Part 1 - The Language

Table of Contents

1. General Information

1.1 The Maclisp Language

Maclisp is a dialect of Lisp developed at M.I.T.'s Project MAC and M.I.T.'s Artificial Intelligence Laboratory 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 augmented.

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. There are reputed to be several other implementations.

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.

The Maclisp system is structured as an *environment*, which is essentially a set of *names* and bindings of those names to data structures and function definitions. The environment contains a large number of useful *functions*. These functions can be used through an *interpreter* to define other functions, to control the environment, to do useful work, etc.

The interpreter is the basic user interface to the system. This is how the user enters "commands." When Maclisp is not doing anything else, such as running a program, it waits for the user to enter a Lisp *form*. This form is evaluated and the value is printed out. The form may call upon one of the system functions (or a user-defined function, of course) to perform some useful task. The evaluation of a form may initiate the execution of a large and complex program, perhaps never returning to the "top level" interpreter, or it may perform some simple action and immediately wait for the user to type another form.

It is also possible to get into the interpreter while a program is running, using the *break* facility. This is primarily used in debugging and related programming activities.

The functions invoked by the top-level interpreter may be executable machine programs, or they may themselves be interpreted. This is entirely a matter of choice and convenience. The system functions are mostly machine programs. User functions are usually first used interpretively. After they work, the *compiler* may be applied to them, turning them into machine programs which can then be *loaded* into the environment.

All of this is done within a single consistent language, Lisp, whose virtue is that the data structure is simple and general enough that programs may easily operate on programs, and that the program structure is simple and general enough that it can be used as a command language.

1.2 Structure of the Manual

The manual is generally structured into sections on particular topics; each section contains explanatory text and function definitions, interspersed. In general, each section contains both elementary and complex material, with complexity increasing toward the end of the section. An axiomatic, step-by-step development is not used. Frequently the more complex information in a section will assume knowledge from other sections which appear later in the manual. The new user is advised to skip around, reading early chapters and early sections of chapters first.

Often descriptions of Lisp functions will be given not only in prose but also in terms of other Lisp functions. These are as accurate as possible, but should not be taken too literally. Their main purpose is to serve as a source of examples.

Accessing information in the manual is dependent 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 topics one can use the Table of Contents that is found at the beginning of the manual, and the more detailed tables of contents found at the beginning of each of the six major parts, to find where information of a general class will be found. Entry into the manual is also facilitated by the Glossary and the Concept Index, which are found at the end. Also at the end of the manual are a Function Index and an Atomic Symbol Index which are 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. When one section of the manual assumes knowledge of another section a page number reference to the other section will generally be given.

1.3 Notational Conventions

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

Most numbers are in octal radix (base eight). Numbers with a decimal point and spelled-out numbers are in decimal radix. It is important to remember that by default Maclisp inputs and outputs all numbers in *octal* radix. If you want to change this, see the variables `base` and `ibase`.

A combination of the characters equal sign and greater than symbol, "=>", will be used in examples of Lisp code to mean evaluation. For instance, "*F* => *V*" means that evaluating the form *F* produces the value *V*.

All uses of the phrase "Lisp reader," unless further qualified, refer to that part of the Lisp system which reads input, and not to the person reading this document.

The terms "S-expression" and "Lisp object" are synonyms for "any piece of Lisp data."

The character "$" always stands for dollar-sign, never for "alt mode," unless that is specifically stated.

The two characters accent acute, "′", and semi-colon, ";", are examples of what are called *macro characters*. Though the macro character facility, which is explained in Part 5, is not of immediate interest to a new user of the dialect, these two macro characters come preset by the Lisp system and are useful. When the Lisp reader encounters an accent acute, or *quote mark*, it reads in the next S-expression and encloses it in a quote-form, which prevents evaluation of the S-expression. 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 character. When the Lisp reader encounters it, the remainder of the line is discarded.

The term "newline" is used to refer to that character or sequence of characters which indicates the end of a line. This is implementation dependent. In Multics Maclisp, newline is the Multics newline character, octal 012. In ITS Maclisp, newline is carriage return (octal 015), optionally followed by line feed (octal 012.) In dec-10 Maclisp, newline is carriage return followed by line feed.

All Lisp examples in this manual are written according to the case 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 use only upper case letters because they exist on systems which are not, or have not always been, equipped with terminals capable of generating and displaying the full ascii character set. However, these implementations will accept input in lower case and translate it to upper case, unless the user has explicitly said not to.

2. Data Objects

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 manifest; that is, it is possible for a program to tell what type an object is just by looking at the object itself, so it is not necessary to declare the types of variables as in some other languages. One can make declarations, however, in order to aid the compiler in producing optimal code. (See part 4.2.)

It is important to know that Lisp represents objects as pointers, so that a storage cell (a "variable") will hold any object, and the same object may be held by several different storage cells. 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 (car and cdr), while non-atomic objects are structures constructed out of other objects. Composite objects are indivisible, atomic, entities which have other objects associated with them. These other objects may be examined and replaced.

The atomic data types are numbers, atomic symbols, strings, and subr-objects. Atomic symbols can also be regarded as composite. See below.

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 machine-dependent. A flonum is a floating-point number whose precision and range of values are machine-dependent. A bignum is an infinite-precision integer. It is impossible to get "overflow" in bignum arithmetic, as any integer can be represented by a bignum. However, fixnum and flonum arithmetic is faster than bignum arithmetic and requires less memory. Sometimes the word "fixnum" is used to include both fixnums and bignums (i.e. all integers); in this manual, however, the word "fixnum" will never be used to include bignums unless that is explicitly stated.

The printed 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 15656565656565656565656565656565. Some examples of flonums are: 4.0, .01, -6e5, 4.2e-1.

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 associated with it a name, a value, and possibly a list of "properties". The name is a sequence of characters, which is the printed representation of the atomic symbol. This name is often called the "pname," or "print-name." A pname may contain any ascii character except the null character, which causes trouble in some implementations. For example, a certain atomic symbol 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 their respective selves as values and their values may not be changed. nil is used as a "marker" in many contexts; it is essential to the construction of data structures such as lists. t is usually used when an antithesis to nil is required for some purpose, e.g. to represent the logical conditions "true" and "false." Another property of the special atomic symbol nil is that its car and its cdr are always nil.

The value of an atomic symbol can be 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 is explained on page 2-52. It is used for such things as recording the fact that an atomic symbol is the name of a function.

An atomic symbol with one or no 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 one-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 page 2-58).

Another Lisp data type is the string. This is a sequence of characters (possibly zero-length). 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 printed 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`. In implementations without strings, atomic symbols are used instead. The pdp-10 implementations currently lack strings.

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 has no printed representation, 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; each cell is named by a tuple of integers. An array may have one or more dimensions; the upper limit on the number of dimensions is implementation-defined. An array is *not* always associated with an atomic symbol which is its name. Rather, an array is always designated by an array-pointer, which is a special kind of atomic Lisp object. Frequently, an array-pointer will be placed on the property list of a symbol under the indicator `array` and then that symbol will be used as the name of the array, since symbols can have mnemonic names and a reasonable printed representation. See page 2-89 for an explanation of how to create, use, and delete arrays.

Another composite data type is the file-object, which is described on part 5.3.

The sole 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 are names of fields in an IBM 7094 machine word.) 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:

```
 ___________________________
|             |             |
|     car     |     cdr     |
|_____________|_____________|
```

where the boxes represent cells of memory large enough to hold a pointer, and "car" and "cdr" are two pointers to objects. The printed 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:

```
--> o -----> cdr
    |
    |
    V
   car
```

There are three Lisp functions associated with conses: **cons**, **car**, and **cdr**. The function cons combines its two arguments into a cons; `(1 . 2)` can be generated by evaluating `(cons 1 2)`. The function car returns the car component of its argument, and the function cdr returns the cdr component 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 three things 1, 2, and 3 is constructed by `(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:

```
--> o ----> o ----> o ----> nil
    |       |       |
    |       |       |
    V       V       V
    1       2       3
```

From this it can be seen that the car of a list is its first element, that the cdr of a list is a list of the elements after 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 printed 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 . (C . 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 identified with the atom `nil`. It may be typed in as either `nil` or `()`.

3. The Basic Actions of LISP

3.1 Binding of Variables

The basic primitives of programming in Lisp are variables, forms, and functions. A variable is an atomic symbol which has a value associated with it; the symbol is said to be *bound* to that value. The value may of course be any Lisp object whatsoever. The atomic symbol acts simply as a name by which the program may refer to the value while it is processing it.

This is similar to the concept of variables in other programming languages. However, Lisp's concept of the scope of names is subtly different from that of most "block-structured" languages. At a given moment, a variable may actually have several bindings in existence. Only the most recent, or current binding, can be used. When a new binding is created, the previous one is pushed onto a stack. It will become accessible again when the binding which superseded it is removed. Creation and removal of bindings is synchronized with subroutine calling (and with certain special forms described below) so this mechanism corresponds closely to the "local variables" concept of other programming languages. However, Lisp considers that there is only one variable whose binding changes, rather than several separate variables which happen to have the same name. Any reference to a variable, even from outside the particular program which gave it its current binding, gets the *current* binding and not one determined by "scope rules." It is possible to simulate the other concept of scope of names by using *binding context pointers*, which are described later (see page 1-22).

Unlike many other languages, Lisp does not combine the concepts of name and storage. Many languages associate with a variable (a name) a piece of storage which can hold one object of a particular type, such as a floating point number. The variable's value resides in this storage. It is then impossible for two variables to really have "the same" value; one could have a copy of the value of another but not the same identical object.

The situation in Lisp is quite different. Binding a variable to a value is not copying the value into storage associated with that variable. Values exist as separate objects in their own right and in their own storage. Binding is simply an association between a variable and a value; consequently there is no reason why two variables cannot have truly identical values. Similarly, erasing the binding between a variable and its value does not destroy or throw away the value; it simply breaks the association. Of course, if there is no other use for the value the storage it occupies will eventually be reclaimed by the system and put to more productive use.

Often these processes of creating a new binding of a variable to a value and reverting to a previous binding are referred to as *binding* and *unbinding* the variable, respectively.

A slightly different way of creating a binding between a variable and a value is *assignment*. When a variable is *bound* to a value, the previous binding is saved and can be restored, but when a variable has a value *assigned* to it, the previous binding is not saved, but is simply replaced. Thus binding may be regarded as creating a new *level* of usage of a variable, while assignment switches a variable to a different value within the same level. For instance, a subroutine or function may bind a variable to an initial value when it is entered, and then proceed to make use of that variable, possibly assigning a different value to it from time to time. The initial binding of the variable establishes the (temporary) ownership of that variable by the subroutine.

Due to the subtlety of the distinction between binding and assignment, some people have proposed that assignment be eliminated wherever possible. The Maclisp do function can often be useful in this regard.

There are several program constructs by which a variable can be bound. These will be explained after forms and functions have been introduced.

3.2 Evaluation of Forms

The process of "executing" a Lisp program consists of the *evaluation* of forms. Evaluation takes a form and produces from it a value (any Lisp object), according to a strict set of rules which might be regarded as the complete semantics of Lisp.

If the form is atomic, it is evaluated in a way which depends on its data type. An atomic symbol is a variable; it evaluates to the value to which it is currently bound. If it is not bound, an error occurs. (See part 3.4.) A number or a string is a literal constant; it evaluates to itself. The special atomic symbols `t` and `nil` are also treated as constants. A constant can also be created by use of the quote special form; the value of `(quote` *x*`)` is *x*.

If the form is a list, its first element specifies the operation to be performed, and its remaining elements specify arguments to that operation. Non-atomic forms come in two types: *special forms*, which include the necessary programming operations such as assignment and conditionals, and *function references*, in which the "operation" is a function which is applied to the specified arguments. Thus functional composition is the method by which programs are built up out of parts - as distinguished from composition of data structures, for example. Lisp functions correspond closely to subroutines in other programming languages.

A function may be either a primitive which is directly executable by the machine, called a *subr* (short for "subroutine"), or a function defined by composition of functions and special forms, called an *expr* (short for "expression.") Most subrs are built in to the language, but it is possible for a user to convert his exprs into subrs by using the compiler (see part 4.) This gains speed and compactness at some cost in debugging features.

There is additional complexity because special forms are actually implemented as if they were function references. There is a special type of subr called an *fsubr* which is used for this purpose. An fsubr is permitted to make any arbitrary interpretation of its argument specification list, instead of following the standard procedure which is described below. It is also possible to define a special form by an expr, which is then called a *fexpr*. Most of the built-in special forms are handled specially by the compiler. They are compiled as the appropriate code rather than as a call to the fsubr.

Other types of functions are *lsubr*, which is just a subr with a variable number of arguments, *lexpr*, which is an expr with a variable number of arguments, and *macro*, which is a type of special form whose result is not a value, but another form; this allows a "transformational" type of semantics.

Consider the form

(*F A1 A2 ... An*)

The evaluator first examines *F* to see if it is a function which defines a special form, i.e. an fsubr, a fexpr, or a macro. If so, *F* is consulted and it decides how to produce a value. If not, *F* must be an ordinary function. The sub-forms *A1* through *An* are evaluated, producing *n* arguments, and then the definition of *F* is *applied* to the arguments. (Application is described in the following section.) This yields a result (some Lisp object), which is then taken as the value of the form.

An atomic form of some random type, such as a subr-object, a file, or an array-pointer, evaluates to something random, often itself; or else causes an error depending on the convenience of the implementation. Note that an array-pointer is different from an atomic symbol which happens to be the name of an array; such an atomic symbol is evaluated the same as any other atomic symbol.

3.3 Application of Functions

When a non-atomic form is evaluated, the function specified by that form is combined with the arguments specified by that form to produce a value. This process is called *application*; the function is said to be *applied to* the arguments.

The first step in application is to convert the function-specifier into a *functional expression* (sometimes confusingly called a functional form.) A functional expression is a Lisp object which is stylized so that Lisp can know how to apply it to arguments. The rules for this conversion will be described after the types of functional expressions have been explained.

There are basically two types of functional expression. A lambda-expression is a functional expression which specifies some variables which are to be bound to the arguments, and some forms which are to be evaluated. One would expect the forms to depend on the variables. The value of the last form is used as the value of the application of the lambda-expression. Any preceding forms are present purely for their side-effects. A lambda-expression looks like:

```
(lambda (a b c d)
     form1
     form2
     form3)
```

Here a, b, c, and d are the variables to be bound to the values of the arguments, called the lambda-variables. If at a certain moment the current binding of a was the one created by this lambda-expression, a would be said to be lambda-bound. Clearly this lambda-expression is a function which accepts four arguments. The application of the functional expression to four arguments produces a value by evaluating form1, then form2, and then form3. The value of form3 is the value of the whole form. For example, the value of the form

```
((lambda (a b) b) 3 4)
```

is 4. The functional expression used is a very simple one which accepts two arguments and returns the second one.

