to get the value (first argument to set).

(JSP T *STORE)

Used for compiling calls to the function store. (The conventions for this routine are undergoing some change, and thus are not described here.)

(JSP T *NSTORE)

This is to the nstore function as *STORE is to store.

(PUSHJ P *UDT)
Used by compiled code for handling undefined computed go tags in compiled progs. The tag is in accumulator A. It handles the case where the tag is really a fixnum; and if not, signals a correctable error and possibly returns with a corrected tag in A.

(JSP TT ERSETUP)
Used for compiling calls to the function ERRSET. Accumulator A has the second argument to ERRSET, and B has the address to go to if an error occurs. This routine pushes various things onto the regular pdl.

(JRST 0 ERUNDO)
If all the code compiled for the first argument to an errset runs without error, it must go to ERUNDO to undo the errset, i.e. to pop the things off the pdl which ERSETUP pushed. Control is returned to the address given in B when ERSETUP was called.

(JSP T GOBRK)
Used by compiled code when a go is done within an errset (yech!). It is similar to ERUNDO, but returns to the instruction following the (JSP T GOBRK), rather than to the place specified to ERSETUP.

(JSP TT (ERSETUP -1))
Used to compile calls to the function catch, which internally is similar to errset. Accumulator A contains the second argument to catch (the catch tag), and B the return address which is used if a throw is done.

(JRST 0 (ERUNDO -1))
Just as ERUNDO undoes an errset, so ERUNDO-1 undoes a catch.

(JSP T (GOBRK -1))
Similar to GOBRK, but breaks out of a catch rather than an errset.

ARGLOC
This is not a routine but a variable, which contains the address of the pdl slot just below the arguments to the most recently called lexpr or user lsubr, or zero if none has been called. Thus the call (ARG 2) may be coded in lap roughly as:

```
(MOVE T ARGLOC)
(ADDI T 2)
(HRRZ A 0 T)
```

This is one of the variables set up by *LCALL.

ARGNUM
This, like ARGLOC, is a variable. It contains the number of arguments to the most recent lexpr or user lsubr call, as a lisp number. (accessing ARGNUM indirectly will of course fetch the machine number.) Thus one might write a function:

```
(LAP ARGN-2 SUBR)
(ARGS ARGN-2 (NIL . 0))
        (MOVE TT @ ARGNUM)      ;get number of args
        (CAIGE TT 3)            ;need at least 3
        (LERR 0 (% SIXBIT LESS THAN 3 ARGS))
        (ADD TT ARGLOC)         ;fetch the last
        (HRRZ A -2 TT)          ; arg but 2
        (POPJ P)
NIL
```

14.5.8 - Routines For Use by Hand-Coded LAP

There are some routines internal to pdp-10 lisp which are not used by code produced by the compiler, but which may be of use to those writing functions in lap. Unless specified otherwise, the symbols for these routines are also predefined to lap.

(PUSHJ P PRINTA)
This routine is the internal lisp print function. It does not actually perform any output, but merely supplies a stream of characters. It is called with the S-expression to be printed in accumulator A, and the address of a routine in R. The sign bit of R controls the use of slashes: zero means produce characters like prin1 and explode would, one means like princ and explodec. PRINTA will generate characters and pass them one at a time to the routine specified in R by placing the ascii code in accumulator A and doing a (PUSHJ P 0 R). (This violates the rule about putting non-S-expressions in gc-protected accumulators, but for numbers less than about 2000 octal this is guaranteed to be a safe procedure anyway.) The routine may do anything it wants to with the character, but must avoid destroying the contents of accumulators B, C, TT, and R, which are assumed by PRINTA to be safe. On the other hand, AR1 and AR2A are not altered by PRINTA and may be used to communicate over successive calls to the routine; e.g. they may hold byte pointers, etc. (Again, a violation of the rule, but this is all right as long as they point to "safe" places, like pdl slots or binary code.) When PRINTA is done it will return to the instruction after the PUSHJ to it. The contents of accumulator A are not preserved. Example: Here

is a funny version of flatc which only counts capital letters.

```
(LAP ALPHLATC SUBR)
(ARGS ALPHLATC (NIL . 1))
        (PUSH P (% 0 0 FIX1))   ;it's NCALLable!
        (PUSH FXP (% 0))        ;counter
        (MOVEI AR2A 0 FXP)      ;remember where it is
        (HRROI R COUNT)         ;princ style
        (PUSHJ P PRINTA)
        (POP FXP TT)            ;pop count
        (POPJ P)
COUNT   (CAIGE A 101)           ;only count capital
        (POPJ P)                ; letters
        (CAIG A 132)
        (AOS 0 0 AR2A)
        (POPJ P)
NIL
```

(PUSHJ P GETCOR)

This symbol is not known to lap; it is intended primarily for systems programmers on ITS who need large blocks of core for special I/O devices; however, it also exists in dec-10 lisp. It is called with the number of 1K blocks of core desired in TT. Lisp allocates a single block of core that large and returns the address of the first word of the block in TT. It may destroy several other accumulators in the process. Lisp may or may not actually cause the core to exist; it merely allocates address space and promises not to use it for anything else. The caller should do the appropriate .CBLK calls on ITS to cause the core to exist. (On dec-10 lisp will cause the core to exist, for the present.)

INHIBIT

This is a variable which, if non-zero, specifies that (a) user interrupts may not be processed, but must be delayed, and (b) lisp may not relocate any arrays when garbage collecting (it may if the array functions are called, however). This is used primarily by the lisp system; the nointerrupt function is usually sufficient for users. When INHIBIT is reset to zero the routine INTREL should be called, to check for any delayed interrupts which may be pending. Note that INHIBIT does not prevent uncorrectable errors and control G or control X quits. Thus, it is preferable to the nointerrupt function when it is desired to inhibit user interrupts but not quits (such situations are rare except in lap code). The standard usage of this switch is:

```
(PUSH FXP INHIBIT)
(SETOM 0 INHIBIT)
... process with user interrupts inhibited ...
(PUSHJ P INTREL)
```

Note that INTREL will do a (POP FXP INHIBIT).

NOQUIT

This switch inhibits all interrupts and quits. The left half is for use by the garbage collector, and only the garbage collector! The right half may be used by user programs by using (HLLOS 0 NOQUIT) to turn it on, and (HLLZS 0 NOQUIT) to turn it back off. After turning it back off the routine CHECKI should be called to check for any delayed interrupts or quits. Thus the standard usage is:

```
(HLLOS 0 NOQUIT)
... process with NOQUIT non-zero ...
(HLLZS 0 NOQUIT)
(PUSHJ P CHECKI)
```

This is somewhat less useful than the user nointerrupt function, but was implemented first. Note that the routine INTREL described above under INHIBIT is equivalent to

```
(POP FXP INHIBIT)
(JRST 0 CHECKI)
```

and thus if for some reason one wants to pop the old value of INHIBIT oneself, CHECKI may be used instead of INTREL. CHECKI preserves all accumulators.

(PUSHJ P UINITA)

This routine sets things up for opening a file, old I/O style. It takes a file name list (name1 name2 dev user) in accumulator A, and on ITS a mode in the right half of TT. If the file name list is short the default file names are applied as for the uread function. In the dec-10 implementation, the device name is placed at location UTIN, and the ppn in USN (the latter tag is not known to lap; beware!); the file names are returned in T and TT. In the ITS implementation, the mode, device, and file names are placed in a three-word block suitable for .OPEN at location UTIN, and the lisp's sname is set to the appropriate user name. The contents of accumulator A are preserved. UINITA also does the equivalent of

```
(PUSH FXP INHIBIT)
(SETOM 0 INHIBIT)
```

thus locking out user interrupts, on the theory that some I/O operation will take place which should not be interrupted. It is up to the caller subsequently to unlock interrupts, e.g. by doing (JRST 0 INTREL). Example: on ITS, these functions provide a (relatively inefficient) method for binary input (I/O channel 17, presently unused

in pdp-10 lisp, is usurped; beware, for this fact will change!):

```
(LAP BINOPEN FSUBR)
        (MOVEI T 4)             ;image unit input
        (PUSHJ P UINITA)        ;set up
        (*OPEN 17 UTIN)         ;try to open it
        (LER3 0 (% SIXBIT BIN FILE NOT FOUND))
        (JRST 0 INTREL)              ;must unlock interrupts
(ENTRY BINGET SUBR)
(ARGS BINGET (NIL . 0))
        (PUSH P (% 0 0 FIX1)) ;NCALLable!
        (*IOT 17 TT)            ;input a binary word
        (POPJ P)                ;return as a fixnum
(ENTRY BINCLOSE SUBR)
(ARGS BINCLOSE (NIL . 0))
        (*CLOSE 17)             ;close the channel
        (POPJ P)
NIL
```

14.6 - Internal Details of the Multics Implementation

***** TO BE SUPPLIED *****

This page intentionally left blank.

15 - The Trace Facility

The LISP trace package provides the ability to perform various actions at the time a function is called and at the time it returns. This can be used for traditional tracing or for more sophisticated debugging actions.

The trace package is not part of the initial environment; however, it is automatically loaded in on the first reference to the function `trace`.

The lisp trace package consists of three main functions, `trace`, `untrace`, and `remtrace`, all of which are fexprs.

A call to trace has the following form:

(`trace` ***trace specs***)

A ***trace spec*** in turn is either an atom (the name of the function to be traced) or a list:

(<function name> <options>)

where the options are as follows:

`break` <pred> — causes a break after printing the entry trace (if any) but before applying the traced function to its arguments, if and only if <pred> evaluates to non-`nil`.

`cond` <pred> — causes trace information to be printed for function entry and/or exit if and only if <pred> evaluates to non-`nil`.

`wherein` <fn> — causes the function to be traced only when called from the specified function <fn>. One can give several trace specs to `trace`, all specifying the same function but with different `wherein` options, so that the function is traced in different ways when called from different functions. Note that if <fn> is already being traced itself, the `wherein` option probably will not work as desired, probably. (Then again, it might.)

`argpdl` <pdl> — specifies an atom <pdl> whose value trace initially sets to `nil`. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of the arguments is consed onto the <pdl> when the function is entered, and cdr'ed back off when the function is exited. The <pdl> can be inspected from a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own pdl, or one pdl may serve several functions.

entry <list> — specifies a list of arbitrary S-expressions whose values are to be printed along with the usual entry trace. The list of resultant values, when printed, is preceded by a || to separate them from the other information.

exit <list> — similar to entry, but specifies expressions whose values are printed with the exit trace. Again, the list of values printed is preceded by ||.

arg
value
both
nil — specify that the function's arguments, resultant value, both, or neither are to be traced. If not specified, the default is both. Any "options" following one of these four are assumed to be arbitrary S-expressions whose values are to be printed on both entry and exit to the function. However, if arg is specified, the values are printed only on entry, and if value, only on exit. Note that since arg, value, both, nil, swallow all following expressions for this purpose, whichever one is used should be the last option specified. Any such values printed will be preceded by a // and will follow any values specified by entry or exit options.

If the variable arglist is used in any of the expressions given for the cond, break, entry, or exit options, or after the arg, value, both, or nil option, when those expressions are evaluated the value of arglist will effectively be a list of the arguments given to the traced function. Thus

```
(trace (foo break (null (car arglist))))
```

would cause a break in foo if and only if the first argument to foo is nil. Similarly, the variable fnvalue will effectively be the resulting value of the traced function; for obvious reasons, this should only be used with the exit option.

There exists a version of the trace package called strace. On the AI and ML pdp-10s it is available in the COMLAP directory. On Multics, it is available in the same directory as lisp. It looks exactly like the normal trace package, except that one extra option is available:

grind — specifies that any trace output is to be done, not with the usual call to print, but through the sprint function from the grind package; thus, trace output for that <trace spec> will be "pretty-printed".

This feature is not included in the regular trace package because it really eats up free storage.

Examples of calls to trace:

(1) To trace function foo, printing both arguments on entry and result on exit:

```
(trace foo) or (trace (foo)) or (trace (foo both)).
```

(2) To trace function foo only when called from function bar, and then only if (cdr x) is nil:

```
(trace (foo wherein bar cond (null (cdr x))))

or (trace (foo cond (null (cdr x)) wherein bar))
```

As this example shows, the order of the options makes no difference, except for arg, value, both, or nil, which must be last.

(3) To trace function quux, printing the resultant value on exiting but no arguments on entry, printing the value of (car x) on entry, of fool, foo2, and (foo3 bar) on exit, and of zxcvbnm and (qwerty shrdlu) on both entry and exit:

```
(trace (quux entry ((car x)) exit (fool foo2 (foo3 bar))
        (qwerty shrdlu)))
```

(4) To trace function foo only when called by functions bar and baz, printing args on entry and result on exit, printing the value of (quux barf barph) on exit from foo when called by baz only, and conditionally breaking when called by bar if a equals b:

```
(trace (foo wherein bar break (equal a b))
        (foo wherein baz exit ((quux barf barph))))
```

(5) To trace functions phoo and fu, never printing anything for either, but saving all arguments for both on a common pdl called foopdl, and breaking inside phoo if x is nil:

```
(trace (phoo argpdl foopdl break (null x) cond nil nil)
        (fu argpdl foopdl cond nil nil))
```

The "cond nil" prevents anything at all from being printed. The second nil in each <trace spec> specifies that no args or value are to be printed; although the cond nil would prevent the printout anyway, specifying this too prevents trace from even setting up the mechanisms to do this.

(6) To trace function foobar, printing args on entry and result on exit, plus the value of moby-expr on exit, and pretty-printing the output:

```
(trace (foobar grind exit (moby-expr)))
```

trace returns as its value a list of names of all functions traced; for any functions traced with the wherein option, say (trace (foo wherein bar)), instead of returning just foo it returns a 3-list (foo wherein bar). If

trace finds a <trace spec> it doesn't like, instead of the function's name it returns a list whose car is ? and whose cdr indicates what trace didn't like. A list of possible error indications:

(? wherein foo) trace couldn't find an expr, fexpr, or macro property for the function specified by the wherein option.

(? argpdl foo) the item following the argpdl option was not a non-nil atomic symbol.

(? foo not function) indicates that the function specified to be traced was non-atomic, or had no functional property. (Valid functional properties are expr, fexpr, subr, fsubr, lsubr, and macro.)

(? foo) foo is not a valid option.

Thus a call to trace such as

```
(trace (foo wherein (nil)) (bar argpdl nil))
```

would return, without setting up any traces,

```
((? wherein (nil)) (? argpdl nil))
```

If you attempt to specify to trace a function already being traced, trace calls untrace before setting up the new trace. If an error occurs, causing (? something) to be returned, the function for which the error occurred may or may not have been untraced. Beware!

It is possible to call trace with no arguments. (trace) returns as its value a list of all functions currently being traced.

untrace is used to undo the effects of trace and restore functions to their normal, untraced state. The argument to untrace for a given function should be essentially what trace returned for it; i.e. if trace returned foo, use (untrace foo); if trace returned (foo wherein bar) use (untrace (foo wherein bar)). untrace will take multiple specifications, e.g. (untrace foo quux (bar wherein baz) fuphoo). Calling untrace with no arguments will untrace all functions currently being traced.

remtrace, oddly enough, expunges the entire trace package. It takes no arguments.

16 - Formatted Printing of LISP Data

Pretty-printing, also called "grinding," is the conversion of list structure to a readable format. This chapter outlines the computational problems encountered in such a task and documents the current algorithm in use. This chapter was taken from AI memo 279 by Ira Goldstein.

The "grind" package is used to print out S-expressions in a more readable form than that provided by the standard functions `print`, `prin1`, and `princ`. The grind package is accessed through three functions: `grindef`, `grind`, and `grind0`.

`grindef` is used to grind the properties of an atom; usually the functional properties `expr`, `fexpr`, and `macro`, and the atom's value. The output of `grindef` is legible and is also readable back in to lisp.

`grind` and `grind0` are used to grind up an entire file. Comments in the file are preserved, but everything is rearranged for readability. The file may contain control lines beginning with ";;*", which are ignored when the file is read into lisp, but are used to specify parameters to grind.

The grind package is not part of the initial environment; however, it is loaded in on first reference to one of the functions `grind`, `grindef`, `grind0`.

16.1 - Introduction

Pretty-printing is a fundamental debugging aid for LISP. List structure presented as an unformatted linear string is very difficult for a person to understand. The purpose of pretty-printing is to clarify the structure of a LISP expression. The simplest class of pretty-printers accomplishes this by the judicious insertion of spaces and carriage returns.

The new grind package differs from earlier ones in providing a larger number of formats in which S-expressions and comments can be ground. A variety of predefined formats exist which can be associated with any LISP function. For unusual formats, the user can design his own procedures to control grinding.

16.2 - Top Level Functions

16.2.1 - grind and grind0 - fexprs

`grind` and `grind0` convert files to pretty-printed form. Their input format is the same as that of the LISP file manipulating functions like `uread` and `uwrite`.

(`grind` *filename1 filename2 device uname*)

`ufile`'s a pretty-printed form of the file under the same name. The usual LISP conventions for default device, user and file names are used. To avoid possible disasters, use ">" as your second file name. `grind0` does not `ufile`. Hence, it is useful for filing the pretty-printed file under a different name. For example,

```
(grind0 geo > dsk ira) (ufile geo print)
```

results in the pretty-printed version being filed as GEO PRINT.

In the Multics implementation, the form

(`grind` *path1 path2*)

may also be used. *path1* and *path2* must be strings. `grind` will take its input from the segment named *path1* and put its output in the segment named *path2*. *path2* may be omitted, in which case the output is sent to a segment named `!grind.output` in the working directory.

16.2.2 - grindef - fexpr

`grindef` takes atoms as arguments. It then pretty-prints their expr, fexpr, and macro properties, and their values. For example,

```
(grindef program1 program2)
```

pretty-prints these two LISP functions.

The default properties pretty-printed by `grindef` can be modified in two ways.

(`grindef` <list of additional properties> <atom1> <atom2> ...)

appends the additional properties to the list of default properties for the duration of the current call to `grindef`. A permanent change to the default properties pretty-printed by `grindef` is made by setting the atom `grindproperties` to a new list of properties.

(grindef) will repeat the last call to grindef. This saves typing when repeatedly grindef'ing the same functions.

16.2.3 - Formatting

The pretty-printer can be programmed in the following ways:

a. (<grind-control-fn> <arguments>) executes the grind-control-fn on the given arguments. A typical grind control function is programspace. (programspace 80) sets the width available for pretty-printing code to 80. Complete documentation follows in section 16.3.

3. (<grindfn or grindmacro> <function> <grind-format>) assigns the grind-format to the function as either a grindfn or grindmacro. Whenever the pretty-printer encounters the function as the first element of a list, the list is printed using the special format. The grind-format can either be the name of a function of no inputs or the body of a lambda definition. A variety of predefined formats such as prog-form are described in the next section. The mechanism for building new formats is presented in section 16.5.1.

c. (unformat <function>) removes any special grindfn or grindmacro properties of the function.

For all of the above specifications, <function> can be replaced by <list of functions>. The grind specification is then applied to each function in the list.

There are two ways in which format statements may be delivered to grind. One is to insert them directly into the file being ground as

```
;;*(grindfn thprog prog-form) (programspace 80.) <newline>
```

Comments beginning with ";;*" cause the pretty-printer to evaluate the remainder of the line. If the line consists of only a single S-expression, the toplevel parentheses are optional.

```
;;*grindfn thprog prog-form
```

The normal LISP read-eval-print loop ignores semi-colon comments. Hence, ;;* comments only have effect when the file is ground.