If we grant the existence of a primitive addition operation, whose functional expression may be designated by +, then the value of the form

```
((lambda (a b) (+ a b)) 3 4)
```

is 7. Actually,

```
(+ 3 4)
```

evaluates to the same thing.

The second basic type of functional expression is the *subr*, which is a program directly executable by the machine. The arguments of the form are conveyed to this program in a machine-dependent manner, it performs some arbitrary computation, and it returns a result. The built in primitives of the language are subrs, and the user may write lambda-expressions which make use of these subrs to define his own functions. The *compiler* may be used to convert user functions into subrs if extra efficiency is required.

It is extremely convenient to be able to assign names to functional expressions. Otherwise the definition of a function would have to be written out in full each time it was used, which would be impossibly cumbersome.

Lisp uses atomic symbols to name functions. The "property list" mechanism is used to associate an atomic symbol with a functional expression. (See page 2-52 for an explanation of property lists.) Because the binding mechanism is not used, it is possible for the same name to be used for both a variable and a function with no conflict. Usually the defun special form is used to establish the association between a function name and a functional expression.

Thus, the car of a form may be either a functional expression itself, or an atomic symbol which names a functional expression. In the latter case, the name of the "property" which associates the symbol with the expression gives additional information:

A lambda-expression is normally placed under the expr property. This defines an ordinary expr.

If a lambda-expression is placed under the fexpr property, it defines a special form. In that case, the first lambda-variable is bound to the cdr of the form being evaluated. For example, if foo is a fexpr, and (foo (a b) (c d)) is evaluated, then foo's lambda-variable would be bound to ((a b) (c d)). A second lambda-variable may optionally be included in a fexpr. It will be bound to a "binding context pointer" to the context of the evaluation of the form. (See page 1-22 for the details of binding context pointers.)

If a lambda-expression with one lambda-variable is placed under the macro property, it defines the "macro" special form mentioned above. The lambda-expression is applied to the entire form, as a single argument, and the value is a new form that is evaluated in place of the original form.

If a subr-object is placed under the subr property, it defines a subr. If a subr-object is placed under the fsubr property, it defines a special form. A subr-object under the lsubr property defines a subr which accepts varying numbers of arguments.

There are some additional refinements. A lambda-expression which accepts varying numbers of arguments, called a lexpr, looks as follows:

```
(lambda n
    form1
    form2)
```

The single, unparenthesized, lambda-variable n is bound to the number of arguments. The function arg, described on page 2-12, may be used to obtain the arguments.

Another property which resembles a functional property is the autoload property. If Lisp encounters an autoload property while searching the property list of a symbol for functional properties, it loads in the file of compiled functions specified by the property, then searches the property list again. Presumably the file would contain a definition for the function being applied, and that definition would be found the second time through. In this way packages of functions which are not always used can be present in the environment only when needed.

An array may also be used as a function. The arguments are the subscripts and the value is the contents of the selected cell of the array. An atomic symbol with an array property appearing in the function position in a form causes that array to be used.

If the function-specifier of a form doesn't meet any of the above tests, Lisp evaluates it and tries again. In this way, "functional variables" and "computed functions" can be used. However, it is better to use the funcall function. (See page 2-13.)

There are some other cases of lesser importance:

There is an obscure type of functional expression called a label-expression. It looks like

(label *name* (lambda (...) ...))

The atomic symbol *name* is bound to the enclosed lambda-expression for the duration of the application of the label-expression. Thus if *name* is used as a functional variable this temporary definition will be used. This is mostly of historical interest and is rarely used in actual programming.

Another type of functional expression is the funarg. A funarg is a list beginning with the atomic symbol funarg, as you might expect, and containing a function and a binding context pointer. Applying a funarg causes the contained function to be applied in the contained binding context instead of the usual context. funargs are created by the `*function` special form.

An expr property may be an atomic symbol rather than a `lambda`-expression. In this case, the atomic symbol is used as the function. The original symbol is simply a synonym for it.

In addition to the variety of application just 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` separately. They are not encoded into a form, consequently macros are not accepted by this version of application. Note that what is passed to `apply` is a list of arguments, *not* a list of expressions which, evaluated, would yield arguments.

3.4 Special Forms

This section briefly describes some of the special forms in Maclisp. For full details on a specific special form, consult the Function Index in the back.

Constants

> `(quote x)` evaluates to the S-expression *x*.
>
> `(function x)` evaluates to the functional expression *x*. There is little real difference between quote and `function`. The latter is simply a mnemonic reminder to anyone who reads the program - including the compiler - that the specified expression is supposed to be some kind of function.

Conditionals

> Conditionals control whether or not certain forms are evaluated, depending on the results of evaluating other forms. Thus both the value and the side effects of the conditional form can be controlled.
>
> `(cond (`*predicate form1 form2...*`) (`*predicate form1 form2...*`)...)`
>
> is a general conditional form. The lists of a predicate and some forms are called *clauses*. The cond is evaluated by considering the clauses one by one in the order they are written. The predicate of a clause is evaluated, and if the result is *true*, that is, anything other than `nil`, then the forms in that clause are evaluated and the cond is finished without examining the remaining clauses. If the result is not *true*, i.e. if it is `nil`, then the next clause is examined in the same way. If all the clauses are exhausted, that is *not* an error. The value of a cond is the value of the last form it evaluates, which could be `nil` if no predicate is true, or the value of a predicate if that predicate is true but has no forms in its clause.
>
> `(and` *form1 form2 form3...*`)` evaluates the *forms* in succession until one is `nil` or the *forms* are exhausted, and the result is the value of the last *form* evaluated.
>
> `(or` *form1 form2 form3...*`)` evaluates the *forms* until one is non-`nil` or the *forms* are exhausted, and the result is the value of the last *form* evaluated.

Non-Local Exits

(catch *form tag*) evaluates the *form*, but if the special form (**throw** *value tag*) is encountered, and the *tag*s are the same, the **catch** immediately returns the *value* without further ado. See page 2-44 for the full details.

Iteration

(prog (*variable...*) *form-or-tag* ...) allows Fortranoid "programs" with *goto*'s, local variables, and *return*'s to be written.

(do ...) is the special form for iteration. See page 2-38 for the details of **prog** and do.

Defining Functions

(defun *name* (*arg1 arg2...*) *form1 form2...*) defines an (interpreted) function. See page 2-60 for full details.

Error Control

(break *name* t) causes ";bkpt *name*" to be typed out and gives control to a read-eval-print loop so that the user can examine and change the state of the world. When he is satisfied, the user can cause the break to return a value. See page 3-5 for the details of break.

(errset *form*) evaluates the *form*, but if an error occurs the **errset** simply returns nil. If no error occurs, the value is a list whose single element is what the value of the *form* would have been without **errset**.

Assignment

(setq *var1 value1 var2 value2...*) assigns the values to the variables. The values are forms which are evaluated.

(store (*array subscript1 subscript2...*) *value*) assigns the value to the array cell selected by subscripting. See part 2.8 for further information on arrays.

Miscellaneous Parameters

(status *name -optional args-*) returns miscellaneous parameters of LISP. *name* is a mnemonic name for what is to be done.

(`sstatus` *name -optional args-*) sets miscellaneous parameters.

See part 3.7 for the details of `status` and `sstatus`.

Pretty-Printing

(`grindef` *x*) prettily prints the value and function definition (if any) of the atomic symbol *x*. Indentation is used to reveal structure, the `quote` special form is represented by ', etc. See part 6.3 for the details..

Tracing

(`trace` *name*) causes the function *name* to print a message whenever it is called and whenever it returns. See page 3-35 for the many features and options of `trace`.

3.5 Binding Context Pointers

There is a special type of object called a *binding context pointer*, or sometimes an "*a-list pointer*", which can be used to refer to a binding context (a set of bindings of variables and values which was extant at a particular instant.) Due to the stack implementation of Maclisp, a binding context 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 retaining a pointer to it.

A binding context 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 four following sources: the function `evalframe`, the function `errframe`, the special form `*function`, or the second `lambda`-variable of a `fexpr`.

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

Part 2 - Function Descriptions

Table of Contents

1. Predicates

A predicate is a function which tests for some condition involving its argument and returns t if that condition is true, or nil if it is not true.

The following predicates are for checking data types. These predicates return t if their argument is of the type indicated 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 returns nil if its argument is a dotted-pair or a list, or t if it is any kind of atomic object such as a number, a character string, or an atomic symbol.

symbolp SUBR 1 arg

The symbolp predicate returns t if its argument is an atomic symbol, or nil if it is anything else.

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.

bigp SUBR 1 arg

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

numberp   SUBR 1 arg

The numberp predicate returns t if its argument is any kind of number, nil if it is not.

hunkp   SUBR 1 arg

The hunkp predicate returns t if its argument is a *hunk* (see page 2-33 for a discussion of hunks). hunkp does not consider list cells to be hunks. This predicate does not exist in the Multics implementation.

typep   SUBR 1 arg

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

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

symbol means atomic symbol. list means a list or a cons. array means array-pointer. 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))
```

The following 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
```

The following are a more miscellaneous set of predicates.

`eq` SUBR 2 args

(eq *x* *y*) => 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 multiple 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)
(eq (cons 'a 'b) (cons 'a 'b)) => nil (always)
(setq x '(a . b)) (eq x x) => t since it is
      the same copy of (a . b) in both arguments.
(setq x (setq y 17)) (eq x y) => t or nil
       depending on the implementation. You can
       never rely on numbers being eq.
```

`equal` SUBR 2 args

The equal predicate returns t if its arguments are similar (isomorphic) 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 characters composing them 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 (numberp x) (numberp y) (numequal x y))
                (and (not (atom x))
                     (not (atom y))
                     (equal (car x) (car y))
                     (equal (cdr x) (cdr y)))))
```

if there was an auxiliary function for numeric equality:

```
        (defun numequal (x y)
               (and (eq (typep x) (typep y))
                    (zerop (difference x y))))
```

This numequal function is not the same as the Maclisp numeric-equality function, =, because the latter only compares non-big numbers.

As a consequence of the above definition, it can 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.

not SUBR 1 arg

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

null SUBR 1 arg

This is the same as not. Both functions are provided for the sake of clarity. null should be used to check if something is nil and return a logical value. not should be used to invert the sense of a logical value. Even though Lisp uses nil to represent logical "false," you shouldn't make understanding your program depend on this. For example, one often writes

```
    (cond ((not (null x)) ... )
          ( ... ))
```

rather than

```
(cond (x ... )
      ( ... ))
```

There is no loss of efficiency since these will compile into exactly the same instructions.

See also the number predicates (page 2-63).

2. The Evaluator

`eval` LSUBR 1 or 2 args

`(eval` *x*`)` evaluates *x*, as a form, atomic or otherwise, and returns the result.

`(eval` *x p*`)` evaluates *x* in the context specified by the binding context pointer *p*. Example:

```
(setq x 43 foo 'bar)
(eval (list 'cons x 'foo))
     => (43 . bar)
```

`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 two arguments except that the application is done with the variable bindings specified by the binding context pointer *p*.

`quote` FSUBR

The special form `(quote` *x*`)` returns *x* without trying to evaluate it. **quote** is used to include constants in a form. For convenience, the **read** function normally converts any S-expression preceded by the apostrophe or acute accent character (') into the quote special form. For example,

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

is converted by the reader to:

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

which when evaluated causes the variable x to be set to the constant list value shown. For more information on input syntax, see the detailed discussion in part 5.1.

quote could have been defined by:

```
     (defun quote fexpr (x) (car x))
```

function   FSUBR

function is like quote except that its argument is a functional expression. To the interpreter, quote and function are identical, but the compiler needs to be able to distinguish between a random piece of data, which should be left alone, and a function, which should be compiled into machine code. Example:

```
   (mapcar (function (lambda (p q)
                       (cond ((eq p '*) q)
                             (t (list p '= q)) )))
           first-list-of-things
           (compute-another-list) )
```

calls mapcar with three arguments, the first of which is the function defined by the lambda-expression. The actual value passed to mapcar depends on whether the form has been compiled. If it is interpreted, the lambda-expression written above will be passed. If it is compiled, an automatically-generated atomic symbol with the compiled code for the lambda-expression as its subr property will be passed. The usual thing to do with functional arguments is to invoke them via apply or funcall, which accept both the compiled and the interpreted functional forms.

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 a binding context 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) ))
```

or, better

```
(defun foo (f)
    (append one-value (funcall 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. Funargs generated by `*function` are intended for use as functional *arguments* and cannot be returned as values of functional applications. Thus, the user should be careful in his use of `*function` to make sure that his use does not exceed the limitations of the Maclisp funarg mechanism.

It is possible to assign a value to a variable when a previous binding of that variable has been made current by a funarg. The assignment will be executed in the proper context. (This has not always been the case in Maclisp; it is a fairly new feature.)

A funarg has the form

`(funarg` *function . context-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 lisp reader.