The second way to deliver format statements to the grind package is to place them in the file which the grind package automatically reads when it is loaded. On ITS this file is called

```
GRIND (INIT)
```

and is located in the user's directory. On Multics this file is called

`start_up.grind`

and it is located in the user's home directory.

16.2.4 - remgrind - fexpr

(remgrind) removes all of the grind package's functions from the LISP environment. Alternatively, the user can be more selective in pruning the space occupied by the grind package by erasing only those features he does not need. This is done as follows:

(remgrind file) - erases grind and grind0. Useful when only grindef is needed.

(remgrind ucontrol) - erases the formatting functions. It does not erase those special formats already defined by the user. But it prevents him from defining any more. Useful after the user has created his special formats.

(remgrind format) - erases both the formatting functions as well as all special formats.

(remgrind semi) - erases special functions for handling semi-colon comments.

16.2.5 - Functions, Atoms, and Properties Used by Grind

***** [THIS SECTION IS QUITE OUT OF DATE] *****

Many functions and atoms are used by the grind package. In addition, the property-list indicators grindfn and grindmacro are used for specifying special grind formats.

The following atoms are reserved by the grind package (at the time of this writing.)

```
/:  /:/:  /:/:? ^d arg chrct comnt comspace fill form gap grindef
grindfn  grindlinct  grindmacro  grindpredict  grindproperties
grindreadtable  grversion  h  l  linel  m  macro  n  noff  nomerge
oldcontrollstatus oldsemistatus pagewidth predict prog?  programspace
readtable remsemi semistatus topwidth
```

The following function and array names are used by grind:

```
block-form  block-predict comment-form comspace def-form done?  fill
flatdata form gblock gerror gflatsize  gprinl  grind  grind-unbnd-vrbl
```

grindO grindargs grindef grindfn grindmacro grindpage grindreadtable gtab gzap indent indent-to lambda-form maxpan mem-form merge newlinel nofill nomerge nopredict oldstatus page pagewidth panmax plnr popl ppage pprin predict prin50com prinallcmnt prog-form prog-predict programspace putgrind readmacro readmacroinverse rem rem-realprop rem/; rem/;/; remgrind remsemi semi-comment semi? setq-form setq-predict slashify slashifyl sprint sprintl stat-tab testl tj6 topwidth turpri unformat

16.3 - Predefined Formats

16.3.1 - Standard Formats

The following formats are used by the pretty-printer in the absence of any special formatting instructions. Choice depends on the available width and the cost in number of lines.

a. ***linear-form*** - The expression is printed with no extra insertion of carriage-returns and spaces. This is the format used by the LISP printing primitives. It is used by grind only when there is sufficient width remaining on the line.

b. ***standard-form*** - This is the preferred format for lists beginning with atomic functions. It is also used on other lists if fewer lines are needed to print the code this way.

```
(<function> <pretty-print of arg(1)>
            <pretty-print of arg(2)>
                  .
                  .
                  .
            <pretty-print of arg(2)>)
```

c. ***miser-form*** - This format conserves the space remaining on the line. When in width trouble, function lists are printed this way.

```
(<pretty-print of element(1)>
 <pretty-print of element(2)>
             .
             .
             .
 <pretty-print of element(n)>)
```

d. ***funny-form*** - Occasionally, this format decreases the number of lines needed to print an expression. It is used whenever this is the case. If predict is nil, computation is saved by ignoring it.

```
(<element1> <element2> ... <pretty-print of elementN>)
```

16.3.2 - Special Grindfns

Each of the following grind-formats can be assigned to any function by:

```
(grindfn <function> <grind-format>)
```

a. ***block-form*** - the entire expression is ground as text where the left margin follows the opening parenthesis of the expression. For example,

```
                (A B C D E F G
H I J K L M N
O P Q R S T U
                V W X Y Z)
```

Typically, argument lists and planner patterns are ground as blocks.

b. ***def-form*** - def-form is the standard format for grinding definitions. The "defun", function-name, indicators, and argument list are always ground on the first line. The argument list is ground as a block. The remaining elements of the definition are ground as a "body", i. e. depending on their size, they are ground one under the other in:

i. either the space remaining on the line, e. g.

```
(defun fcn-name <arglist ground as block> ******
                                          ******
                                          ******)
```

ii. in standard format, i. e. aligned under the function name:

```
(defun fcn-name indicator <arglist ground as block>
       ******
       ******
       ******)
```

iii. or in miser format, i. e. aligned under the defun:

```
(defun fcn-name indicator <arglist ground as block>
 ******
 ******
 ******)
```

c. ***lambda-form*** - the lambda and its variable-list are ground on the first line. The variable-list is ground as a block. The remaining elements of the lambda are ground as a "body" i. e. depending on their size, and in order of preference,:

i. in either the space remaining on the line, e.g.

```
(lambda <variable-list ground as block> ******
                                        ******
                                        ******)
```

ii. in standard format:

```
(lambda <variable-list ground as block>
        ***********
        ***********)
```

iii. or in miser format:

```
(lambda <variable-list ground as block>
 *************
 *************)
```

d. ***prog-form*** - This format used for progs is similar to ***lambda-form***, except that tags are unindented.

e. ***mem-form*** - The first argument is ground as code. The remainder are also ground as code unless quoted, in which case, they are ground as a block. For example,

```
(member x
        '(a b c d e f g h i j k l
          m n o p q r s t u v w x
          y z))
```

By default, member, memq, the map functions, and the assoc functions are ground in this format.

f. ***comment-form*** - The cdr of the expression is ground as a block. For example,

```
(comment this is a very long
         comment that takes
         several lines)
```

comment and *fexpr, *expr, *lexpr, **array, special, and unspecial clauses of declare's are ground in this format.

g. ***setq-form*** - Space permitting, variables and values are ground as pairs. For example,

```
(setq a (plus 1 1)                     b 0)
```

If there is insufficient space, standard or miser format is used.

16.3.3 - Inverting Read Macros

Quote-type read macros can be inverted when pretty-printed.

```
                 reader                        grind
&lt;char&gt; &lt;expr&gt; - - -> (function &lt;expr&gt;) - - -> &lt;char&gt; &lt;expr&gt;
```

This is accomplished via the `readmacro` function:

(readmacro <function> <macro character or characters>)

The macro character is `princ`'ed and then the <expr> is pretty-printed. Two examples are:

```
(readmacro quote /')
        and
(readmacro thv /$/?)
```

16.3.4 - System Packages

A package of special formats currently exists for MICRO-PLNR. To utilize them, place either (PLNR) in your GRIND (INIT) file or ;;*PLNR directly in your micro-plnr files.

16.4 - Comments

Semi-colon comments are defined as a semi-colon followed by text and concluded by a carriage return. These comments can be inserted anywhere in an S-expression or appear alone at the top level. They are completely ignored by the LISP reader. The grind package pretty-prints these comments in several formats depending on whether the comment begins with 1, 2 or 3 semi-colons.

16.4.1 - Single Semicolons

Comments beginning with a single semi-colon are printed to the right of the code. Sequences of single-semi's are merged. The code is normally ground in the first 70 spaces of the line (programspace) while the single semi's are ground in the final 49 spaces (comspace). gap = 1 is the space between code and comments.

```
-----programspace-----    --gap--    ------comspace-----
          70                 1               49

--------------------pagewidth  =  120-----------------
```

These values can be altered, for example, by inserting the following comment into a file:

```
;;*(pagewidth 120. 89. 1. 30.)
```

This results in programspace becoming 89, gap 1, and comspace 30.

For code that contains no single semi's, a programspace of 80. is preferable.

16.4.2 - Double Semicolons

These comments are printed as part of the code with the proper indentation. Sequences of double semi's are merged. At the top level, topwidth = pagewidth is used. Inside code, double semi's are limited to programspace. To alter topwidth, execute:

```
(topwidth <new-value>)
```

16.4.3 - Triple Semicolons

";;;..." are similar to ";;..." with respect to indentation. However, they are otherwise not modified by grind. Spaces are not filled and sequences of comments are never merged. They are thus useful when the user desires his comment to be printed exactly as originally typed.

16.5 - Grind Control

These functions set various switches and variables for the pretty-printer. They may be invoked by the use of ";;*" comments. For example, a comment like

```
;;*comspace 43.
```

will cause `(comspace 43.)` to be evaluated when the file is ground.

1. `fill` causes multiple spaces appearing in single and double semi's to be merged. Periods ending sentences are followed by two spaces. This is the default case.

2. `nofill` causes multiple spaces to be treated as such. Triple semi's are always nofill'ed.

3. `merge` causes double semi's to be merged, if sufficient comspace remains on the line.

4. `nomerge` causes double semi's not to be merged. This is the manner in which triple semi's are handled. The full pagewidth is used.

5. `page` causes the output of a formfeed character.

6. `ff` causes grind to insert formfeeds approximately every 60 lines. Formfeeds are only inserted at the toplevel, never appearing within S-expressions. This is the default case.

7. `noff` limits the insertion of formfeeds to explicit calls of page.

8. `ppage` causes grind to preserve original paging of user's file.

9. `nopredict` - This switch makes the grind dumber but faster. The algorithm no longer considers as many alternatives for grinding each expression. For PROG-FORM and DEF-FORM, format 1 is no longer considered. Similarly, FUNNY-FORMAT is never considered. Dumb mode is the default state.

10. `predict` - All of the formats discussed in the previous pages are considered.

11. `pagewidth` <pagewidth> <programspace> <gap> <commentspace>

12. `programspace` <value> - resets the value of the programspace. Enlarging programspace shrinks comspace.

13. `comspace` <value> resets the width used for single semi comments. The tradeoff is again with the programspace.

14. `topwidth` <value> - resets the width used for toplevel double semi comments.

16.5.1 - Defining New Formats

The user may wish to go beyond the predefined formats discussed in section 16.2. To do this, grindfn can be used to define special grind functions (SGF's) of his own design. The syntax is as follows:

```
(grindfn <atom or list of atoms> <grind-format>)
```

where the definition is either the name of a 0-input procedure or the body of a lambda expression.

Grindfns are processed as follows: assume the atom L1 has a SGF associated with it. Then, whenever expressions of the form (L1 ... LN) are encountered, grind prints "(" and then transfers control to the definition of the SGF. Upon entering the SGF, the following free variables are relevant:

L <--- (L1 ... LN)
N <--- CHRCT = remaining line width, following the "(".

Note that these variable names are lower case in the actual code but are printed in upper case here for the sake of increased legibility.

A SGF generally processes some initial segment of L, cdr'ing L in the process. Note that the SGF must at least process L1. Upon completion, if L has been set to nil, grind simply prints the closing parenthesis ")". If, on the other hand, L has been rebound to some terminal segment of itself,

L = (Li ... Ln)

then grind prints the remainder of L as the body of a ***def-form***, i.e. the elements of L are printed one under the other in either

a. the space remaining on the line
b. aligned under L2
or c. aligned under L1.

16.5.2 - Vocabulary

The following vocabulary is useful for defining SGF's:

1. (remsemi) - expr - This function processes any ; comments that occur as initial elements of L, cdr'ing L in the process.

2. (pprin S F) - expr - S is printed in the format specified by F where F can be:

'line - equivalent to prinl
'block - block-form
'list - comment-form

'code - applies pretty-printer to S.

pprin should not be given ; comments as input. (remsemi) is generally used to avoid this. pprin does not print a space following S.

3. (form F) - expr - This function is designed to relieve the user of an explicit concern for comments. It also frees him from printing spaces between elements of L. Its definition is:

```
(remsemi)
(pprin (car l) f)
(and (setq l (cdr l))
     (princ '/ ))
```

Its action is to first apply remsemi, removing any initial comments from L. It then pretty-prints (car L) in the specified format F. Finally it cdr's L and prints a space if there is still more to go.

4. (turpri) - expr - A carriage return is printed. terpri should not be used.

5. (indent-to N) - expr - This function causes chrct to be set to N by printing a carriage return if necessary (N > chrct) and spaces. Note that chrct is the current width. This number is equal to the indentation subtracted from the total line width. A common bug is to treat N as the indentation.

6. (indent M) - expr - M spaces are printed. An error results if M exceeds the space remaining on the line.

7. (popl) - expr - L is set to (cdr L). Then remsemi is applied. The net result is to cdr L until its car is not a comment.

8. a. (testl) - lexpr - returns the first element of L that IS NOT a ";" comment.
   b. (testl j) - returns the jth element of L that is not a comment.
   c. (testl j t) - returns the entire remainder of L beginning WITH the jth element.

9. (semi? K) - expr - returns t only if K is a semi-colon comment.

16.5.3 - Examples

Following are some examples of SGF's. Lambda's are ground by default in *def-form*. The user could achieve the same effect by defining the following SGF:

```
1          (grindfn lambda (form 'line)
2                          (form 'block))
```

(form 'line) in line 1 prints lambda and pops L. (form 'block) in line 2 prints the argument list of the lambda in *block-form* and again pops L. Control is then returned to grind and the remainder of the lambda is printed as a body.

Another example might be where the user wishes to grind all expressions of the form:

```
(defprop <atom> <definition> <expr, fexpr, or macro>)
```

as defun's. This would be done by:

```
 1 (grindfn defprop
 2         (cond ((memq (testl 4) '(expr fexpr macro))
 3                (setq l
 4                      (append (list 'defun (testl 2))
 5                              (cond ((eq (testl 4) 'expr)
 6                                     nil)
 7                                    ((list (testl 4))))
 8                              (cdr (testl 3)) ))
 9                (def-form))
10               ((form line)) ))
```

The memq of line 2 checks for whether the indicator is a function property. If so, L is redefined as the appropriate defun:

(cadr L) = function name
The cond of line 5 puts fexpr/macro into the defun
(cdr (caddr L)) is the argument list of the function
(cddr (caddr L)) is the body of the function

and then ground in *def-form*. If not, defprop is printed and control is returned to grind.

Finally, consider a function called cmeans whose arguments are property lists. It is to be ground as follows:

```
(cmeans
   ((<ind-11> <grind prop-11>
              .
    <ind-1n> <grind prop-1n>)
              .
              .
   ((<ind-m1> <grind prop-m1>
              .
    <ind-mn> <grind prop-mn>))
```

Suppose the additional subtlety is desired that properties with indicator foo are ground as blocks while all other properties are ground ordinarily as code. The following SGF achieves this format.

```
(grindfn cmeans (prog nil
1                  (form 'line)
2                  (setq n (- n 4.))
3                  (remsemi)
4             a    ((lambda (1)
5                          (prog nil
6                                (indent-to (addl n))
7                                (princ '/()
8                          b     (remsemi)
9                                (indent-to n)
10                               (cond ((eq (car 1) 'foo)
11                                      (form 'line)
12                                      (form 'block))
13                                     ((form 'line)
14                                      (form 'code)))
15                               (and (testl) (go b))
16                               (princ '/))
17                               (remsemi)))
18                  (car 1))
19                 (cond ((popl) (go a)))))
```

Line 1 prints cmeans. Line 2 establishes the indentation of the arguments of cmeans. Line 3 processes any comments preceding the first argument. Line 4 binds the special free variable L to the current argument of cmeans for use by form and rem. Line 6 indents for the current argument. Line 8 processes any initial comments embedded in the argument. The cond of line 10 forks depending on whether or not the indicator is "foo". In line 15, testl returns nil if L contains no more indicator-property pairs. Line 16 prints the closing parenthesis. 17 processes any remaining comments. By line 19, the current argument of cmeans has been ground. Hence, L is popped. If there are no more arguments, popl returns nil and the SGF is done.

16.5.4 - Grindmacros

A grindmacro differs from the above grindfunctions in that the grind package takes nothing for granted. It does not automatically print the opening parenthesis, the balance of L and the closing parenthesis. If the grindmacro function returns t, then the pretty-printer does nothing more on L. The assumption is that the grindmacro has done all the work. This would be the case for a grindmacro for quote:

```
(grindmacro quote (princ '/')
                  (pprin (cadr 1) 'code)
                  t)
```

Alternatively, if the grindmacro returns nil, the pretty-printer prints L as though nothing had happened. This mode is useful for a grindmacro used to print "index" information as comments preceding the S-expression.

Grindmacros can be defined similarly to grindfns.

```
(grindmacro <atom or list of atoms> <grind-format>)
```

Again the definition can be either the body of a lambda or a function of 0 inputs.

17 - The LISP "Indexer"

TO BE SUPPLIED

This page intentionally left blank.

18 - The LISP Editor

At present the editor exists only in the pdp-10 implementation of lisp. It is quite likely that a different editor will be substituted in the near future. You have been warned!

Evaluating (edit) enters edit mode, wherein commands are given similar to teco commands, action is taken on some expression currently in the working space, and a window around the pointer is printed out after every command execution. (edit t) enters edit mode but does not type out the window after every command. (the p command will cause printing of the window - useful when at a teletype). In this chapter the "$" character represents <altmode>. Commands are:

Q<space> exit from the editor back to lisp.

Y<space>atom<space> causes the function property of atom to be brought in for editing.

YP<space>atom<space>prop<space> yank in the prop property of the atom atom.

YP<space>atom<space>$$<space> yanks the whole property list of atom.

J<space> causes the pointer (which is printed out as $$) to jump to the top of the working expression.

S<space>el . . . en<space>$$<space> searches for an occurrence of the sequence of S-expressions el . . . en and moves the pointer just to the right if successful. Note that the lisp reader is used for read-in by the editor, so that the atom <altmode><altmode> must be followed by some atom terminating character (such as <space>).

I<space>el . . . en<space>$$<space> inserts at the current pointer position the sequence el . . . en

K<space> kills the S-expression just to the right of the pointer, and saves it as the value of the atom $$.

IV<space>exp inserts the value of the S-expression exp. especially useful when inserting stuff deleted from some prior point.

EV<space>exp merely evaluates exp.

Henceforward, <space> will not be explicitly written out, but will be understood to be the command termination character. The next group of commands admit an optional numeric argument (base 10.), preceeding the command, to be interpreted as a replication number:

F move forward (rightwards) past one token. A token is either a parenthesis or an atom.

C same as F

-B same as F

B move back (leftwards) over one token.

-C same as B

-F same as B

R move rightwards past the next S-expression.

L move left over one S-expression.

D move "down" into the first non-atomic S-expression to the right of the pointer.

U move "up" out of the S-expression containing the pointer.

K kill also admits a replication number.

PW the argument is not really a replication number, but rather the "window width" in number of tokens.

The following little used commands may possibly be of some interest:

(insert a virtual open parenthesis.

) insert a virtual close parenthesis.

D(virtually delete an open parens

D) virtually delete a close parens

() restructure the working S-expression according to the virtual parentheses and deletions.

Variables Used by the Editor

`edit` VARIABLE

The value of the variable `edit` is a list of the properties considered functional by the editor. It is used by the Y command.

`$$$` VARIABLE

The value of the atom <altmode><altmode><altmode> is a push-down list of "edit cursors."

Function used by the Editor

`edit` SUBR no args

Typing `(edit)` causes the editor to be entered.

This page intentionally left blank.