Example:

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

A problem with such comments is that they are discarded when the S-expression is read into lisp. If it is edited within lisp and printed back into a file, the comments will be lost. However, most users edit the original file and read the changes into lisp, since this allows them to use the editor of their choice. Thus this is not a real problem.

prog2 LSUBR 2 or more args

The expressions in a prog2 form are evaluated from left to right, as in any lsubr-form. The result is the second argument. prog2 is most commonly used to evaluate an expression with side effects, then return a value which needs to be computed before the side effects happen.
Examples:

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

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

(defun prog2 nargs (arg 2)) ;a lexpr definition for prog2
```

progn LSUBR 1 or more args

The expressions in a progn form are evaluated from left to right, as usual, and the result is the value of the last one. In other words, progn is an lsubr which does nothing but return its last argument. Although lambda-expressions, prog-forms, do-forms, cond-forms, and iog-forms all use progn implicitly, that is, they allow multiple forms in their bodies, there are occasions when one needs to evaluate a number of forms for side-effects and make them appear to be a single form. progn serves this purpose. Example:

```
(progn (setq a (cdr frob)) (eq (car a) (cadr a)))
```

might be used as the antecedent of a cond clause.

progn could have been defined by:

```
(defun progn nargs
    (and (> nargs 0)
         (arg nargs)))
```

progv FSUBR

progv is a special form to provide the user with extra control over lambda-binding. It binds a list of variables to a list of values, and then evaluates some forms. The lists of variables and values are computed quantities; this is what makes progv different from lambda, prog, and do.

(progv *var-list value-list form1 form2* ...)

first evaluates *var-list* and *value-list*. Then the variables are bound to the values. In compiled code the variables must be *special*, since the compiler has no way of knowing what symbols might appear in the *var-list*. If too few values are supplied, the remaining variables are bound to nil. If too many values are supplied, the excess values are ignored.

After the variables have been bound to the values, the *forms* are evaluated, and finally the variable bindings are undone. The result returned is the value of the last form. Note that the "body" of a progv is similar to that of progn, not that of prog.
Example:

```
(setq a 'foo b 'bar)

(progv (list a b 'b) (list b) (list a b foo bar))
    => (foo nil bar nil)
```

During the evaluation of the body of this progv, **foo** is bound to **bar**, **bar** is bound to nil, b is bound to nil, and a remains bound to foo.

`arg` SUBR 1 arg

(arg nil), when evaluated during the application of a lexpr, gives the number of arguments supplied to that lexpr. This is primarily a debugging aid, since lexprs also receive their number of arguments as the value of their lambda-variable.

(arg *i*), when evaluated during the application of 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 2))           ;print the second argument.
    (+ (arg 1)                ;return the sum of the first
       (arg (- nargs 1))))    ;and next to last arguments.
```

`setarg` SUBR 2 args

setarg is used only during the application of 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` *n*`)` 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 argument *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)
            (listifyl (arg nil) (+ (arg nil) n 1)))
           (t
            (listifyl n 1)) ))

(defun listifyl (n m)       ; auxiliary function.
     (do ((i n (1- i))
          (result nil (cons (arg i) result)))
         ((< i m) result) ))
```

`funcall` LSUBR 1 or more args

`(funcall` *f a1 a2 ... an*`)` calls the function *f* with the arguments *a1, a2, ..., an*. It is similar to `apply` except that the separate arguments are given to `funcall`, rather than a list of arguments. 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
```

`subrcall` FSUBR

`subrcall` is used to invoke a subr-pointer directly, rather than by referring to an atomic symbol of which the subr-pointer is the subr property. The form is:

`(subrcall` *type p a1 a2 ... an*`)`

All arguments except the first are evaluated. *type* is the type of result expected: `fixnum`, `flonum`, or `nil` (any type). *p* is the subr pointer to be called. *a1* through *an* are the arguments to be passed to the subr. `subrcall` compiles into efficient machine code.

lsubrcall FSUBR

lsubrcall is identical to subrcall except that the subr-pointer called has to be an lsubr instead of a subr. This is because many Lisps use different internal calling sequences for lsubrs than for subrs.

arraycall FSUBR

arraycall is similar to subrcall and lsubrcall except that an array-pointer is used instead of a subr-pointer. The first argument of **arraycall** must correspond to the type that the array was given when it was created. An arraycall expression may be used as the first argument to **store**.

3. Manipulating List Structure

3.1 Conses

car SUBR 1 arg

Returns the first component of a cons.

Example: `(car '(a b)) => a`

cdr SUBR 1 arg

Returns the second component of a cons.

Example: `(cdr '(a b c)) => (b c)`

car SWITCH

cdr SWITCH

Officially car and cdr are only applicable to lists. However, as a matter of convenience the car and cdr of `nil` are `nil`. This allows programs to car and cdr off the ends of lists without having to check, which is sometimes helpful. Furthermore, some old programs apply car and cdr to objects other than lists in order to hack with the internal representation. To provide control over this, the value of car can be set to control which data types are subject to the car operation. Similarly, the value of cdr controls the cdr operation. Illegal operations will cause errors. For reasons of efficiency, this error checking is only enabled in `(*rset t)` mode (see page 3-29) and is mostly turned off in compiled programs. The values to which the switches may be set are:

Value	Operation applicable to
`list`	lists.
`nil`	lists and `nil`.
`symbol`	lists, `nil`, and symbols.
`t`	anything.

The default value of the switches is `nil`.

`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; caar, to car twice.

The car'ing and cdr'ing operations of these functions have error checking under the control of the car and cdr switches explained above, just as the car and cdr functions themselves do.

`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 (except when numbers are involved; as always numbers are not well-behaved with respect to eq):

```
    (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)
```

3.2 Lists

```
last                    SUBR 1 arg
```

`last` returns the last cons 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)))) ))
```
or by:

```
(defun length (x)
    (do ((n 0 (1+ n))
         (y x (cdr y)))
        ((null y) n) ))
```

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

`list` LSUBR 0 or more args

`list` constructs and returns a list of its arguments.

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

`list` could have been defined by:

```
(defun list nargs
    (do ((n nargs (1- n))
         (s nil (cons (arg n) s)))
        ((zerop n) s) ))
```
(This depends on parallel assignment to the control variables of do.)

`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)
```

To make a copy of the top level of a list, that is, to copy the list but not its elements, use `(append` *x* `nil)`.

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

```
(defun append2 (x y)
    (cond ((null x) y)
          ((cons (car x) (append2 (cdr x) y)) )))
```

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

```
(defun append argcount
    (do ((i (1- argcount) (1- i))
         (val (arg argcount) (append2 (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) e)) => (e (c d) b a)
```

reverse could have been defined by:

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

`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)**. This is a danger you always have to watch out for when using **nconc**.

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)
          ((nreversel x nil))))

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

nreconc SUBR 2 args

(nreconc *x y***)** is exactly the same as **(nconc (nreverse** *x***)** *y***)** except that it is more efficient.

nreconc could have been defined by:

```
(defun nreconc (x y)
    (cond ((null x) y)
          ((nreversel x y)) ))
```

using the same **nreversel** as above.

3.3 Alteration of List Structure

The functions `rplaca` and `rplacd` are used to make alterations in already-existing list structure. The structure is not copied but physically altered; hence caution should be exercised when using these functions as strange side-effects can occur if portions of list structure become shared unbeknownst to the programmer. The **nconc**, **nreverse**, and **nreconc** functions already described have the same property. However, they are normally not used for this side-effect; rather, the list-structure modification is purely for efficiency and compatible non-modifying functions are provided.

`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)`

See also `setplist` (page 2-55).

`subst`   SUBR 3 args

(`subst` *x* *y* *z*) substitutes *x* for all occurrences of *y* in *z*, and returns the modified copy of *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 just 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 z y) x)      ;if item eq to y, replace.
              ((atom z) z)      ;if no substructure, return arg.
              ((cons (subst x y (car z))  ;otherwise recurse.
                     (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 an association list (see the next section). The second argument is the S-expression in which substitutions are to be made. sublis looks at all atomic symbols in the S-expression; if an atomic symbol appears in the association list occurrences of it are replaced by the object it is associated with. 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 substitutions are made, the result is eq to the old S-expression.
Example:

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

In some implementations sublis works by putting temporary sublis properties on the atomic symbols in the dotted pairs, so beware.

3.4 Tables

Maclisp includes several functions which simplify the maintenance of tabular data structures of several varieties. The simplest is a plain list of items, which models (approximately) the concept of a *set*. There are functions to add (cons), remove (delete, delq), and search for (member, memq) items in a list.

Association lists are very commonly used. An association list is a list of dotted pairs. The car of each pair is a "key" and the cdr is "data". The functions assoc and assq may be used to retrieve the data, given the key.

Structured records can be stored as association lists or as stereotyped S-expressions where each element of the structure has a certain car-cdr path associated with it. There are no built-in functions for these but it easy to define macros to implement them (see part 6.2).

Simple list-structure is very convenient, but may not be efficient enough for large data bases because it takes a long time to search a long list. Maclisp includes some hashing functions (sxhash, maknum) which aid in the construction of more efficient, hairier structures.

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.
Example:

```
(catch (rplaca (or (member x z)
                   (throw nil lose))
               y)
       lose)
```

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))) ))
```

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 should be used for** value, not for effect. That is, use

```
(setq a (delete ´b a))
```

rather than

```
(delete ´b a))
```

The latter is *not* equivalent when the first element of the value of a is b.

(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 occurrences 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 for 2 or 3 args
    (deletel (arg 1)         ; pass along arguments...
             (arg 2)
             (cond ((= nargs 3) (arg 3))
                   (123456789.)))) ; infinity

(defun deletel (x y n)        ;auxiliary function
    (cond ((or (null y) (zerop n)) y)
          ((equal x (car y)) (deletel x
                                      (cdr y)
                                      (1- n)))
          ((rplacd y (deletel 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.

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, probably.

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.

Here is an example of how to use sxhash in maintaining hash tables of S-expressions:

```
(defun knownp (x)      ;look up x in the table
  (prog (i bkt)
    (setq i (plus 76 (remainder (sxhash x) 77)))
      ;The remainder should be reasonably randomized between
      ;-76 and 76, thus table size must be > 175 octal.
    (setq bkt (table i))
      ;bkt is thus a list of all those expressions that hash
      ;into the same number as does x.
    (return (member x bkt))))
```

To write an "intern" for S-expressions, one could

```
(defun sintern (x)
  (prog (bkt i tem)
    (setq bkt (table (setq i (+ 2n-2 (\ (sxhash x) 2n-1)))))
        ;2n-1 and 2n-1 stand for a power of 2 minus one and
        ;minus two respectively.  This is a good choice to
        ;randomize the result of the remainder operation.
    (return (cond ((setq tem (member x bkt))
                   (car tem))
                  (t (store (table i) (cons x bkt))
                     x)))))
```

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, if your intention is to "update" the "table" that was assoc's second argument.

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)

A typical trick is to say `(cdr (assoc x y))`. Since the cdr of `nil` is guaranteed to be `nil`, this yields `nil` if no pair is found (or if a pair is found whose cdr is `nil`.)

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)))
```

`sassoc` and `sassq` (see below) are of limited use. These are primarily leftovers from Lisp 1.5.

`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)))
```

`maknum` SUBR 1 arg

(maknum x) returns a positive fixnum which is unique to the object x; that is, (maknum x) and (maknum y) are numerically equal if and only if (eq x y). This can be used in hashing.

In the pdp-10 implementations, maknum returns the memory address of its argument. In the Multics implementation, an internal hash table is employed.

Note that unlike sxhash, maknum will not return the same value on an expression which has been printed out and read back in again.

`munkam` SUBR 1 arg

munkam is the opposite of maknum. Given a number, it returns the object which was given to maknum to get that number. It is inadvisable to apply munkam to a number which did not come from maknum.

3.5 Sorting

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. The array sort is not necessarily stable, that is equal items may not stay in their original order. However the list sort *is* stable.

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 `fillarray` 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**.

3.6 Hunks

This section applies only to the pdp10 implementation.

Hunks are a generalization of conses, useful in constructing more efficient data structures. A hunk is like a cons but it has more components; hunks come in several convenient sizes. The advantage of an n-element hunk over an n-element list is that the hunk occupies less space (half as much if n is a power of 2). The elements of a hunk can be referenced more efficiently than the elements of a list, since the compiler knows the relative locations of the components and addresses them directly.

The advantage of lists over hunks is flexibility; lists can be any length, can vary in length, can be altered by `rplacd`, and can be manipulated with a library of useful searching, sorting, and combining operations, previously described in this chapter.

Another feature of hunks is that at times one may treat a hunk as a cons, ignore the extra components. This allows the construction of list structure which has extra "frobs" stuck on at certain points. The `atom` function does not consider hunks to be atomic; it returns `nil` if given a hunk.

`print` represents hunks using an extended form of dot-notation; `read` however does not yet understand this notation. See the writeup on `print`.

hunk LSUBR 0 or more args

hunk takes any number of arguments, and returns a hunk whose components are the arguments. The first argument is the car, and the last is the cdr; that is, the arguments are in the order 1, 2, 3, ..., N-1, 0. This is the same order as `print` and makhunk use.

The maximum size of a hunk is 128 components. This may vary from implementation to implementation.

With no arguments, hunk returns `nil`. With one or two arguments, hunk returns a cons.

`cxr` SUBR 2 args

(`cxr` *n h*) returns the *n*'th component of the hunk *h*. car of a hunk returns the 1st component, and cdr of a hunk returns the 0th component.

`rplacx` SUBR 3 args

(`rplacx` *n h z*) replaces the *n*'th component of the hunk *h* with *z*. The value of `rplacx` is its (modified) second argument. `rplaca` of a hunk replaces its 1st component, and `rplacd` of a hunk replaces its 0th component.

`makhunk` SUBR 1 arg

(`makhunk` *n*), where *n* is a fixnum, creates and returns an *n*-element hunk, filled with `nil`s. (`makhunk` *l*), where *l* is a list, creates and returns a hunk of the appropriate length, initialized from *l*. This is like (`apply 'hunk` *l*).

Like hunk, makhunk will return `nil` or a cons if you ask for a hunk of 0, 1, or 2 elements.

`hunksize` SUBR 1 arg

`hunksize` returns the number of components in its argument. `hunksize` of `nil` is 0 and `hunksize` of a cons is 2.

`hunkp` VARIABLE

If the value of hunkp is `nil`, the functions `print`, `equal`, and purcopy treat hunks as conses, as most other system functions do. The extra elements are simply ignored. If the value of hunkp is non-`nil`, which is the default, these three functions deal with all the elements. sxhash always deals with all the elements.

4. Flow of Control

Maclisp provides a variety of structures for flow of control.

Functional application is the basic method for construction of programs. All operations are written as the application of a function to arguments. Maclisp programs are often written as a large collection of small functions which implement simple operations. Some of the functions work by calling others of the functions, thus defining some operations in terms of others.

Recursion exists when a function calls itself. This is analogous to mathematical induction.

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

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.

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 such as 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 on part 3.4.

Maclisp does not directly provide "hairy control structure" such as multiple processes, backtracking, or continuations.

4.1 Conditionals

and FSUBR

(and *form1 form2*...) evaluates the *forms* one at a time, from left to right. If any *form* evaluates to nil, and immediately returns nil without evaluating the remaining *forms*. If all the *forms* evaluate non-nil, and returns the value of the last one. 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 *form1 form2*...) evaluates the *forms* one by one from left to right. If a *form* evaluates to nil, or proceeds to evaluate the next *form*. If there are no more *forms*, or returns nil. But if a *form* evaluates non-nil, or immediately returns that value without evaluating any remaining *forms*. 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

The cond special form consists of the word cond followed by several *clauses*. Each clause consists of a *predicate* followed by zero or more *forms*. Sometimes the predicate is called the *antecedent* and the forms are called the *consequents*.

(cond (*antecedent consequent consequent...*)
      (*antecedent ...*)
      ...)

The idea is that each clause represents a case which is selected if its predicate is satisfied and the predicates of all preceding clauses are not satisfied. When a case is selected, its consequent forms are evaluated.

cond processes its clauses in order from left to right. First the predicate of the current clause is evaluated. If the result is nil, cond advances to the next clause. Otherwise, the cdr of the clause is treated as a list of forms, or consequents, which are evaluated in order from left to right. After evaluating the consequents, cond returns without inspecting any remaining clauses. The value of the cond special form 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, that is, if every antecedent is nil, that is, if no case is selected, the value of the cond is nil.

Example:
```
    (cond ((zerop x)    ;First clause:
           (+ y 3))     ; (zerop x) is antecedent.
                        ; (+ y 3) is consequent.
          ((null y)     ;A clause with 2 consequents:
           (setq x 4)   ; this
           (cons x z))  ; and this.
          (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.