Appendix A - Glossary

GLOSSARY

a-list pointer

An a-list pointer is a number which can be passed as an extra argument to eval or apply to indicate a particular binding context in which variables will be evaluated. It is similar to, but not the same as, a pdl pointer. An a-list pointer may also be nil, which indicates the global or "top level" binding context. The name "a-list pointer" is of historical meaning only.

abbreviation

Abbreviation is a feature which allows large lists to be truncated when printed out. See section 13.7.

alarmclock

A facility by which the user can specify a function to be called after a specified amount of time has elapsed. "Time" may be measured as real elapsed time or as CPU run time.

altmode

A character used in the pdp-10 implementation of MACLISP. Also called escape or prefix.

application

Application consists of "applying" a function to a list of arguments and obtaining the value of the function for those arguments. Application is explained in detail in chapter 3. cf. evaluation.

argument

An argument is an object which is given to a function to operate on. Part of the process of evaluating a form consists of deriving arguments from the form, which is just a list of items. For example, when the form (foo 3 4) is evaluated the arguments to the function foo are 3 and 4.

arithmetic

MACLISP contains functions to perform arithmetic operations on integers of arbitrary size and on floating-point numbers with a precision of about eight decimal places. See chapter 7 for a discussion of these functions.

array

An array is an ordered set of cells. Each cell may contain a LISP

object. The name of the array is also the name of an accessing function which when given subscripts as arguments, returns the contents of the cell selected by the subscripts. The function store may be used to assign values to the cells of an array. LISP arrays are similar to FORTRAN arrays except that the subscripts begin at 0 instead of 1, and they are more general because the occupants of the several cells need not all be of the same type.

ascii

"ascii" is the American Standard Code for Information Interchange. This is the character code used internally by MACLISP.

assignment

A value may be assigned to a variable in two ways: 1) by using the function setq, which is similar to the assignment statement in some other languages, and by the related functions set and makunbound. 2) by "binding," accomplished by lambda, prog, or do. Binding is a local assignment. When control leaves the function which caused the binding, the value of the variable is restored to what it was prior to the binding.

association list

An association list is a list of dotted pairs, often used with the functions assoc, assq, sassoc, and sassq. For example, ((a . 4) (b foo 1) (x . y)) associates a with 4, b with (foo 1), and x with y.

atom

An atom is a LISP object which is usually thought of as indivisible. Atoms come in several types: fixnums, flonums, bignums, strings, and atomic symbols.

atomic symbol

An atomic symbol is a type of atom which has a pname, a value cell, and a property list. A pname is a string of characters which identify the symbol. A value cell is a place, associated with an atom, in which any LISP object may be stored. A property list is used to remember named "properties" or "attributes" of an atomic symbol. The value cell allows atomic symbols to be used as variables. The pname and property list make atomic symbols useful as terminal symbols in symbolic manipulation.

autoload

The autoload feature allows the definitions of functions not initially present in the environment to be loaded in from a file automatically when they are required. It is used in the implementation of special utility packages, such as trace, grind, and large application systems.

backtrace

A display of pending evaluations, which can be used in debugging to determine the chain of calls leading to the point of error. The function `baktrace` prints this out.

base

The value of the variable `base` is the radix in which the output routines represent numbers. It is initially 8.

bignum

A bignum is an integer of arbitrarily large magnitude. The arithmetic functions `plus`, `times`, `difference`, etc. use bignums where necessary and automatically manage the varying storage required. For example, bignums make the computation of 1000 factorial easy to write. Because of this power bignum arithmetic is noticeably slower than fixnum arithmetic.

binary program space

In the pdp-10 implementation of MACLISP, an area of memory in which arrays and compiled functions are stored.

binding

A variable may be "bound" to a value by use of `lambda`, `prog`, or `do`. The value of the variable is set temporarily, but will be restored to the previous value when the variable is "unbound." Unbinding happens when the form that bound the variable is exited, whether normally, by an error, or by `throw`.

binding context

The binding context consists of the values of all bound variables. MACLISP includes a partial ability to manipulate binding contexts, the a-list pointer facility, which allows binding contexts to be used as long as control is nested somewhere within them.

Boolean operations

MACLISP includes a full set of Boolean operations on bits. The 36. bits which make up a fixnum may be operated on by the `boole` function. The "and," "or," and "not" operations on logical values are also included, with `t` standing for true and `nil` for false. A variety of predicate functions, which return a true or false (i.e. `t` or `nil`) value, are included.

bound variable

A bound variable is an atomic symbol whose current value was assigned to it by means of a binding. It is something like a local variable.

break level

A level of control at which computation has been temporarily suspended by a breakpoint (q.v.), allowing typein from the console. cf. top level, where typein is allowed from the console because no computation is in progress.

breakpoint

A breakpoint is a point in a program where computation is temporarily suspended and control is returned to the console, enabling the user to explore the state of the computation. Most errors cause a breakpoint, and the trace facility can be used to insert breakpoints. A user can make a breakpoint with the function break.

car

The first member of a dotted pair or a list. The name derives from Contents of Address Register on the IBM 7094, where LISP was first implemented.

catch tag

An object which is used to relate throw's and catch'es.

cdr

The second member of a dotted pair, or the "rest" of a list (i.e. all members except the first.) The name derives from Contents of Decrement Register on the IBM 7094, where LISP was first implemented.

character

One of the 128. ASCII characters. On a typewriter a character is represented by a printed mark or by a formatting operation such as a backspace. Internally a character may be represented as a number, which is the ascii code for the character, or as a character object (q.v.). Characters are also used in pnames and in strings.

character object

An atomic symbol which symbolizes a character. The null character is symbolized by the atomic symbol whose pname is of zero length; each of the other characters is symbolized by an atomic symbol whose pname is that character.

character translation

A feature in the reader which allows characters to be translated to other characters when they are read in. For example, pdp-10 MACLISP uses this feature to translate lower-case letters to upper-case.

charpos

The number of character positions from the left margin. Describes the position of the typing element on a typewriter or the cursor on a display. The notation is extended to files on any device.

chrct

This is an older version of charpos. It is the number of character positions from the *right* margin.

closing a file

Some operating systems require a "cleaning up" operation after all use of a file has been completed. This is called closing the file. The MACLISP garbage collector will usually do this automatically.

comment

Comments are descriptive text, not interpreted by the LISP system, which are inserted into programs for the edification of a reader of that program. In MACLISP there is a comment function, which does nothing with its arguments and so may be used for comments, but a better way to write comments is with the semicolon macro character, which makes everything from it to the end of the line a comment. For example,

```
(foo (bar x))      ;whizzo the frammis.
```

compilation

Compilation is a process which can be applied to a MACLISP function to make it run faster. The cost of compilation is that debugging is made more difficult. Generally debugging is done by interpretation (q.v.)

cons

A cons, also called a dotted pair, is the basic unit for the construction of data structures in MACLISP. A cons contains two members, the car and the cdr, which can be any objects whatsoever.

control characters

Control characters are used to tell the MACLISP system to perform some action immediately, no matter what it is currently doing. See section 12.3.

correctable errors

Most errors in MACLISP are correctable. This means that they cause a user interrupt, which either invokes a user-specified function to correct the error, or causes a breakpoint, which allows the user to determine how to correct it, inform MACLISP of the correction, and continue the interrupted computation.

cross reference

The "index" package may be used to produce "cross references" of LISP programs. See chapter 17.

data types

In MACLISP, objects come in several types, which are explained in chapter 2.

declaration

Declarations are used to give the compiler extra information, not needed by the interpreter, which clarifies the programmer's intent and makes possible the compilation of more efficient code. The function `declare` is provided for this purpose.

debugging

Debugging is the usually long and painful process of finding mistakes (bugs) in programs and removing them. MACLISP provides a number of tools to assist in debugging. See errors, user interrupts, baktrace, breakpoints, trace, the *rset switch, and interpretation.

display slave

The "display slave" is part of the Moby I/O facility in MACLISP. When the pdp-10 implementation of MACLISP is running on the MIT A.I. Lab pdp-10, the display slave may be used to display text and graphics. Extension of the display slave to other sites and implementations is anticipated.

do loop

A clear and concise notation for iterative algorithms, provided by the `do` function in MACLISP.

dot notation

A notation in which a dotted pair is written with parentheses and a period. A dotted pair whose car is a and whose cdr is b is written:

```
(a . b)
```

Any structure of dotted pairs can be written unambiguously, but not necessarily clearly, this way:

```
(((a . b) . c) . (d . e))
```

dotted list

A structure which would be a list except that it does not end in nil. It is written in a hybrid of dot notation and list notation. For example:

```
(a . (b . c))
```

would be written:

(a b . c)

dotted pair

See cons.

edit

The pdp-10 implementation of LISP contains an S-expression editor, described in chapter 18. The Multics implementation does not presently have a built-in editor, however several editors, written in LISP, exist.

end-of-file

When an input file is being read, eventually it comes to the end and some special action may have to be performed. See section 13.2.4.

endpagefn

A function, associated with each output file, which is invoked whenever a new page of output is started.

environment

A LISP environment consists of a complete set of objects, variable values, function definitions, and files, which together make up an application system or a user's current work. Section 12.8.3 describes ways to save the current environment and later resume working with it.

eoffn

A function, associated with each input file, which is invoked when an attempt is made to read past the end of the data in the file.

eq

eq is a function for comparing two objects, which returns t if they are completely identical, nil if they are not. (In machine terms, completely identical means they have the same storage address.) eq is not defined for numbers and strings. cf. equal.

equal

equal is a function for comparing two objects, which returns t if they are similar, nil if they are not. Similar means approximately that they would look the same if printed out. equal works for numbers and strings: numbers are equal if their values are numerically equal, strings are equal if they contain the same characters. Atomic symbols are equal if they are eq. Two dotted pairs are equal if their cars are equal and their cdrs are equal.

errors

Handling of errors in MACLISP is very flexible, in recognition of the fact that errors are a major tool in debugging. See section 12.4.

escape

See altmode.

evaluation

The process by which a form, which may be almost any LISP object, is made to produce a value. Evaluation may involve taking the values of variables and applying functions when a function call is indicated by a list as a form. Evaluation is explained in detail in chapter 3.

expr

An expr is an interpreted function which takes a specific number of evaluated arguments.

fail-act

A catch-all category of errors, which cause a breakpoint to occur. The atom args is bound to useful information about the error.

fexpr

An interpreted function which does not receive its arguments evaluated. (At least it does not evaluate them in the regular way.)

file

A sequence of characters in the external world, and also an object within the lisp environment which is used to communicate with that sequence of characters. See chapter 13.

file name defaults

There is a system of defaults for file names which is intended to increase the convenience of users and programmers. See chapter 13.

file object

An object within the LISP environment which symbolizes a file in the outside world. See file.

fixnum

A fixnum is a type of number, specifically an integer whose absolute value is less than some machine-dependent maximum. In the pdp-10 and Multics implementations, this maximum is 2**35-1, or 34359738367. Compiled code can perform fixnum arithmetic very efficiently.

flonum

A flonum is a type of number, specifically a floating-point number, similar to REAL in FORTRAN, which has machine-dependent range and precision. In the Multics and pdp-10 implementations the range is about 10**-38 to 10**38 and the precision is about 8 decimal digits.

flow of control

The logical sequence in which parts of a program are executed. This includes decision, recursion, iteration, and function calling. In LISP flow of control is generally linear except as otherwise specified, except that the use of functional composition causes the arguments to a function to be evaluated before the function. The functions `cond`, `and`, `or`, `do`, `prog`, `go`, `throw`, and `return` are among those functions used for their effect on the flow of control.

form

A form is an S-expression which is intended to be evaluated. It may be an atomic symbol, an atom such as a string or a number which evaluates to itself, or a list of forms, the first of which is a functional form and the rest of which are argument forms.

formatting

The grind package may be used for the formatted printing of LISP functions or data. See chapter 16.

free storage

In the pdp-10 implementation of MACLISP, free storage is that part of memory set aside for various types of LISP data objects. In some versions the size of this area must be specified when LISP is first entered. The free storage area is managed by a garbage collection algorithm.

free variable

A free variable is an atomic symbol whose current value was not determined by binding within the currently evaluating function. Either it has a global value or it was bound in some function which then called the current function.

fsubr

An fsubr is a machine-language function which does not receive its arguments evaluated. When a fexpr is compiled it becomes an fsubr. A number of the builtin functions, such as `cond`, are fsubrs.

funarg

A functional form, passed as an argument usually, which carries with it the binding context in which it is to be applied. See the

*function function.

function
A LISP object suitable for application. Given arguments, it performs some arbitrary calculation and returns some LISP object as a value. Functions are the fundamental control (and syntactic) structure in LISP.

functional form
Any S-expression which can be used as a function. However, often the term "functional form" is reserved for non-atomic functions, such as lambda-expressions, labels, or random forms which are evaluated to produce a function.

garbage collection
The basic memory management scheme of all LISP implementations. Objects are retained until there are no references to them, at which time since the object can never again be used the storage it occupies can be reclaimed. Reclamation (garbage collection) occurs periodically when the system decides it would be a good thing to do.

gc-daemon
A user interrupt which occurs after each garbage collection, allowing a user-specified function to gain control and monitor the program or make decisions based on the efficiency of storage usage of the program.

gensym
An atomic symbol which has a unique name of the form g0001, g0002, etc. gensym'ed atoms are not "interned," so they cannot be referenced from the console. They are generated by the function gensym.

global variable
A variable which does not currently have a local binding. Its value is whatever value has been assigned to it in the global binding context, for instance if it has been setq'ed at top level.

grind
A package for printing LISP functions and list structure in an indented form that is easy to read. See chapter 16.

ibase
The value of the variable ibase is the radix in which numbers read by the reader will be converted. It is initially 8.

index

A cross-referencing package for LISP programs. See chapter 17.

indicator

An atomic symbol (usually) which serves to label an item in a property list.

input source

The file from which input is taken, when no source is specified. Determined by the variable `infile`.

integer

A type of number which can be represented in MACLISP by either a fixnum or a bignum, depending on how large it is.

intern

When an atomic symbol is read in, it is placed in a special table called the obarray; this is called interning the atomic symbol. The obarray allows the same (according to the function eq) atomic symbol to be used the next time the same pname is read.

interpretation

A method for executing LISP programs in which S-expressions are processed by an interpreting program without preliminary translation. This is the usual mode for execution of lisp. It is more efficient than compilation (q.v.) for evaluating once-only expressions such as directly typed-in input, and for debugging.

I/O

Input/Output, or communication between LISP programs and the outside world. See chapter 13.

iteration

See do loop.

label

1) see prog tag.
2) a type of functional form.

lambda

lambda-expressions are the most common type of (non-atomic) functional form. A lambda-expression is written as a list `(lambda ( <vars> ) <body> )`.

lambda-variables
Variables bound in a lambda-expression are called lambda-variables.

lexpr
An interpreted function which takes a variable number of evaluated arguments. An expr with an atomic symbol in place of the lambda-variables list is a lexpr.

linel
The number of character positions per line.

linenum
The line number, starting from 0 at the top of the page, of the current input or output position in a file.

LISP
A language for list processing and manipulation of symbolic and structured information. The MACLISP dialect of LISP is described in this manual.

list
A data structure in LISP, composed of several conses. The car of each cons is a member of the list, and the cdr of each cons is the next cons, except that the cdr of the last cons is nil, which marks the end of the list.

list notation
A more concise form than dot notation for writing lists. For example,

```
(a . (b . (c . nil)))
```

is `(a b c)` in the list notation.

load
Loading is the process of bringing function definitions, variable values, atomic symbol properties, etc. into the current LISP environment from an outside source, such as a file. See the `load` function.

looping
See do loop.

lsubr
An lsubr is a machine-language function which takes a variable number

of evaluated arguments. A compiled lexpr is an lsubr. A number of the builtin functions, such as plus, are lsubrs.

macro

A type of function which produces as its value a form which is then automatically evaluated to yield the final result of the function call.

macro character

A macro character is a character which, when read, causes a function to be invoked. Macro characters are used to implement complicated special input syntax. The ' character is an example of a macro character.

mapping

A type of iteration in which a function is applied to successive parts of a list. See chapter 10.

moby I/O

A feature in some versions of the pdp-10 implementation of MACLISP by which various peculiar hardware devices may be manipulated by LISP programs.

namelist

A list of atomic symbols which specifies the name of a file in the form of multiple components.

namestring

A character string which specifies the name of a file in implementation-dependent format.

newio

The I/O system described in chapter 13. Some LISPs still use an older I/O system which is less general, described in section 13.5.

newline

The character or sequence of characters used in the host operating system to indicate the separation between lines.

nil

An atomic symbol which indicates "false," "default," or "end of list." nil is a constant since its value is initially nil and cannot be changed.

non-local exits
Escaping from nested function calls without going through the normal function-return mechanism. See catch and throw.

number
See fixnum, flonum, bignum.

obarray
A table of interned atomic symbols, used by the reader to insure that each time a pname is typed in it will refer to the same (according to the function eq) atomic symbol.

object
Any piece of data used by LISP. Programs are also objects.

octal
The number system used by MACLISP, unless some other is specified. Fixnums and bignums are converted for input and output in octal (base 8). Note that flonums are always in decimal.

opening a file
Creating a file object so that a file in the outside world is usable by LISP.

output destinations
Those files to which output is sent if a destination is not explicitly specified. The value of the variable outfiles is a list of the output destinations.

pagel
The number of lines per page in a file.

pagenum
The current page number in a file, starting with 0 at the time it is opened.

pdl
Push-down list or push-down stack. MACLISP uses several pdls internally for binding and recursive evaluation.

pdl overflow
Pdl overflow is what happens when a depth of recursion is used that is more than the implementation can handle. It generally indicates an

error.

pdl pointer
A fixnum which indicates a particular point in a pdl. Pdl pointers are used to denote particular pending evaluations in evalframe and related debugging functions.

pname
The pname, or print-name, of an atomic symbol is a sequence of characters which are typed in or out to denote that symbol.

predicate
A function which tests the truth or falsity of a perticular condition, returning t if it is true or nil if it is false.

prinlength
A variable which can be set to the maximum number of atoms in a list that will be printed before the printer will give up and put "..." Operates under the control of (sstatus abbreviate).

prinlevel
A variable which can be set to the maximum depth of nested lists which will be printed before the printer will give up and put "*". Operates under the control of (sstatus abbreviate).

prog
A prog is a LISP form based on the function prog which allows a control structure of sequential statements and gotos, rather than composed functions, to be used.

prog tag
An atom which tags or labels a particular statement in a prog so that it can be referred to with the go function.

prog variable
A variable which is bound by a prog; each prog contains a list of prog variables which are bound to nil when the prog is entered and can be used as temporary variables within the prog.

property
Associated with each atomic symbol are properties, which can be any LISP object. Each property is named by an "indicator," which is just an atomic symbol used to refer to that property. Thus we would refer to the "fsubr property" of cond, which has the atom fsubr as indicator

and is an internal pointer to the machine code for cond.

property list
: The list of indicators and properties kept on the cdr of each atomic symbol.

quote
: A special function which is used to prevent the evaluation of arguments to other functions. (quote a) evaluates to the atomic symbol a, while just a evaluates to the value of a. (quote a) is usually abbreviated 'a.

readtable
: A table which specifies the lexical significance of each ascii character. The readtable is used by the function read to direct the parsing of input. It can be altered by the user to implement special extensions to LISP syntax or to allow use of the read function to lexically analyze languages other than LISP. There can be more than one readtable; at any given time the one that is used is the one that is the value of the atom readtable.

recursion
: See recursion!

rplaca
: Changing the car of a previously-existing cons to something other than what it was originally created as. All references to that cons will find that its car has been changed on them. This operation has hidden dangers and should not be used lightly.

rplacd
: Changing the cdr of a previously-existing cons. Similar to rplaca.

S-expression
: Another name for "LISP object."

single character object
: An atomic symbol whose pname is a single character is a "single character object" if the syntax of that character has been set so that the character reads as a seperate atomic symbol even if it is not surrounded by spaces or other delimiters.

slashify
: "Slashifying" a character is preceding it with a slash (/) character.