4.2 Iteration

prog FSUBR

prog is the "program" special form. It provides temporary variables, sequential evaluation of statements, and the ability to do "gotos." A prog looks something like:

```
(prog (var1 var2...)
 tag1
    statement1
    statement2
 tag2
    statement3
    . . .
  )
```

var1, var2, ... are temporary variables. When the prog is entered the values of these variables are saved. When the prog is exited they are restored. The variables are initialized to `nil` when the prog is entered, thus they are said to be "bound to `nil`" by the prog. However, variables which have been declared fixnum or flonum will be initialized to `0` or `0.0` instead, but only in compiled programs. You should be careful about relying on the initial value of prog-variables.

The part of a prog after the temporary variable list is the body. An item in the body may be an atomic symbol or a number, which is a *tag*, or a non-atomic form, which is a *statement*.

prog, after binding the temporary variables, processes 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 processing its body and returns the value *x*. If (go *tag*) is evaluated, 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 Maclisp prog is an extension of the Lisp 1.5 prog, 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 special form, which uses a body similar to prog. The do, catch, and throw special forms 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 reasonable.

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

The do special form provides a generalized "do loop" facility, with an arbitrary number of "index 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 index 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 index variables are changed according to specified rules. do allows the programmer to specify a predicate which determines when the iteration will terminate. The value to be returned as the result of the form may optionally be specified.

do comes in two varieties.

The newer variety of do looks like:

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

The first item in the form is a list of zero or more index variable specifiers. Each index variable specifier is a list of the name of a variable *var*, an initial value *init*, which defaults to `nil` (or possibly zero, as mentioned under prog) if it is omitted, and a repeat value *repeat*. If *repeat* is omitted, the *var* is not changed between loops.

All assignment to the index variables is done in parallel. At the beginning of the first iteration, all the *inits* are evaluated, then the *vars* are saved, then the *vars* are setq'ed to the values of the *inits*. To put it another way, the *vars* are `lambda`-bound to the values of the *inits*. Note that the *inits* are evaluated *before* the *vars* are bound. At the beginning of each succeeding iteration those *vars* that have *repeats* get setq'ed to the values of their respective *repeats*. Note that all the *repeats* are evaluated before any of the *vars* is changed.

The second element of the do-form is a list of an end testing predicate *end-test* and zero or more forms, the *exit-forms*. At the beginning of each iteration, after processing of the *repeats*, 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-forms*. Note that the second element of the do-form resembles a cond clause.

If the second element of the form is `nil`, there is no *end-test* nor *exit-forms*, and the *body* of the do is executed only once. In this type of do it is an error to have *repeats*. This type of do is a "prog with initial values."

If the second element of the form is the S-expression `(nil)`, there is no *end-test* or *exit-forms*, and the *body* of the do is executed over and over. This is a "do forever." The infinite loop can be terminated by use of `return` or `throw`.

The remainder of the do-form constitutes 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 variety of do 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 older variety of do:

```
(setq n (cadr (arraydims x)))
(do i 0 (1+ i) (= i n)
        (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 ((n (cadr (arraydims x)))
     (i 0 (1+ i)))
    ((= i n)
  (store (x i) 0))
   ;this is like the example above,
   ;except n is local to the 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. On the other hand,

(do ((x) (y) (z)) nil *body*)

is identical to the prog above (it does not loop.)

```
    (do ((x y (f x))) ((p x)) body)
```

is like

```
    (do x y (f x) (p x) body)
```

The construction

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

exploits parallel assignment to index 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. Very often an iterative algorithm can be most clearly expressed entirely in the *repeats* and *exit-forms* of a new-style do, and the *body* is empty.

```
    (do ((x x (cdr x))
         (y y (cdr y))
         (z nil (cons (f x y) z)))  ;exploits parallel
        ((or (null x) (null y))     ; assignment.
         (nreverse z))              ;typical use of nreverse.
        )                           ;no do-body required.
```

is like (maplist ′f x y).

go FSUBR

The (go *tag*) special form is used to do a "go-to" within the body of a do or a prog. If the *tag* is an atom, it is not evaluated. Otherwise it is evaluated and should yield an atom. Then go transfers control to the point in the body labelled by a tag eq or = to the one given. (Tags may be either atomic symbols or numbers). If there is no such tag in the body, it is an unseen-go-tag error.

"Computed" go's should be avoided in compiled code, or altogether.

Example:

```
(prog (x y z)
  (setq x some frob)
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 typed-in S-expression (`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:

```
(do ((x x (cdr x))
     (n 0 (* n 2)))
    ((null x) n)
 (cond ((atom (car x))
        (setq n (1+ n)))
       ((memq (caar x) '(sys boom bleah))
        (return n))))
```

4.3 Non-local Exits

catch FSUBR

catch is the Maclisp 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 advised 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.

The catch-tag break is used by the break function.

`throw`   FSUBR

`throw` is used with **`catch`** as a structured non-local 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.

4.4 Causing and Controlling Errors

See the complete description of the Maclisp error system (part 3.4) for more information about how these functions work.

`error` LSUBR 0 to 3 args

This is a function which allows user functions to signal their own errors using the Maclisp 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 and *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, and the special conditions concerning **errset** (see page 3-16) are satisfied. *uint-chn* is the name of the user-interrupt channel to be used (an atomic symbol); see part 3.4.2. 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; in all other cases control is thrown back to top level, or to the nearest enclosing **errset**.

`errset` FSUBR

The special form `(errset` *form flag*`)` is used to trap an expected error. **errset** evaluates the *form*. If an error occurs during the evaluation of the *form*, the error is prevented from escaping from inside the **errset** and **errset** returns **nil**. If no errors occur, a list of one element, the result of the evaluation, is returned. The result is listified so that there will no ambiguity if it is **nil**. **errset** may also be made to return any arbitrary value by use of the **err** function.

The *flag* is optional. If present, it is evaluated before the *form*. If it is **nil**, no error message will be printed if an error occurs during the evaluation of the *form*. If it is not **nil**, or if it is omitted, any error messages generated will be printed.

Examples:

If you are not sure x is a number:

```
(errset (setq x (addl x)))
```

This example may not work in compiled code if the compiler chooses to open-code the addl rather than calling the addl subroutine. In general, one must be extremely foolhardy to depend on error checking in compiled code.

To suppress the error 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)
    (error ′(not a variable) a))
```

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.

5. Atomic Symbols

5.1 The Value Cell

Each atomic symbol has associated with it a *value cell,* which is a piece of storage that can refer to one Lisp object. This object is called the symbol's *value,* since it is what is returned if the symbol is evaluated. The binding of atomic symbols to values allows them to be used in programming the way "variables" are used in other languages.

The value cell can also be empty, in which case the symbol has no value and is said to be *unbound* or *undefined.* This is the initial state of a newly-created atomic symbol. Attempting to evaluate an unbound symbol causes an error to be signalled.

An object can be placed into a symbol's value cell by *lambda-binding* or by *assignment.* (See page 1-11.) The difference is in how closely the value-changing is associated with control structure and in whether it is considered a *side-effect.*

setq FSUBR

The setq special form is used to assign values to variables (atomic symbols.) **setq** processes the elements of its form in pairs, sequentially from left to right. The first member of each pair is a variable, the second is a form which evaluates to a value. The form is evaluated, but the variable is not. The value-binding of the variable is made to be 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 result of the evaluation of the last element of the **setq**-form.

Example: `(setq x (+ 1 2 3) y (cons x nil))`

This returns `(6)` and gives x a value of 6 and y a value of `(6)`.

Note that the first assignment is completed before the second assignment is started, 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)
    (eval (list 'setq x (list 'quote y))))
```

Alternatively, setq could have been defined by:

```
(defun setq fexpr (x)
  ((lambda (var val rest)
     (set var val)
     (cond ((null rest) val)
           ((apply (function setq) rest)) ))    ;if more, recurse
   (car x)
   (eval (cadr x))
   (cddr x)))
```

symeval   SUBR 1 arg

(symeval *a*) returns the value of *a*, which must be an atomic symbol. The compiler produces highly optimal code for symeval, making it much better than eval when the value of a symbol needs to be taken and the particular symbol to be used varies.

boundp SUBR 1 arg

The argument to boundp must be an atomic symbol. If it has a value, t is returned. Otherwise nil is returned.

makunbound SUBR 1 arg

The argument to makunbound must be an atomic symbol. Its value is removed, i.e. it becomes unbound.

Example:

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

makunbound returns its argument.

5.2 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" or, loosely, the "property." The indicator is generally an atomic symbol which serves as the name of the property. The value is any Lisp object.

For example, one type of functional property uses the atom expr as its indicator. 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 can be retrieved with the `plist` function. It is also possible to have "disembodied" property lists which are not associated with any symbol. These keep the property list on their cdr, so the form of a disembodied property list is `(<anything> . plist)`. The way to create a disembodied property list is `(ncons nil)`. Atomic symbols also (usually) keep their property list on their cdr, but you aren't allowed to know that. Use the `plist` function to get the property list of a symbol.

Property lists are useful for associating "attributes" with symbols. Maclisp uses properties to remember function definitions. The compiler uses properties internally to keep track of some of what it knows about the program it is compiling.

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. However, the same effect can be achieved by using properties with a value of `t` or `nil`.

Some property names are used internally by Maclisp, and should therefore be avoided in user code. These include `args`, `array`, `autoload`, `expr`, `fexpr`, `fsubr`, `lsubr`, `macro`, `pname`, `sublis`, `subr`, `value`, used by the Lisp system proper; `arith`, `*array`, `atomindex`, `*expr`, `*fexpr`, `*lexpr`, `numfun`, `number`, `numvar`, `ohome`, `special`, `sym`, used by the compiler; `grindfn`, `grindmacro`, used by the grinder.

`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. It is not an error for *x* to be a number, but `nil` will always be returned since numbers do not have property lists.
Example:

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

get could have been defined by:

```
(defun get (x y)
  (do ((z (cond ((numberp x) nil)
                ((atom x) (plist x))
                (t (cdr x)))
          (cddr z)))
      ((or (null z) (eq y (car z)))
       (cadr z))))
```

This relies on the fact that the car and the cdr of `nil` are `nil`, and therefore `(cadr z)` is `nil` if `z` is `nil`.

`getl` SUBR 2 args

(`getl` *x* *y*) is like get except that *y* is a list of indicators rather than just a single indicator. `getl` searches *x*'s property list until a property whose indicator appears in the list *y* is found.

The portion of *x*'s property list beginning with the first such property is returned. The car of this is the indicator (property name) and the cadr is the property value. `getl` returns `nil` if none of the indicators in *y* appear on the property list of *x*.

`getl` could have been defined by:

```
(defun getl (x pl)
   (do ((q (plist x) (cddr q))) ; scan down P-list of x
       ((or (null q) (memq (car q) pl))
        q)))
```

This definition is simplified and doesn't take numbers and disembodied property lists into account.

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 'Nixon 'not 'crook)
```

If the symbol already has a property with the same name that property is removed first. This ensures that `getl` will always find the property which was put on most recently. For instance, if you were to redefine an expr as a subr, and then redefine it as an expr again, this effect of putprop causes the evaluator to find the latest definition always.

A lisp definition of the basic putprop without the complications of numbers and disembodied property lists might be:

```
(defun putprop (x y z)
  (remprop x z)
  (setplist x (cons z (cons y (plist x))))
  y)
```

defprop   FSUBR

`defprop` is a version of putprop with no argument-evaluation, which is sometimes more convenient for typing. For instance,

```
(defprop foo bar oftenwith)
```

is equivalent to

```
(putprop 'foo 'bar 'oftenwith)
```

`remprop`   SUBR 2 args

`(remprop` *x* *y*`)` removes *x*'s *y*-property, by splicing it out of *x*'s property list. The value is `nil` if *x* had no *y*-property. If *x* did have a *y*-property, the value is a list whose car is the property, and whose cdr is part of *x*'s property list, similar to `(cdr (getl` *x* `'(`*y*`)))`.

x may be an atomic symbol or a disembodied property list. Example:

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

undefines the function `foo`, assuming it was defined by

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

`plist`   SUBR 1 arg

`(plist` *x*`)` returns the property list of the atomic symbol *x*.

`setplist`   SUBR 2 args

`(setplist` *x* *l*`)` sets the property list of the atomic symbol *x* to *l*. This is to be used with caution, since in some implementations property lists contain internal system properties which are essential to the workings of the Lisp system.

5.3 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 to be `print`'ed, the string is typed out.

See also page 2-83 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 come before all lower case letters.

pnget SUBR 2 args

(pnget *symbol n*) returns the pname of the *symbol* as a list of fixnums containing packed *n*-bit bytes. The legal values of *n* depend on the implementation; in the pdp-10 implementation, 6 (*SIXBIT*) and 7 (*ASCII*) are allowed. If this seems obscure, that's because it is. Example:

```
(pnget 'MUMBLERATOR 7) =>
               (-311246236550 -351327625542 -270_33)
```

pnput SUBR 2 args

This is a sort of inverse of pnget. (pnput (pnget *foo* 7) *flag*) returns a symbol with the same pname as *foo*. The symbol is interned if *flag* is non-nil.

5.4 Interning of Symbols

One normally wants to refer to the same (eq) atomic symbol every time the same pname is typed. Maclisp implements this through what is called the *obarray*. The obarray is a hash-table of atomic symbols. These symbols are said to be *interned*, or registered in the obarray. Whenever a pname is read in Lisp input, the obarray is searched for a symbol with the same pname. If one is found, the pname is considered to refer to that symbol. If not, a new symbol is created and added to the obarray.

The representation of an obarray is a Lisp array. The first 510. (or thereabouts) elements of the array contain lists which are buckets of a hash table. The last 128. elements of the array contain the "character objects," symbols with 1-character pnames. (These entries contain `nil` if the corresponding symbol has not yet been interned.) The character objects are treated specially for efficiency. There are usually one or two unused array elements between these two areas.

In order to allow for multiple name spaces, Maclisp allows multiple obarrays. An obarray can be made "current" by binding the symbol obarray to the appropriate array-pointer. See page 2-89 for details on how to manipulate obarrays and arrays in general.

It is possible to have a symbol interned on several obarrays at the same time. It is also possible to have two different (non-eq) symbols with the same pname interned on different obarrays. Furthermore it is possible to have a symbol which is not interned on any obarray, which is called an *uninterned* symbol. These are useful for purely-internal functions, but can cause difficulty in debugging since they can't be accessed directly. Such a symbol can be accessed via some data structure that contains it, set up by the program that created it.

Normally symbols are never removed from obarrays. It is possible for the user to explicitly remove a symbol from the current obarray. There is also a feature by which "truly worthless" symbols may be removed automatically (see page 3-60).

`intern` SUBR 1 arg

> (`intern` *x*), where *x* is an atomic symbol, returns the unique atomic symbol which is "interned on the obarray" and has the same pname as *x*. If no symbol on the current obarray has the same pname as *x*, then `intern` will place *x* itself on the obarray, and return it as the value.

`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`.

`copysymbol` SUBR 2 args

A subr of two arguments. The first argument must be a symbol, and the second should be `t` or `nil`. The result is a new, uninterned symbol, with the same pname as the argument. "Uninterned" means that the symbol has not been placed on any obarray. If the second argument is t, the new symbol will be given the same value as the original and will have a copy of its property list. Thus the new will start out with the same value and properties as the old, but if it is `setq`'ed or `putprop`'ed, the value or property of the old will not be changed. If the second argument is `nil`, the new symbol has no value and no properties (except possibly internal system properties.)

`gensym` LSUBR 0 or 1 args

`gensym` creates and returns a new atomic symbol, which is *not* interned on an obarray (and therefore 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.

5.5 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 page 2-89) 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 is *applied*, the function which it names is substituted.

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

The first example above is really just defining a synonym. Another way to do this is:

```
(defprop addone 1+ expr)
```

That is, an atomic functional property indicates synonyming. This can be particularly useful to define a macro by an expr or fexpr, or even by a subr.

The functions defprop and putprop may also be used for defining functions.

There is a feature by which, when a file of functions has been compiled and loaded into the lisp environment, the file may be edited and then only those functions which were changed may be loaded for interpretive execution. This is done by compiling with the "E" switch, and then reading in the source file with the variable defun bound non-nil. Each function will have an expr-hash property maintained, which contains the sxhash of the interpreted definition of the function. defun will only redefine the function if this hash-code has changed. This feature is rather dangerous since reasonable alterations to the function definition may not change the sxhash and consequently may not take effect. Because of its general losingness, this feature is only available in the pdp-10 implementation and sometimes not even there.

defun could have been defined by:

```
(defun defun fexpr (x)  ;circular, but you get the idea
  (prog (name type body)

    ; first, analyze the form, get arguments.
    (cond ((memq (car x) '(expr fexpr macro))
           (setq type (car x)
                 name (cadr x)
                 body (cddr x)))
          ((memq (cadr x) '(expr fexpr macro))
           (setq name (car x)
                 type (cadr x)
                 body (cddr x)))
          ((setq name (car x)
                 type 'expr
                 body (cdr x))))

    (setq body (cons 'lambda body))

    ; now, check for expr-hash hair.
    (cond ((and defun
                (get name 'expr-hash)
                (= (get name 'expr-hash)
                   (sxhash body)))
           )
          ; actually make the definition.
          ((putprop name body type)))
    (return name)))
```

`args` LSUBR 1 or 2 args

`(args` *f*`)` tells you 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.

`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
```

6. Numbers

For a description of the various types of numbers used in Maclisp, see part 1.2.

6.1 Number Predicates

`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`. It is an error if the argument is not a number. If that is possible `signp` should be used.

`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. The result is always `nil` if *x* is not a number. *c* can be one of the following:

l	means	x<0
le	"	$x \leq 0$
e	"	x=0
n	"	$x \neq 0$
ge	"	$x \geq 0$
g	"	x>0

Examples:
```
    (signp le -1) => t
    (signp n 0) => nil
    (signp g '(foo . bar)) => 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.
```

6.2 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. Use `equal` to compare bignums.

`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. If any argument is a flonum, the result will be a flonum. Otherwise, it will be a fixnum or a bignum depending on its magnitude.

`min`   `LSUBR 1 or more args`

`min` returns the smallest of its arguments, which must be numbers. If any argument is a flonum, the result will be a flonum. Otherwise, it will be a fixnum or a bignum depending on its magnitude.

6.3 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
(fix 104) => 104
```

`ifix` SUBR 1 arg

(`ifix` *x*) converts *x* from a flonum to a fixnum. `ifix` will never return a bignum, unlike `fix`. This allows it to be efficiently open-coded. This is not the same function as IFIX in Fortran; rounding is down rather than towards zero. It is like ENTIER in Algol.

`ifix` does not exist in the Multics implementation.

`float` SUBR 1 arg

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

```
(float 4) => 4.0
(float 3.27) => 3.27
```

`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))
                     (t 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 $|x|$. If *n* is negative, the result contains the $|n|$ low-order bits of $|x|$. If $|n|$ is bigger than the number of significant bits in *x*, $|x|$ is returned.

Examples:

```
(haipart 34567 7) => 162

(haipart 34567 -5) => 27

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

6.4 Arithmetic

General Arithmetic

These functions will perform arithmetic on any kind of numbers, and always yield an exact result, except when used with flonums. (Flonums have limited precision and range.) 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, or bignum representation if it cannot.

The two sections after this one describe other arithmetic functions which are more efficient but less powerful.

`plus` LSUBR 0 or more args

`plus` returns the sum of its arguments, which may be any kind of numbers.

`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.

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 z)` = x^z

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

As a special feature, expt allows its second argument to be a flonum, in which case the first argument is converted to a flonum and the exponentiation is performed in floating point, using logarithms. The result is a flonum in this case.

`*dif`   SUBR 2 args

`*dif` is a subr form of `difference`. It is documented here because some people use it. There is no reason to use it, since the compiler automatically converts `difference` into `*dif` as required.

`*quo` SUBR 2 args

`*quo` is a subr form of `quotient`. It is documented here because some people use it. There is no reason to use it, since the compiler automatically converts `quotient` into `*quo` as required.

Fixnum Arithmetic

These functions require their arguments to be fixnums and produce only fixnum results. If the true result, which would be returned by the more general functions described previously, is too large to be represented as a fixnum, the result actually returned will be truncated to an implementation-dependent number of bits, which is 36. (including the sign) in the Multics and pdp-10 implementations. The compiler produces highly-optimized code for these operations.

+ LSUBR 0 or more args

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

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

- LSUBR 0 or more args

This is the fixnum-only subtraction function. With one argument, it returns the number's negation. With more than one argument, it returns the first argument minus the rest of the arguments.

```
(-) => 0, the identity element
(- 3) => -3
(- 5 3) => 2
(- 2 6 -1) => -3
```

etc.

* LSUBR 0 or more args

* returns the product of its arguments. 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.

If used with more than one argument, it divides the first argument by the rest of the arguments. If used with only one argument, it returns the fixnum reciprocal of that number, which is -1, 0, 1, or undefined depending on whether the number is -1, large, 1, or 0.

```
(//) => 1, the identity element.
(// 20. 5) => 4
(// 100. 3 2) => 16.
etc.
```

1+ SUBR 1 arg

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

1- SUBR 1 arg

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

\ 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
```

\\ SUBR 2 args

This subr of two arguments is like gcd, but only accepts fixnums. This makes it faster than gcd.

SUBR 2 args

^ is the fixnum only exponentiation function. It is somewhat faster than **expt**, but requires its arguments to be fixnums, uses fixnum arithmetic, and always returns a fixnum result, which will be incorrect if the true result is too large to be represented as a fixnum.

Flonum Arithmetic

These functions require their arguments to be flonums, and always produce flonum results. If the true result is too large or too small to be represented as a flonum, an arithmetic underflow or overflow error will occur. (In the pdp-10 implementation these errors are not detected in compiled programs.) The compiler produces highly-optimized code for these operations.

+$ LSUBR 0 or more args

+$ returns the sum of its arguments.

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. When used with only one argument, it returns the number's negation. Otherwise, it returns the first argument minus the rest of the arguments.

```
(-$) => 0.0, the identity element
(-$ x) => the negation of x.
(-$ 6.0 2.5) => 4.5
(-$ 2.0 1.5 -3.6) => 3.1
etc.
```

***$** LSUBR 0 or more args

*$ returns the product of its arguments. 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. This function computes the reciprocal if given only one argument. If given more than one argument, it divides the first by the rest.

```
(//$) => 1.0, the identity element
(//$ 5.0) => 0.2
(//$ 6.28 3.14) => 2.0
(//$ 100.0 3.0 2.0) => 16.5
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.

^$ SUBR 2 args

`^$` is the flonum-only exponentiation function. The first argument must be a flonum, the second must be a fixnum (repeat, a *fixnum*), and the result is a flonum. To raise a flonum to a flonum power, use `(expt x y)` or `(exp (*$ y (log x)))`.

6.5 Exponentiation and Logarithm Functions

`sqrt` SUBR 1 arg

(sqrt *x*) => a flonum which is the square root of the number *x*. This is more accurate than (expt *x* 0.5). The following code, which is due to Gosper, should be written if the square root of a bignum is desired. It is essentially a Newton iteration, with appropriate precautions for integer truncation.

```
(defun bsqrt (n)
   (bsqrtl (abs n)
           (expt 2 (// (1+ (haulong n)) 2))))

(defun bsqrtl (n guess)
       ((lambda (next)
          (cond ((lessp next guess)
                 (bsqrtl n next))
                (t guess)))
        (quotient (plus guess (quotient n guess))
                  2)))
```

`exp` SUBR 1 arg

(exp *x*) => e^x

`log` SUBR 1 arg

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

6.6 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` SUBR 2 args

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

6.7 Random Functions

`random` LSUBR 0 to 2 args

(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. A useful function is:

```
(defun frandom ()
    (//$ (float (random 10000.)) 10000.0)))
```

which returns a random flonum between 0.0 and 1.0.

(random *n1 n2*) sets the random number seed from the pair of integers *n1, n2*.

`zunderflow` SWITCH

If an intermediate or final flonum result in the interpretive arithmetic functions (times, *$, expt, etc.) is too small in magnitude to be represented by the machine, corrective action will be taken according to the zunderflow switch.

If the value of zunderflow is non-nil, the offending result will be set to 0.0 and computation will proceed. If the value of zunderflow is nil, an error will be signalled. nil is the initial value.

In the pdp-10 implementation compiled code is not affected by zunderflow if the arithmetic in question was open-coded by the compiler. Instead, computation proceeds using a result with a binary exponent 256 higher than the correct exponent. In the Multics implementation zunderflow works the same for compiled code as for interpreted code.

See (sstatus divov), which controls division by zero (part 3.7).

6.8 Logical Operations on Numbers

These functions may be used freely for bit manipulation; the compiler recognizes them and produces efficient code.

`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)`.

The following macros are often convenient:

```
(defun logand macro (x)
  (subst (cdr x) 'f '(boole 1 . f)))

(defun logor macro (x)
  (subst (cdr x) 'f '(boole 7 . f)))

(defun logxor macro (x)
  (subst (cdr x) 'f '(boole 6 . f)))
```

Alternatively, these could be defined with **macrodef** (see part 6.2):

```
(macrodef logand x (boole 1 . x))

(macrodef logor x (boole 7 . x))

(macrodef logxor x (boole 6 . x))
```

`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. Zero-bits are shifted in to fill unused positions. The result is undefined if |*y*| > 36. The number 36 is implementation dependent, but this is the number used in both the Multics and pdp-10 implementations. 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. As with `lsh`, the number 36 depends on the implementation. Examples:

```
(rot 1 2) => 4
(rot -1 7) => -1
(rot 601234 36.) => 601234
(rot 1 -2) => 200000000000
(rot -6 6) => -501 .
```

The following feature only exists in the pdp-10 implementation.

The internal representation of flonums may be hacked using these functions. `lsh` or `rot` applied to a flonum operates on the internal representation of the flonum and returns a fixnum result. For example, `(lsh 0.5 0) => 200400000000` (octal). The following function also exists:

`fsc`   `SUBR 2 args`

(`fsc` *x* *y*) performs a FSC instruction on the two numbers *x* and *y*, and returns the result as a flonum. Consult the pdp-10 processor manual if you want to use this.

x and *y* may be fixnums or flonums; `fsc` just uses the machine representations of the numbers. If *y* is greater than 777777 octal, the FSC instruction is omitted and the possibly-unnormalized flonum with the same machine representation as *x* is returned.

7. Character Manipulation

7.1 Character Objects

An atomic symbol with a one-character pname is often called a *character object* and used to represent the ascii character which is its pname. In addition the atomic symbol with a zero-length pname represents the ascii null character. 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 page 2-58), 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) => /.
```

`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; *n* = 1 selects the leftmost character. The character is returned as a character object. `nil` is returned if *n* is out of bounds.

`getcharn` SUBR 2 args

`getcharn` is the same as `getchar` except that the character is returned as a fixnum instead of a character object.

`maknam` SUBR 1 arg

`maknam` takes as its argument a list of characters and returns an uninterned atomic symbol whose pname is constructed from 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 atomic symbol is interned. It is more efficient than doing `(intern (maknam x))`, although it is less efficient than plain `maknam` which should be used when interning is not required.

`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 part 5.1.) If macro characters are used, any usage in the macro character function of `read`, `readch`, `tyi`, or `tyipeek` not explicitly specifying an input file takes input from `readlists`'s argument rather than from an I/O device or a file. This causes macro characters to work as you would expect.

Examples:

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

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

`explode`		SUBR 1 arg

(`explode` *x*) returns a list of characters, which are the characters that would be typed out if (`prin1` *x*) were 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 '(+ /12 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 be typed out if (`princ` *x*) were 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 '(+ /12 3)) => ( /( + /  /1 /2 /  /3 /) )
```

`exploden`		SUBR 1 arg

(`exploden` *x*) returns a list of characters which are the characters that would be typed out if (`princ` *x*) were 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 number which is the ascii code for that character. cf. `explodec`. Example:

```
(exploden '(+ /12 3)) => (50 53 40 61 62 40 63 51)
```

`flatsize`		SUBR 1 arg

(`flatsize` *x*) returns the number of characters `prin1` would use to print *x* out.