This can be done to special characters such as space or parenthesis to indicate that they should be treated the same as alphabetic letters and their special meanings should be ignored. Slashification is the convention by which pnames may contain these special characters.

sorting
: MACLISP includes a generalized sorting facility. An array or a list of objects can be sorted if a function can be written to determine for any pair of such objects, which is the lesser. See chapter 11.

special array cell
: Some MACLISP implementations use "special array cells" as values of array properties. These cells are communication words which allow the array to be addressed by both compiled and interpreted code.

stack
: "stack" is synonymous with "pdl," q.v.

string
: One of the MACLISP data types is the string of characters, written "foo".

subr
: A subr is a machine language function which takes a fixed number of evaluated arguments. When an expr is compiled, it becomes a subr. A number of the builtin functions, such as memq, are subrs. Occasionally the term subr is used to include all machine executable functions, fsubrs and lsubrs as well as true subrs.

subr object
: The value of a subr, fsubr, or lsubr property. In some implementation dependent way, a subr object tells lisp how to get to the machine language function given its name (an atomic symbol with a subr, lsubr, or fsubr property.)

substitution
: One S-expression may be substituted for another within a third by using the functions subst and sublis. See chapter 4.

switch
: A "switch" is an atomic symbol whose value is by convention either t or nil, representing on and off respectively. There are a number of switches which affect the operation of the lisp system.

symbol
See atomic symbol.

syntax
See readtable.

t
An atomic symbol which is used for expressing truth. Like `nil`, it is a constant because its value is always itself.

tag
See prog tag, catch tag.

terminal
MACLISP is almost always used interactively by a user communicating with it through a terminal. The phrase "the terminal" or "the console" is used in this document to mean the particular terminal which is controlling the computation under discussion.

time
MACLISP keeps track of two types of time. "time" is elapsed time in seconds, since some arbitrary event such as the last time the computer system was started. "runtime" is the number of microseconds of CPU running time that has been used.

top level
The level of recursion which lisp is at when first entered. The user at his terminal is in control. Lisp will accept typed-in forms, evaluate them, and print the results.

trace
A package for debugging LISP programs which allows control to be seized whenever specified functions are called. Various operations to be performed, such as displaying of arguments, examination of specified variables, and temporarily returning control to the console via a breakpoint. See chapter 15.

truly worthless atom
An atomic symbol which is not referenced by any list structure, has no value, and has no properties. In most cases no one would notice if a truly worthless atom was removed from the environment and recreated when someone later referred to its pname. Therefore MACLISP provides the gctwa function which can be used to direct the garbage collector to remove truly worthless atoms, in the interests of saving memory.

type

See chapter 2 for a description of the builtin data types in MACLISP, and a list of predicates for type-checking. Numeric type-conversions can be done with the functions listed in section 7.1.3. Other type conversions can be done with a host of functions listed mostly in chapter 13. The user may efficiently define new data types simply by defining functions to manipulate them.

type checking

See section 2.1 for a list of predicates which return `t` if their argument is of a specified data type. In the interpreter most functions automatically check their arguments for correct type, but in compiled code types are usually assumed to be correct, and if they are not, the internal mechanisms which support MACLISP may be damaged.

unbound variable

A variable which has no value is called "unbound." Attempting to evaluate such a variable will cause an error.

user interrupts

The user interrupt facility allows a user-specified function to gain control when a specified condition occurs, no matter what else may be happening (except that response to a user interrupt may be delayed while garbage collection takes place.) User interrupts are used for error recovery, alarmclock timers, and real-time response to the entry of certain "attention getting" characters from the terminal. See section 12.4 for details.

uuo

For historical reasons, the term "uuo" is used to describe the direct linkage between compiled or builtin functions which is used to increase the efficiency of function calling. This linkage eliminates the necessity to search property lists each time a function is called when both the caller and the callee are machine language (compiled or builtin) functions. In the pdp-10 implementation of MACLISP, this linkage is accomplished by a mechanism which includes the use of UUO instructions, hence the term "uuo." Note that the function `(sstatus uuolinks)` can be used to break this linkage, for example so that tracing may be used. Setting the variable `nouuo` non-`nil` prevents the linkage from being established in the first place.

value cell

The value cell is that part of an atomic symbol in which its value is kept. In some implementations the value cell is kept on a "value" property, but in others it is kept in a "hidden" cell which is associated with the atomic symbol and is not accessible except to set or get the symbol's value.

Appendix B - Index of Functions

Function	Page	Type
*	65	LSUBR0 or more args
*$	67	LSUBR0 or more args
*array	80	LSUBR3 or more args
*function	18	FSUBR
*rearray	80	LSUBR1 or more args
*rset	109	SUBR1 arg
%include	182	FSUBR
+	65	LSUBR0 or more args
+$	67	LSUBR0 or more args
-	65	LSUBR0 or more args
-$	67	LSUBR0 or more args
/	65	LSUBR0 or more args
/$	67	LSUBR0 or more args
1+	66	SUBR1 arg
1+$	68	SUBR1 arg
1-	66	SUBR1 arg
1-$	68	SUBR1 arg
<	59	SUBR2 args
=	59	SUBR2 args
>	59	SUBR2 args
abs	61	SUBR1 arg
add1	63	SUBR1 arg
alarmclock	104	SUBR2 args
allfiles	154	SUBR1 arg
alloc	116	SUBR1 arg
alphalessp	52	SUBR2 args
and	36	FSUBR
append	30	LSUBR0 or more args
apply	17	LSUBR2 or 3 args
arg	20	SUBR1 arg
args	54	LSUBR1 or 2 args
array	80	FSUBR
arraydims	80	SUBR1 arg
ascii	73	SUBR1 arg
assoc	24	SUBR2 args
assq	25	SUBR2 args
atan	70	LSUBR1 or 2 args
atom	9	SUBR1 arg
baktrace	109	LSUBR0 to 2 args
baktrace1	110	LSUBR0 to 2 args
baktrace2	111	LSUBR0 to 2 args
bigp	57	SUBR1 arg
bltarray	81	SUBR2 args
boole	72	LSUBR3 or more args
boundp	48	SUBR1 arg
break	91	FSUBR
caaaar	23	SUBR1 arg
caaadr	23	SUBR1 arg
caaar	23	SUBR1 arg
caadar	23	SUBR1 arg

caaddr . . . . 23 SUBR1 arg
caadr . . . . 23 SUBR1 arg
caar . . . . 23 SUBR1 arg
cadaar . . . . 23 SUBR1 arg
cadadr . . . . 23 SUBR1 arg
cadar . . . . 23 SUBR1 arg
caddar . . . . 23 SUBR1 arg
cadddr . . . . 23 SUBR1 arg
caddr . . . . 23 SUBR1 arg
cadr . . . . 23 SUBR1 arg
car . . . . 23 SUBR1 arg
catch . . . . 43 FSUBR
catenate . . . . 76 LSUBR0 or more args
cdaaar . . . . 23 SUBR1 arg
cdaadr . . . . 23 SUBR1 arg
cdaar . . . . 23 SUBR1 arg
cdadar . . . . 23 SUBR1 arg
cdaddr . . . . 23 SUBR1 arg
cdadr . . . . 23 SUBR1 arg
cdar . . . . 23 SUBR1 arg
cddaar . . . . 23 SUBR1 arg
cddadr . . . . 23 SUBR1 arg
cddar . . . . 23 SUBR1 arg
cdddar . . . . 23 SUBR1 arg
cddddr . . . . 23 SUBR1 arg
cdddr . . . . 23 SUBR1 arg
cddr . . . . 23 SUBR1 arg
cdr . . . . 23 SUBR1 arg
charpos . . . . 166 LSUBR1 or 2 args
chrct . . . . 166 LSUBR1 or 2 args
clear-input . . . . 154 SUBR1 arg
cline . . . . 134 SUBR1 arg
close . . . . 143 SUBR1 arg
comment . . . . 19 FSUBR
cond . . . . 36 FSUBR
cons . . . . 29 SUBR2 args
cos . . . . 70 SUBR1 arg
crunit . . . . 156 FSUBR
CtoI . . . . 77 SUBR1 arg
cursorpos . . . . 152 LSUBR0 to 2 args
declare . . . . 182 FSUBR
defaultf . . . . 141 SUBR1 arg
definedp . . . . 48 SUBR1 arg
defprop . . . . 50 FSUBR
defsubr . . . . 188 LSUBR3 to 7 args
defun . . . . 54 FSUBR
delete . . . . 33 LSUBR2 or 3 args
deletef . . . . 153 SUBR1 arg
delq . . . . 33 LSUBR2 or 3 args
difference . . . . 63 LSUBR1 or more args
disaline . . . . 173 LSUBR3 to 5 args
disapoint . . . . 173 LSUBR2 or 3 args

disblink . . . . 172 SUBR2 args
dischange . . . . 173 SUBR3 args
discopy . . . . 172 SUBR1 arg
discreate . . . . 171 LSUBR0 or 2 args
discribe . . . . 172 SUBR1 arg
discuss . . . . 174 LSUBR4 or 5 args
diset . . . . 173 SUBR3 args
disflush . . . . 172 LSUBR0 or more args
disgobble . . . . 174 SUBR1 arg
disgorge . . . . 174 SUBR1 arg
disini . . . . 171 LSUBR0 or 1 args
dislink . . . . 173 SUBR3 args
dislist . . . . 173 LSUBR0 or 1 arg
dislocate . . . . 172 SUBR3 args
dismark . . . . 172 SUBR2 args
dismotion . . . . 174 SUBR4 args
display . . . . 172 SUBR2 args
do . . . . 39 FSUBR
edit . . . . 247 SUBRno args
endpagefn . . . . 167 LSUBR1 or 2 args
eoffn . . . . 149 LSUBR1 or 2 args
eq . . . . 23 SUBR2 args
equal . . . . 24 SUBR2 args
err . . . . 45 FSUBR
errframe . . . . 109 SUBR1 arg
error . . . . 44 LSUBR0 to 3 args
errprint . . . . 110 SUBR1 arg
errset . . . . 44 FSUBR
eval . . . . 17 LSUBR1 or 2 args
evalframe . . . . 110 SUBR1 arg
exp . . . . 69 SUBR1 arg
explode . . . . 74 SUBR1 arg
explodec . . . . 74 SUBR1 arg
exploden . . . . 74 SUBR1 arg
expt . . . . 64 SUBR2 args
fasload . . . . 188 FSUBR
filepos . . . . 154 LSUBR1 or 2 args
fillarray . . . . 81 SUBR2 args
fix . . . . 61 SUBR1 arg
fixp . . . . 9 SUBR1 arg
flatc . . . . 74 SUBR1 arg
flatsize . . . . 74 SUBR1 arg
float . . . . 61 SUBR1 arg
floatp . . . . 9 SUBR1 arg
force-output . . . . 154 SUBR1 arg
freturn . . . . 110 SUBR2 args
funcall . . . . 21 LSUBR1 or more args
function . . . . 18 FSUBR
gc . . . . 113 FSUBR
gcd . . . . 64 SUBR2 args
gctwa . . . . 113 FSUBR
gensym . . . . 53 LSUBR0 or 1 args