`flatc`		SUBR 1 arg

(`flatc` *x*) returns the number of characters `princ` would use to print *x* out, without slashifying special characters.

7.2 Character 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. Also see the functions on page 2-56.

`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 "foo" "-" "bar") => "foo-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 whose pname is given as a character string argument. Contrary to previous editions of this manual, this atomic symbol *is* interned. Example:

```
(make_atom "foo") => foo
```

8. Arrays

As explained in part 1.2, an array is a group of cells which may contain Lisp objects. The individual cells are selected by numerical *subscripts*.

An array is designated by a special atomic object called an array-pointer. Array-pointers can be returned by the array-creation functions array and *array. An array-pointer may either be used directly to refer to the array, or, for convenience in referring to the array through input/output media, it may be placed on the property list of an atomic symbol under the indicator array, and then that symbol can be used as the name of the array.

There are several types of arrays. The main types are ordinary arrays, whose cells can hold any type of object, and number arrays, whose cells can only hold numbers. Number arrays permit more efficient code to be compiled for numerical applications, and take less space than an ordinary array which contains the same number of numbers. See the `array*` declaration (part 4.2) and the `arraycall` function (page 2-14).

When an array is created its type must be declared by giving a "type code." The type code for ordinary arrays is t. For number arrays, the type code is either `fixnum` or `flonum`. A particular number array can only hold one type of numbers because its cells contain the machine representation of the number, not the Lisp-object representation.

Some other types of arrays are: un-garbage-collected arrays, with a type-code of `nil`, which are the same as ordinary arrays except that they are not protected by the garbage collector and therefore can be used for certain esoteric hacks; obarrays, with a type-code of obarray, which are used to maintain tables of known atomic symbols so that the same atomic symbol will be referenced when the same pname is typed in; and readtables, with a type-code of `readtable`, which are used to remember the syntax specifications for the Lisp input reader. Normally, there is only one readtable and one obarray, supplied by the system, but the user may create additional readtables and obarrays in order to provide special non-Lisp environments or to gain additional control over the Lisp environment. Lisp functions such as read can be made to use an additional readtable or obarray by re-binding the variable `readtable` or obarray, respectively.

An array-pointer may also be *dead*, in which case it does not point to any array. One of the functions array, *array, or *rearray may be used to revivify a dead array-pointer.

The functions array and *array are used to create arrays. The first argument may be an atomic symbol, which makes that atomic symbol the name of an array, putting an array-pointer on its property list, or redefining an array-pointer that was already on the property list to point to the new array. Alternatively the first argument may be an array pointer, which causes that array pointer to be redefined to point to a new array, or it may be nil, which causes a new array pointer to be created and returned. Except in the latter case, array returns its first argument. *array always returns the array pointer, never the atomic symbol.

A readtable or an obarray may not be created with user-specified dimensions. The dimensions are always determined by Lisp. Other types of arrays allow any reasonable number (at least 3, anyway) of dimensions to be specified when they are created. The subscripts range from 0 up to 1 less than the dimension specified.

Ordinary and un-garbage-collected arrays are initialized to nil. Fixnum arrays are initialized to 0. Flonum arrays are initialized to 0.0.

Obarrays are initialized according to the third argument given to array or *array. nil causes a completely empty obarray to be created. Not even nil will be interned on this obarray. t causes the current obarray (value of the symbol obarray) to be copied. An array-pointer which is an obarray, or an atomic symbol which names an obarray, causes that obarray to be copied. If no third argument is given, the current obarray is copied.

Readtables are initialized in a similar fashion. If the third argument of array or *array is nil, then the current readtable is copied. If it is t, then the readtable being created is initialized to the initial standard Lisp readtable, including the macro characters ' and ;. (Note that this is the opposite of the t-nil convention for obarrays. This is for compatibility with the makreadtable function, which no longer exists.) An array-pointer or symbol of a readtable to be copied may also be given. If no third argument is given, the current readtable is copied.

An array-pointer may be *redefined* to an entirely different type and size of array, using the *array function. It remains the same array-pointer, eq to itself. If a variable was setq'ed to the array-pointer, that variable will now indicate the new array. If a symbol has that array-pointer on its property list, it will now be the name of the new array.

The *rearray function can be used to redefine the size or arrangement of dimensions of an array without losing its contents, or to make an array-pointer not point to any array (become dead). If there is only one argument, the array-pointer is killed, the array's contents are discarded, and the array-pointer becomes a "dead array" as described above. *array may now be used to redefine it as a new array.

If more than one argument is given to *rearray, they are the same arguments as to *array. *rearray with more than one argument cannot be used to change the type of an array, and cannot operate on a readtable or an obarray, but it can be used to change the dimensions of an array. The modified array will be initialized from its old contents rather than nil, 0, or 0.0. The elements are taken in row-major order for initialization purposes, and if there are not enough, nil, 0, or 0.0 will be used to fill the remaining elements of the modified array, according to the type.

The Multics implementation also has a type of arrays called *external* arrays. External arrays reside in a Multics segment rather than within the Lisp environment. They behave much like fixnum arrays, and should be declared as such to the compiler. To create an external array, use a form such as

(array foo external *pointer length*)

The *pointer* is a packed pointer to the beginning of the array, i.e. a fixnum whose first six octal digits are the segment number and whose second six octal digits are the word address. The *length* is the number of words in the array. External arrays can only have one dimension, can only contain fixnums, and are not initialized when they are created. They cannot usefully be saved by the save function. This type of array can be used for communication between Lisp programs and Multics programs or subsystems written in other languages, when large amounts of numerical data or machine words must be passed back and forth. See also defpl1 (part 4.6).

If you want the range of subscripts on arrays to be checked, it is necessary to set the *rset flag non-nil (i.e. run in (*rset t) mode) and to avoid the use of in-line array accessing (i.e. the array* declaration) in compiled programs. The amount of checking performed when *rset is nil and/or compiled code is used depends on the implementation.

Here is an example of a use of arrays:

```
(defun matrix-multiply (arr1 arr2 result)
   (and (eq (typep arr1) 'symbol)       ;convert arguments
        (setq arr1 (get arr1 'array)))  ;to array-pointers
   (and (eq (typep arr2) 'symbol)
        (setq arr2 (get arr2 'array)))
   (and (eq (typep result) 'symbol)
        (setq result (get result 'array)))
   (do ((ii (cadr (arraydims result)))  ;get relevant
        (jj (caddr (arraydims result))) ;dimensions
        (kk (cadr (arraydims arr2))))
       ()
     (do i 0 (1+ i) (= i ii)        ;result := arr1 x arr2
       (do j 0 (1+ j) (= j jj)
         (do ((k 0 (1+ k))
              (r 0.0))
             ((= k kk)
              (store (arraycall flonum result i j) r))
           (setq r (+$ r (*$ (arraycall flonum arr1 i k)
                             (arraycall flonum arr2 k j)
              )))))))
   result)
```

`*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 *b1* minus 1, the second from 0 to *b2* minus 1, etc. *y* is the type of array, as explained above. It may be chosen from among: **`t`**, **`nil`**, **`fixnum`**, **`flonum`**, **`readtable`**, **`obarray`**.

`array`   FSUBR

(`array` *x y b1 b2 ... bn*) has the same effect as (**`*array (quote`** *x*) (**`quote`** *y*) *b1 b2 ... bn*). This special form is provided for your typing convenience.

***rearray** LSUBR 1 or more args

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

(*rearray *x*) kills the array-pointer *x*, or the array-pointer which is the array property of the atomic symbol *x*. The storage used by the associated array is reclaimed. The value returned is t if *x* 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*. If it is a multi-dimensional array, row-major order is used. This means the last subscript varies the most rapidly as the array is traversed.

store FSUBR

The special form (store *array-ref value*) is used to store an object into a particular cell of an array. The first element of the form, *array-ref*, must be a subscripted reference to an array, or an invocation of arraycall. By coincidence, certain other forms work as *array-ref*, for instance (apply *f l*) where *f* turns out to be an array. The second element, *value*, is evaluated and stored into the specified 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))

(store (arraycall fixnum az i j) 43)
```

arraydims SUBR 1 arg

(arraydims *x*), where *x* is an array-pointer or an atomic symbol with an array property, 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)
```

`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 cells in the array.

(`fillarray` *x y*) fills the array *x* from the contents of the array *y*. If *y* is bigger than *x*, the extra elements are ignored. If *y* is smaller than *x*, the rest of *x* is unchanged. *x* and *y* must be atomic symbols which have array properties, or array-pointers. The two arrays must be of the same type, and they may not be readtables or obarrays.

The list-into-array case of `fillarray` could have been defined by:

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

An extension to the above 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` LSUBR 1 or 2 args

(`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 zero'th element. If the array has more than one dimension row-major order is used.

(`listarray` *array-name n*) is the same, except that at most the first *n* elements will be listed.

array-name may be an array-pointer or an atomic symbol with an array-property.

Number arrays may be efficiently saved in the file system and restored by using the functions `loadarrays` and `dumparrays`.

`loadarrays` SUBR 1 arg

(`loadarrays` *file-spec*) reloads the arrays in the file, and returns a list of 3-lists, of the form:

((*newname oldname size*) ...)

newname is a gensym'ed atom, which is the name of the reloaded array. (*newname* ought to be an array-pointer, but this function was defined before array-pointers were invented.) *oldname* is the name the array had when it was dumped. *size* is the number of elements in the array.

`dumparrays` SUBR 2 args

(`dumparrays` (*array1 array2* ...) *file-spec*) dumps the listed arrays into the specified file. The arrays must be fixnum or flonum arrays.

In both of the above, the *file-spec* argument is dependent on the system. In ITS or DEC-10 Lisp, the *file-spec* is a list of zero to four items, as in `uread`, and the same defaults apply. In Multics Lisp, the *file-spec* is an atomic symbol or a string which gives the pathname of the segment to be used. The defaults and other features of the Lisp I/O system are not applied. Only a segment may be specified, not a stream.

As a special compatibility feature, in Multics Lisp `loadarrays` will recognize a pdp-10 `dumparrays` file. (One can be moved to Multics through the ARPA Network File Transfer Protocol if the "type image" and "bytesize 36" commands are employed.) The pnames will be converted to lower case and flonums will be converted to the H6880 machine representation. `dumparrays` can create a file which pdp-10 `loadarrays` can read, including upper-case pnames and pdp-10 format flonums, if it is invoked as follows:

(`dumparrays` (*array1 array2*...) '(`pdp10` *file-spec*))

9. 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 is 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 list 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`. That is,

```
(defun mapcon (f x y)
    (apply 'nconc (maplist f x y)))
```

Of course, this definition is far less general than the real one.

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 functional form 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. If go or return is used the program may have to be compiled with the (mapex t) declaration, depending on the implementation, so watch out! Consider this function which is similar to and, except that it works on a list, instead of on separate arguments.

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

Admittedly this could be better expressed as a do:

```
(defun andl (x)
  (do ((y x (cdr y)))
      ((null y) t)
   (or (car y) (return nil))
 ))
```

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

		applies function to	
		successive sublists	successive elements
returns	its own second argument	map	mapc
	list of the function results	maplist	mapcar
	nconc of the function results	mapcon	mapcan

mapatoms   LSUBR 1 or 2 args

(mapatoms *fn obarray*) applies the function *fn* to all the symbols on the specified obarray. If the second argument is omitted, the current obarray is used. Note that the *obarray* argument must be an array-pointer, not a symbol which names an array. The symbol obarray is bound to the obarray being mapped over during the execution of mapatoms.

This function exists because some of the cells in an obarray contain lists of symbols and others contain single symbols, and user programs shouldn't have to know this. Example:

```
(mapatoms
  (function
    (lambda (x)
      (and (sysp x)
           (print (list x (sysp x) (args x))) ))))
```

Part 3 - The System

Table of Contents

1. The System

1.1 The Top Level Function

The following function is an approximation to what Maclisp does when it is at its "top level."

```
(defun standard-top-level nil
       (prog (^q ^w ^r evalhook base ibase ... )
        errors           ;errors, uncaught throws, etc. come here
        ^g               ;ctrl/G quits come here
             (reset-bound-vars-and-restore-pdls)
             (setq ^q nil)
             (setq ^w nil)
             (setq evalhook nil)
             (nointerrupt nil)
             (do-delayed-tty-and-alarmclock-interrupts)
;Recall that errors do (setq // errlist) so lambda-binding
;   errlist will work properly. See errlist.
    (mapc (function eval) //)
    (or (status linmode)(terpri))
    (do ((eof (list nil))      ;internal variables
         (prt '* *))
        (nil)            ;do forever (until ^g or error)
        (setq * (cond ((status toplevel)
                       (eval (status toplevel)))
                      (t (terpri)
                         (cond (prin1 (funcall prin1 prt))
                               (t (prin1 prt)))
                         (typ 40)
                         (setq (do ((form))(nil)
                                   (setq form
                                         (cond (read
                                                (funcall read
                                                         eof))
                                               (t (read eof))))
                                   (or (eq form eof)
                                       (return form))
                                   (terpri)))
                         (and (null read)
                              (atom -)
                              (is-a-space (tyipeek))(tyi))
                         ((lambda (+) (eval -))
                          (prog2 nil + (setq + -)))))))))
```

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. Errors and ^g quit to top level. That is they reinitialize and then re-enter this loop, printing a * (but not destroying the value of the variable *). Notice that there is a place in the middle where the user can insert his own special form to be evaluated, using (sstatus toplevel). It is also possible to change just the reader or just the printer by setq'ing read or prin1. See the sstatus function (page 3-77).

Variables used by the top-level read-eval-print loop:

*   VARIABLE

Contains the last S-expression printed out by the read-eval-print loop, that is, the value of the last form typed in. This is true even after an error return to top level, allowing one to refer to the value printed out before the aborted computation.

+   VARIABLE

Contains the last S-expression typed in. This can be used to edit it or to do it over again. (Notice how + is bound in the read-eval-print loop. This causes + to receive the correct value even if the evaluation aborts, since an error or ^g quit will undo the binding.)

-   VARIABLE

Contains the current S-expression typed in. This can be used by user-written error handlers. It can't be usefully accessed by expressions typed in, since it is set before the expression is evaluated.

By special arrangement the values of +, *, and - are preserved across a break. When the break is first entered these have the values for the last top-level operation, during the break they behave the same as at top level, and after the break returns they are restored to the values for the top level loop. (See break, page 3-5).

/   VARIABLE

/ is used to temporarily hold the value of errlist when an error returns to top level. This is so that lambda-binding errlist will have an effect (assuming no one lambda-binds /). Note that / must be typed in as // since the slash character is special to the LISP reader.

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. It doesn't apply if the environment started up was saved using (suspend). This feature is used to provide special error handling for subsystems written in LISP.

The symbol errlist is evaluated to get the list of forms in the binding context in which the error occurred, but the forms themselves are evaluated in the top-level binding context.

Example:

```
((lambda (errlist)
         (putprop 'foo 'bar 'baz)
         (hack)
         (remprop 'foo 'baz))
 (cons '(remprop 'foo 'baz)
       errlist))
```

The property list of foo will be properly restored even if the computation (hack) is aborted.

1.2 Breakpoints

Breakpoints are a mechanism to allow the user to gain control at any point in a program. Use of the function break causes a read-eval-print loop, similar to the one at top level, to be entered. (This is also called a *break loop*.) The user may evaluate any S-expressions, inspect the bindings of variables, and exit from the break in several ways. Normal execution then proceeds. (See page 2-43 and page 3-31)

This mechanism can be used to permit human intervention when an unexpected condition occurs. It is used in this way by the Maclisp error system. See the section on Exceptional Condition Handling, page 3-15. A break loop makes the full power of the LISP interpreter available for debugging.

`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 (using `princ`) to identify the break. It is not evaluated. If *pred* is omitted, `t` is assumed. Thus (break *tag*) is equivalent to (break *tag* t). (break *tag* nil) returns `nil`, and produces no action whatsoever.

A break loop is a read-eval-print loop similar to top level. break does an `errset` so that errors cannot cause an abnormal return from the break. A ^x quit, which causes an ordinary error, will thus return to the *break loop* if used to interrupt a computation started in the break loop. A ^g quit, however, returns back to LISP top level, resetting the environment using the `errlist`, as described above.

Two forms, $P and (return *x*), may be typed in a break loop. If $P is typed in, break returns `nil` and execution continues. This "$P" is <dollar> P in the Multics implementation, but <altmode> P in the PDP-10 implementations, followed of course in either case by a <space> or <newline> as appropriate (see (status linmode)). (An atom other than $P can be used to perform this function by changing the value of $P to another (non-`nil`) atom. The initial value of $P is always $P).

If (return *x*) is typed in, break evaluates *x* and returns that value. If as a result of the evaluation of a typed-in form, (throw *x* break) is evaluated, break returns *x* as its value. (Notice the distinction - executing a form (return *x*) does not return from break unless it was typed directly at the break loop.) See return, page 2-43.

When break returns, the state of the I/O system is restored.

An approximate LISP definition of what break does follows. Note that the user program can modify this by using (sstatus breaklevel).

```
(defun break fexpr (x)
       (*break (eval (cadr x)) (car x)) ;note argument reversal

(declare (special ^q ^w evalhook * + -))

(defun *break (breakp breakid)
 (and breakp
  (do ((^q nil) (^w nil) (evalhook nil) (terpri t) (* *) (+ +) (- -))
      ()                          ;bind key variables
    (terpri msgfiles)             ;msgfiles arguments
    (princ '|;bkpt | msgfiles)     ;  used for
    (princ breakid msgfiles)      ;  Newio only
    (terpri msgfiles)
    (setq + -)                    ;last form typed
    (return
     (prog2 nil
      (catch
       (do () (nil)               ;do forever (until throw)
         (errset
           (do ((eof (list nil)) (form))
               (nil)
            (cond ((status breaklevel)
                   (eval (status breaklevel)))
                  (t (setq form (cond (read (funcall read eof))
                                      (t (read eof))))
                     (and (null read)
                          (atom form)
                          (is-a-space (tyipeek))
                          (tyi))
                     (cond ((eq form eof) (terpri))
                           ((and $P (eq form '$P)) (throw nil break))
                           ((eq (car form) 'return)
                            (throw (eval (cadr form)) break))
                           (t (setq - form)
                              (print
                               (setq * ((lambda (+) (eval form))
                                        (prog2 nil + (setq + -)))))
                              (terpri))))))))
       break)
      (terpri)
    ))))
```

The arguments to break are a breakpoint identification and (optionally) a break switch. If the break switch evaluates to nil, then nil is returned. Otherwise, the variables ^q, ^w, evalhook and terpri are bound to nil, the variables *, +, and - are bound to their current values, and the message ";bkpt *<breakid>*" is printed. A read-eval-print loop similar to the top level loop is then entered. This break loop is surrounded by an errset. Errors or typing ^x merely cause the break loop to be re-entered. The value of (status breaklevel) serves a function similar to that of (status toplevel) in the top level loop.

As each form is read in the default break loop, there are four cases:

1. End of file. For console input this merely indicates rubout beyond the number of input characters. Whether input is from console or elsewhere, the (terpri) is done and the reader is entered.

2. The form is the atom $P or eq to the non-nil value of $P. nil is returned from the break.

3. The form is (return *value*). The form *value* is evaluated and returned from the break.

4. Otherwise the form is evaluated and the result printed out in a manner analogous to the top level read-eval-print loop. The variables +, -, and * are updated appropriately. (Recall, however, that they were bound on entry to *break, and so will be restored eventually.)

The way to return from a break is to do a throw with a tag of break; this will return from the catch which surrounds the break loop. This is how cases 2 and 3 return their values; case 4 may also cause a return from the break.

1.3 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 by directly calling the function associated with the particular control key (see below).

Although control characters are usually processed as soon as they are typed, they may be delayed if there is a garbage collection in progress or LISP is in `(nointerrupt tty)` mode - see the `nointerrupt` function (page 3-18).

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 normally echo as an uparrow or circumflex followed by the character.

In Newio, any character at all may be made an interrupt character. See `(sstatus tty)` and `(sstatus ttyint)`.

Interrupt characters are also read as part of the terminal input stream. Normally they are marked in the `readtable` as "worthless" characters.

Entering Control Characters in TOPS-10 LISP

Most 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 typing the CTRL/c character one or two times, thereby returning to the monitor. The monitor command REENTER may then be used to re-enter the LISP. LISP will print "?^" and read a character, which may be a control character or a character whose "control" meaning is to be used. Thus typing either ^g or "G" will cause the ^g interrupt to occur. If LISP is not ready to take an interrupt, there may be a delay before the "?^" is printed.

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" (@S S or @S I P from a TIP) signal should be transmitted.) LISP responds by typing out "CTRL/". Now you may type one letter from the list later in this section, which will be interpreted to have its "control" meaning. This control character must be followed by a newline.

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.

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.

Example for Multics LISP:
(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
        *
```

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 binding the appropriate symbol to the function to handle it, or by using the `(sstatus ttyint)` function (page 3-78).

In the following descriptions of control characters, those which can always be processed immediately, even during a garbage collection or in `(nointerrupt 'tty)` mode, will be indicated by an "!" for the ITS and TOPS-10 implementations, and an "☆" for the Multics Implemention. When appropriate, equivalent LISP code is given for producing the same result from a user program.

Control Characters that have initially defined meanings in all implementations:

B !☆	runs a break loop with *breakid* "^b" (see `break`, page 3-5). (In the PDP-10 Oldio implementation ^h is used instead of ^b.) `(break ^b)`.
C !☆	sets the value of the atom ^d to `nil`, turning off garbage collector messages. Because ^c is trapped by DEC-10 monitors, this control character can only be typed using the REENTER mechanism and typing "C". `(setq ^d nil)`.
D !☆	sets the value of the atom ^d to `t`, turning on garbage collector messages. `(setq ^d t)`
G	quits back to the top level of LISP, rebinding all variables to their global values, resetting various system variables, and evaluating the `errlist` forms. This is used to stop a running program when there is no intention of restarting it again. (Prints out a *; see the "top level" function and the `^g` function.) H is used instead of ^b in some implementations (see above).
Q !	sets the value of the atom ^q to `t`, enabling input from the source selected by the value of `infile`, or selected by use of the function `uread`. In the PDP-10 Newio implementation, this is not an interrupt character; instead it is a macro character, and takes effect only if processed by `read`. `(setq ^q t)`.
R !	sets the value of the atom ^r to `t`, enabling output to the destinations selected by the value of `outfiles`, or selected by use of the `uwrite` function. `(setq ^r t)`.
S !	turns off typeout until input is read. This is used to suppress the rest of the typeout from the current request, without affecting typeout from the next request that is typed in. It is implemented by setting ^w to `t`, then putting a macro character in the input stream which sets ^w to `nil` and does a `(terpri)` when it is read.
T !	sets the value of the atom ^r to `nil`, disabling output to the destinations that CTRL/r enables. `(setq ^r nil)`.

U	causes the current call to `read` to be restarted from the beginning. (Not available in PDP-10 implementations).
V !	sets the value of the atom `^w` to `nil`, enabling output to the terminal. `(setq ^w nil)`.
W !	sets the value of the atom ^w to t, disabling output to the terminal. `(setq ^w t)` (and possible also `(clear-output tyo)`).
X	causes an error which can be caught by **errset**. This is a less drastic "quit" than CTRL/g. If it is typed within a break loop, it will return no further than the break loop, since break uses **errset**. `(error 'quit)`.
Z !*	On ITS returns to ITS command level, i.e. DDT. On Multics returns to Multics command level. (start re-enters LISP.) On TOPS-10 goes to DDT if a DDT has been loaded with LISP.

The following control characters only exist in the Multics implementation.

. !*	does nothing, and is used merely to speed up a slow process by causing an interaction.
? !*	asks the LISP subsystem what it is doing: running, waiting for input, collecting garbage, or running with tty-interrupts masked off.

The following control characters only exist in PDP-10 implementations with the "moby I/O" capability. (For now, this means only at the MIT AI Laboratory.) (see `(sstatus ttyscan)`, page 3-81).

F	Cause graphics display slave to seize a display.
N	Turn on display for character output.
O	Turn off display for character output.
Y	Cause display slave to release display.

The following control characters only work in the PDP-10 implementation. They are not really interrupts, but occur only when processed by the terminal input prescanner (see `(sstatus ttyscan)`, page 3-81).

K	redisplay the current input. This allows you to get a clean copy of your

input after rubouts have been used. (On a display console, LISP will attempt to make rubbed-out characters actually disappear from the screen. ^k is still useful if a timing error disrupts this process.)

L erases the screen if the terminal is a display, then does a CTRL/k.

Control-Character Functions

^g SUBR 0 args

Produces a quit to top level just as if a CTRL/g had been typed.

These functions exist only in the PDP-10 Oldio and the Multics implementations. They are being phased out, so using them in new programs is not recommended.

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, atomic symbols' pnames are processed character by character, except that nil is ignored. 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."

```
        (iog a x1 x2 ... xn)
```

can also be written

```
        ((lambda (^q ^r ^w)
                 (ioc a)
                 x1
                 x2
                 ...
                 xn)
         nil nil nil)
```

1.4 Exceptional Condition Handling

1.4.1 The LISP Error System

The errors detected by the LISP system 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 evaluation in which it occurs to be aborted. When an uncorrectable error occurs, the first thing that happens is the printing of an error message. In Oldio, the error message goes to the terminal and nowhere else (unless suppressed, see errset, page 3-20), no matter how the I/O switches and variables are set. In Newio, the variable msgfiles is a list of the files to which error messages should be routed; this initially is just the terminal. The error message consists of some explanatory text and (usually) 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 function errset (and break, which uses errset). 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 free (without being bound).

What happens next depends on how the error-catcher was set up. At top level, the forms on the errlist are evaluated and the top level loop (or a user specified top level form) is re-entered. (The symbol errlist is evaluated prior to the above restoration of bindings, and saved in the variable /. In this way the errlist used is the one current at the time of the error, despite the restoration of bindings.) 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 ringing the bell on the terminal. If an error returns to errset, errset returns nil and evaluation proceeds. If an error returns to top level, the state of the world is reset and * is typed.

The above description is slightly simplified. The user can request an interrupt to occur between the typing of the message, and the unwinding of bindings and return of control to an error-catcher. If the error is going to return to top level, the *rset-trap user interrupt is signalled. This user interrupt is initially a system-supplied 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. In (*rset t) mode a break loop is entered, but in (*rset nil) mode the user interrupt is ignored. If the error is going to return to a break or an errset, and *rset is non-nil, the errset user interrupt 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 as if it were an uncorrectable error.

When a correctable error occurs, a user interrupt is signalled. See page 3-19 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 an S-expression describing the error. See section 1.4.2 for details. If the user interrupt handler is nil, or if it returns a non-list, the error is treated as an uncorrectable error. But if the handler returns a list, the first element 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 not top-level, correctable errors will be treated as uncorrectable errors unless there is a non-null handler for the errset interrupt. This is to prevent confusing "multiply nested" error breaks unless the user indicates that he is sophisticated by setting up a handler for the errset interrupt. (The errset handler itself is only invoked if *rset is non-nil, however.)

See the functions error, err, and errset.

1.4.2 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, asynchronous I/O conditions, the garbage collector, and many of the errors that are detected by the interpreter or by the system functions. Errors detected by user functions can use this mechanism also.

The user interrupts are divided up into several channels. 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. (A few interrupt handlers take more than one argument. See the specific descriptions.) 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.

Interrupts can be either synchronous (e.g. errors and garbage collector interrupts) or asynchronous (control characters, `alarmclock`, etc.). To prevent timing errors, asynchronous interrupts are always run in `(nointerrupt t)` mode. A handler for an asynchronous interrupt must explicitly do `(nointerrupt nil)` to permit other asynchronous errors to interrupt it. (For example, the system-supplied ^b handler does this so that control character interrupts can be used within the ^b **break** loop.)

The service functions for most user interrupts are kept as the values of symbols with mnemonic names. A list of these symbols begins on page 3-19. There are also user interrupts for control characters. The service functions for these are declared using `(sstatus ttyint)`. See page 3-78.

The initial values for the service functions of the various interrupts are provided by the system as **break** loops for some interrupts and `nil` for others (except for some control characters).

There are some special considerations for user interrupts signalled by correctable error conditions. The argument to the service function is a description of the error whose exact form is described in the catalogue at the end of this section. If the service function returns `nil` (or any other 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, the first element of the list is used to attempt recovery from the error. The exact way that it is used is described in the catalogue. If recovery is successful execution proceeds from the point where the error occurred. If recovery is unsuccessful another error is signalled.

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. Note that the system-supplied error service functions consistently bind *args* to the argument supplied. The user can check the value of this variable to see what is wrong. Note too that the system-supplied error handlers restore `readtable` and **obarray** before breaking.