get . . . . 49 SUBR2 args
getchar . . . . 53 SUBR2 args
getl . . . . 50 SUBR2 args
get_pname . . . . 77 SUBR1 arg
go . . . . 41 FSUBR
greaterp . . . . 59 LSUBR2 or more args
grind . . . . 227 FSUBR
grind0 . . . . 227 FSUBR
grindef . . . . 227 FSUBR
haipart . . . . 61 SUBR2 args
haulong . . . . 58 SUBR1 arg
implode . . . . 73 SUBR1 arg
index . . . . 76 SUBR2 args
inpush . . . . 147 SUBR1 arg
intern . . . . 53 SUBR1 arg
ioc . . . . 97 FSUBR
iog . . . . 97 FSUBR
isqrt . . . . 69 SUBR1 arg
ItoC . . . . 77 SUBR1 arg
last . . . . 26 SUBR1 arg
length . . . . 26 SUBR1 arg
lessp . . . . 59 LSUBR2 or more args
linel . . . . 165 LSUBR1 or 2 args
linenum . . . . 166 LSUBR1 or 2 args
list . . . . 29 LSUBR0 or more args
listarray . . . . 81 SUBR1 arg
listen . . . . 153 SUBRno args
listify . . . . 20 SUBR1 arg
load . . . . 188 SUBR1 arg
log . . . . 69 SUBR1 arg
lsh . . . . 72 SUBR2 args
macdmp. . . . . 132 LSUBR0 or 1 args
make_atom . . . . 77 SUBR1 arg
maknam. . . . . 73 SUBR1 arg
makoblist . . . . 158 SUBR1 arg
makreadtable . . . . 163 SUBR1 arg
makunbound. . . . . 48 SUBR1 arg
map . . . . 85 LSUBR2 or more args
mapc . . . . 85 LSUBR2 or more args
mapcan . . . . 86 LSUBR2 or more args
mapcar . . . . 85 LSUBR2 or more arg
mapcon . . . . 86 LSUBR2 or more args
maplist . . . . 85 LSUBR2 or more args
max . . . . 60 LSUBR1 or more args
member. . . . . 26 SUBR2 args
memq. . . . . 27 SUBR2 args
mergef . . . . 151 LSUBR2 or more args
min . . . . 60 LSUBR1 or more args
minus . . . . 61 SUBR1 arg
minusp . . . . 57 SUBR1 arg
namelist . . . . 141 SUBR1 arg
namestring . . . . 141 SUBR1 arg

nconc . . . . 32 LSUBR0 or more args
ncons . . . . 29 SUBR1 arg
nointerrupt . . . . 105 SUBR1 arg
not . . . . 27 SUBR1 arg
nouuo . . . . 183 SUBR1 arg
nreverse . . . . 32 SUBR1 arg
null . . . . 27 SUBR1 arg
numberp . . . . 9 SUBR1 arg
oddp . . . . 57 SUBR1 arg
opena . . . . 143 SUBR1 arg
openi . . . . 142 SUBR1 arg
openo . . . . 142 SUBR1 arg
or . . . . 36 FSUBR
pagebporg . . . . 201 SUBRno args
pagel . . . . 166 LSUBR1 or 2 args
pagenum . . . . 166 LSUBR1 or 2 args
plus . . . . 63 LSUBR0 or more args
plusp . . . . 57 SUBR1 arg
prinl . . . . 146 LSUBR1 or 2 args
princ . . . . 147 LSUBR1 or 2 args
print . . . . 146 LSUBR1 or 2 args
prog . . . . 38 FSUBR
prog2 . . . . 19 LSUBR2 or more args
progn . . . . 20 LSUBR1 or more args
purcopy . . . . 183 SUBR1 arg
purify . . . . 201 SUBR3 args
putprop . . . . 50 SUBR3 args
quit . . . . 131 SUBRno args
quote . . . . 17 FSUBR
quotient . . . . 63 LSUBR1 or more args
random . . . . 71 LSUBR0 or 1 arg
read . . . . 145 LSUBR0 to 2 args
readch . . . . 145 LSUBR0 to 2 args
readline . . . . 145 LSUBR0 to 2 args
readlist . . . . 75 SUBR1 arg
remainder . . . . 64 SUBR2 args
remob . . . . 53 SUBR1 arg
remprop . . . . 51 SUBR2 args
remtrace . . . . 226 FSUBR
rename . . . . 153 SUBR2 args
return . . . . 42 SUBR1 arg
reverse . . . . 30 SUBR1 arg
rot . . . . 72 SUBR2 args
rplaca . . . . 32 SUBR2 args
rplacd . . . . 32 SUBR2 args
runtime . . . . 128 SUBRno args
samepnamep . . . . 52 SUBR2 args
sassoc . . . . 25 SUBR3 args
sassq . . . . 25 SUBR3 args
save . . . . 131 FSUBR
set . . . . 47 SUBR2 args
setarg . . . . 20 SUBR2 args

setq	47	FSUBR
setsyntax	162	SUBR3 args
shortnamestring	141	SUBR1 arg
signp	58	FSUBR
sin	70	SUBR1 arg
sleep	128	SUBR1 arg
sort	87	SUBR2 args
sortcar	88	SUBR2 args
sqrt	69	SUBR1 arg
sstatus	124	FSUBR
status	120	FSUBR
store	80	FSUBR
stringlength	76	SUBR1 arg
stringp	10	SUBR1 arg
subl	64	SUBR1 arg
sublis	31	SUBR2 args
subrp	10	SUBR1 arg
subst	30	SUBR3 args
substr	76	LSUBR2 or 3 args
sxhash	27	SUBR1 arg
sysp	127	SUBR1 arg
terpri	147	LSUBR0 or 1 args
throw	43	FSUBR
time	128	SUBRno args
times	63	LSUBR0 or more args
trace	223	FSUBR
tyi	145	LSUBR0 to 2 args
tyipeek	146	LSUBR0 or 1 arg
tyo	147	LSUBR1 or 2 args
typep	9	SUBR1 arg
ufile	156	FSUBR
ukill	156	FSUBR
untrace	226	FSUBR
uread	156	FSUBR
uwrite	156	FSUBR
valret	134	LSUBR0 or 1 args
xcons	29	SUBR2 args
zerop	57	SUBR1 arg
\	66	SUBR2 args

Appendix C - Index of Atomic Symbols

* 90
*nopoint 164
*pure 205
*rset 126
*rset-trap 106
$$$ 247
$p 91
? 102
alarmclock 104
arrayindex 102
autoload 107
base 164
bignum 9
bpend 204
bporg 204
edit 246
errlist 89
errset 105
fail-act 105
fixnum 9
flonum 9
funarg 19
gc-daemon 105
gc-lossage 105
ibase 168
infile 144
instack 144
list 9
nouuo 183
obarray 158
outfile 102
outfiles 144
prinlength 167
prinlevel 167
pure 204
random 9
read-eof 102
readtable 162
runtime 104
s 121
stream 140
string 9
symbol 9
time 104
tty 152
unbnd-vrbl 106
undf-fnctn 106
unseen-go-tag 106
uread 156
uwrite 157
value 122
wrng-no-args 106
wrng-type-arg 106
zunderflow 71
^a 97
^d 114
^q 144
^r 144
^w 144

Appendix D - Concept Index

a-list pointer 16
abbreviation 166
alarmclock 104
application 13
argument 13
arithmetic 57
array 6
association list 24
atom 5
atomic symbol 6
autoload 107
back trace 109
bignum 5
binding 11
boolean operations 72
break loop 91
breakpoint 91
car 7
cdr 7
character manipulation . . 73
character object 6
charpos 166
chrct 166
closing a file 143
comment 19
compilation 177
cons 7
control characters 92
correctable errors 98
debugging 108
declarations 184
defining functions 54
display slave 170
dot 7
dotted pair 7
editing 245
end of file 148
eoffn 148
eq versus equal 23
errors 35
evaluation 12
expr 13
fexpr 13
file 137
file name defaults 150
file-object 137
fixnum 5
flonum 5
flow of control 35
form 12
fsubr 13
funarg 13
funarg problem 18
function 13
functional property . . . 13
garbage collection 113
gc 113
gensym 53
grinding 227
I/O 135
indicator 49
input source 143
iteration 35
label 13
lambda 13
lambda variable 13
lap 194
lexpr 13
linel 165
list 7
loading 188
looping 35
lsubr 13
macro 13
macro character 169
mapping 83
mathematical functions . . 69
moby I/O 170
namelist 138
namestring 138
nil 6
non-local exit 35
number 5
obarray 158
object 5
opening a file 142
output destinations . . . 143
pdl pointer 108
pname 52
predicate 9
pretty-printing 227
property 49
property list 49
quote 17
readtable 159
recursion 35
S-expression 5
saving 129
sorting 87
special array cell 7

splicing macro 169
sstatus functions 124
status functions 120
storage spaces 114
string 6
subr 13
subr-object 6
substitution 30
symbol 6
t 6
time 128
top level 89
tracing 223
truly worthless atom . . . 113
user interrupts 100
uuo link 180
value cell 47