```
(defun +internal-ubv-break (args)
       (declare (special args))
       (errprint nil msgfiles)          ;print error message
       ((lambda (readtable obarray)
                (nointerrupt nil)
                (break unbnd-vrbl))
        (get 'readtable 'array)
        (get 'obarray 'array)))
(setq unbnd-vrbl '+internal-ubv-break)
```

alarmclock   SUBR 2 args

alarmclock is a function for controlling timers. It can start and stop two separate 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 non-negative number (fixnum or flonum) 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, the alarmclock user interrupt occurs. The service function is run in (nointerrupt t) mode so that it will not be interrupted while it is performing its service. 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 the atom time or the atom **runtime**, depending on which timer went off. See also the function **nointerrupt**.

`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. A similar deferral technique is used by the LISP system itself to protect against interrupts in the garbage collector.

`(nointerrupt 'tty)` prevents control characters (typed on the terminal, or "tty") from causing interrupts; however, `alarmclock` interrupts (and other asynchronous interrupts) are still allowed. Any non-`tty` asynchronous interrupts which were saved will now go off.

`(nointerrupt nil)` turns interrupts back on. Any interrupts which were saved will now get processed. This is the normal, initial state.

The result returned from `nointerrupt` is the previous interrupt status - `nil`, t, or `tty`.

Example:

```
((lambda (oldstatus)
         <protected code>
         (nointerrupt oldstatus))
 (nointerrupt t))
```

1.4.3 Catalogue of User Interrupt Channels

Each user interrupt channel (except some associated with files) has a variable whose value is a functional form, the *service function* for that channel. The name of the interrupt channel is the same as the name of the variable. The following lists the user interrupt channels in alphabetical order. The argument to which the service function is applied and the value which it should return are described. By convention, almost all service functions receive one argument. Some user interrupts are initially set to a system-supplied handler which binds the variable args to this argument and enters a `break` loop. The name of the interrupt is used as the `break` identifer.

Some user interrupts ignore the value returned by the service function, while others distinguish two cases. If the value is atomic, the service function was not able to recover from the condition that caused the interrupt. LISP will take its default action, such as returning control to the most recent `errset`. If the value is a list, the car of that list is used to recover from the condition that caused the interrupt. It is usually a new piece of data to be used in place of the one that was being complained about, or a new form to be evaluated in place of the form that erred.

If the value of the service-function variable is `nil` instead of a functional form, the user interrupt is considered to be turned off. The system behaves as if the function had run and returned `nil`.

Some user interrupts are asynchronous in nature, and are executed in `(nointerrupt t)` mode to prevent timing errors. The interrupt handler may choose to run in `(nointerrupt nil)` mode, however, as the initial `^b` handler does. The `nointerrupt` mode is restored after the handler is run. Such interrupts are themselves deferred by `(nointerrupt t)` mode.

`alarmclock`   VARIABLE

The value of `alarmclock` is the service function for the user interrupt signalled when a timer set up by the `alarmclock` function goes off. The argument is the name of the timer which went off, `time` or `runtime`. The returned value is ignored. The service function is executed in `(nointerrupt t)` mode. This interrupt is initially turned off.

`autoload`   VARIABLE

The value of `autoload` is the service function for the user interrupt which provides automatic loading of program packages into the environment. The argument is `(`*function-name . autoload-property*`)`. The returned value is ignored. See page 3-26 for details. This interrupt is initially set to a function which simply loads a file.

`cli-message`   VARIABLE

The value of `cli-message` service handler for the user interrupt signalled when another job has interrupted the LISP job via the CLI device. The argument is `nil` and the returned value is ignored. A user handler is expected to open a file on the CLA device and read the message from the other job. The service function is run in `(nointerrupt t)` mode. This interrupt is initially turned off. Currently, it exists only in the ITS Newio implementation.

`errset` VARIABLE

The value of `errset` is the service function for the user interrupt which is signalled when an error is caught by an `errset` and `*rset` is non-`nil`. The argument is `nil` and the returned value is ignored. This user interrupt is initially off. Turning it on affects the behavior of the error system (see page 3-16).

`fail-act` VARIABLE

The value of `fail-act` is the service function for the user interrupt which is signalled when any of a large variety of miscellaneous error conditions occurs. The argument is a list whose first element is generally a symbol which describes the type of error condition. The rest of the list contains various objects related to the error. The returned value depends on the error. These are not standardized and will not be described here. This interrupt is initially set to a `break` loop.

`gc-daemon` VARIABLE

The value of `gc-daemon` is the service function for the user interrupt which is signalled after each garbage collection. The argument is a list of items; in the PDP-10 implementation each item is of the form (*space-name free-before free-after size-before size-after*) and in the Multics implementation, each item is of the form (*space-name free-before* . *free-after*). The returned value is ignored. This interrupt is initially turned off.

`gc-lossage` VARIABLE

The value of `gc-lossage` is the service function for the user interrupt which is signalled when there is no more available address space or when the Time Sharing Monitor rejects a request for more memory. In the Multics implementation, there is always enough memory, so this user interrupt never occurs. In the PDP-10 implementation the argument is the name of the space that lost, and the returned value is ignored. This interrupt is initially set to a `break` loop.

`gc-overflow` VARIABLE

The value of `gc-overflow` is the service function for the user interrupt which is signalled when a space overflows its gcmax. (see `alloc` and `(status gcmax)`.) The argument is the name of the space. The returned value is ignored. This interrupt is initially set to a break loop.

io-lossage VARIABLE

The value of io-lossage is the service function for the user interrupt which is signalled when the I/O system encounters an error (for example, a file which was being opened was not found). The argument is a list of the name of the function which erred and its arguments, which may have been standardized or otherwise partially digested. The returned value is a list of a new form to be evaluated in place of the call to the function which erred. This interrupt is initially set to a break loop.

machine-error VARIABLE

The value of machine-error is the service handler for the user interrupt signalled when some difficulty is experienced by the host machine. The service function receives four arguments instead of one. The first is an atomic symbol indicating the type of error:

eval	illegal machine operation
examine	attempt to reference non-existent memory
deposit	attempt to write into read-only memory
oddp	parity error

The other three arguments are fixnums which are addresses of memory locations. The second is the location of the error; the third is the program counter when the error occurred; and the fourth is the JPC (the program counter as of the last jump instruction before the error occurred). The machine-error handler may signal a different kind of error or a ^g quit (see the ^g function) if desired, or enter a break loop. The subr function (see page 3-101) may be useful in decoding the three fixnum arguments. If the handler returns, the value is ignored and the erroneous operation is retried. If the user provides no machine-error handler (the interrupt is initially turned off), the error is handled in the default manner for the host machine. On ITS, this puts the user in DDT. This currently exists only in the PDP-10 Newio implementation.
Example:

```
(defun machine-error-handler (type loc pc jpc)
       ((lambda (args terpri)
                (declare (special args terpri))
                (terpri msgfiles)
                (cond ((eq type 'examine)
                       (princ '|;REFERENCE TO NON-EXISTENT
                              MEMORY| msgfiles))
                      ((eq type 'deposit)
                       (princ '|;WRITE INTO READ-ONLY MEMORY|
                              msgfiles))
                      ((eq type 'eval)
                       (princ '|;ILLEGAL MACHINE OPERATION|
                              msgfiles))
                      ((eq type 'oddp)
                       (princ '|;MEMORY PARITY ERROR|
                              msgfiles))
                      (t (princ '|;UNKNOWN MACHINE ERROR|
                                msgfiles)))
                (princ '| FROM LOCATION | msgfiles)
                (princ pc)
                (princ '| IN FUNCTION | msgfiles)
                (prinl (subr pc))
                (break machine-error))
        (list type loc pc jpc)
        t))
```

mar-break VARIABLE

The value of mar-break is the service handler for the user interrupt signalled when the memory location specified by (sstatus mar) (see page 3-95) has been accessed in the specified manner. The argument is nil and the returned value is ignored. The service function is run in (nointerrupt t) mode. Also, LISP implicitly performs (sstatus mar 0 nil) before running the user interrupt; this helps to prevent infinite loops. This interrupt is initially turned off. It currently exists only in the ITS Newio implementation. See page 3-55 for more information on using this interrupt.

pdl-overflow VARIABLE

The value of pdl-overflow is the service function for the user interrupt which is signalled when a pushdown list exceeds its pdlmax. (see alloc and (status pdlmax).) The argument is the spacename of the pushdown list. The returned value is ignored. This interrupt is initially set to a break loop.

sys-death VARIABLE

The value of sys-death is the service handler for the user interrupt signalled when the time-sharing system is about to go down, has been revived from that state, or is being debugged. The argument is nil and the returned value is ignored. A user handler may wish to examine the result of (status its) to determine the state of the system. The service function is run in (nointerrupt t) mode. This interrupt is initially turned off. This currently exists only in the ITS Newio implementation.

tty-return VARIABLE

The value of tty-return is the service handler for the user interrupt signalled when control of the terminal is returned to the LISP job by its superior. This allows LISP to determine that the display screen may have been changed by other jobs. The argument is nil and the returned value is ignored. The service function is run in (nointerrupt t) mode. This interrupt is initially turned off. This currently exists only in the ITS Newio implementation.

unbnd-vrbl VARIABLE

The value of unbnd-vrbl is the service function for the user interrupt which is signalled when an attempt is made to evaluate an atomic symbol which does not have a value (an unbound variable.) The argument is a list of the symbol which could not be evaluated. The returned value is a list of a new symbol to be evaluated in its place. This interrupt is initially set to a break loop.

undf-fnctn VARIABLE

The value of undf-fnctn is the service function for the user interrupt which is signalled when an attempt is made to apply an undefined function. The argument is a list of the functional form which could not be applied. The returned value is a list of a new functional form to take its place. This interrupt is initially set to a break loop.

unseen-go-tag   VARIABLE

The value of unseen-go-tag is the service function for the user interrupt 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. The argument is a list of the erroneous tag. The returned value is a list of a new tag to replace it. This interrupt is initially set to a break loop.

wrng-no-args   VARIABLE

The value of wrng-no-args is the service function for the user interrupt which is signalled when a function is called with the wrong number of arguments. The argument is a list of two items: First, a list of the function and the arguments that were passed. Second, the lambda-list if the function was interpreted, or the same dotted pair as args returns if the function was compiled, or the atom ? if this information could not be determined. The returned value is a list of a new form to be evaluated in place of the losing one. This interrupt is initially set to a break loop.

wrng-type-arg   VARIABLE

The value of wrng-type-arg is the service function for the user interrupt which is signalled when an argument is passed to a system function which is not acceptable to that function. The argument is a list of the argument which was not accepted. The returned value is a list of a new argument to replace it. That is, directly an argument, *not* a form to be evaluated to get an argument. This interrupt is initially set to a break loop.

*rset-trap   VARIABLE

The value of *rset-trap is the service function for the user interrupt 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. This is so that the user will not be bothered unless he has put LISP in debugging mode. The argument is nil and the returned value is ignored. This interrupt is initially set to a function which enters a break loop if *rset is non-nil.

There are other interrupt handlers which are associated with I/O files or inferior jobs. See eoffn, endpagefn, (sstatus ttyint), and create-job.

1.4.4 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, and fails to find one, it looks for a property under the indicator autoload, and it it finds one, automatic loading will occur.

Automatic loading is performed by means of the autoload user interrupt; thus the user may assert any desired degree of control over it. When the autoload property is encountered, the user interrupt 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 this user interrupt could have been defined by:

```
(setq autoload
      (function (lambda (x) (load (cdr x)) )))
```

From this one can see that the value of the autoload property should be the name of the file which contains the definition of the function. Note: in the TOPS-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.

When the interrupt handler returns, it should 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."

Examples of setting up functions to be autoloaded:

In the Multics implementation:

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

In the PDP-10 Oldio implementation:

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

In the PDP-10 Newio implementation:

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

or (putprop ′foo ′|dsk:me;foo fasl| ′autoload) or the Oldio version also works.

1.5 Debugging

1.5.1 Binding, Pdl Pointers, and the Evaluator

The Maclisp evaluator is 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` in the next section.)

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 a binding context 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. Since the fixnum value of a pdl pointer has only internal meaning, generally a pdl pointer cannot be obtained from user input, except by the user typing in a pdl pointer chosen from a list of pdl pointers typed out at him. The "frame" functions described in the next section may be used to obtain pdl pointers.

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.

1.5.2 Functions for Debugging

`*rset` SUBR 1 arg

(`*rset` *x*) sets the `*rset` switch to `nil` if *x* is `nil`, or to `t` if *x* is non-`nil`, and returns the value it set it to. (See below). This function exists primarily for user typing convenience.

*rset SWITCH

If the *rset switch is non-nil, extra information is kept by the interpreter to allow the debugging functions, such as baktrace and evalframe, to work. In addition, the interpreter will make extra checks such as checking the number of arguments passed to a subr or lsubr and checking that array subscripts lie within the declared bounds. Generally, the *rset switch being on means "I am debugging"; this is known as "*rset mode". The initial state of the switch is nil.

nouuo SUBR 1 arg

(nouuo t) sets the nouuo switch.

(nouuo nil) turns off the nouuo switch. (This is the initial state.)

nouuo returns t or nil according to whether it turned the nouuo switch on or off. (See below.) This function exists primarily for user typing convenience.

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, which is the normal case, compiled calls can be made to go directly, which is much faster.

The nouuo switch may be turned off at any time. Each compiled function call will only go through the interpreter once more, at which time it will be linked directly. If the compiled code has been reloaded into the system with the PURE option (see page 3-71) then this direct link may be unsnapped and the Interpreter route re-established by (sstatus uuolinks). Because the PURE option requires an amount of extra space and time, it is not normally on; thus links snapped in code loaded as non-PURE cannot be unlinked.

The trace package turns this switch on when a function is traced, in order to ensure that tracing will work even for compiled functions. Compiled function calls which have been "snapped" to go directly do not push debugging information in *rset mode and cannot be traced. See also (status uuolinks).

baktrace LSUBR 0 to 2 args

baktrace displays the stack of pending function calls. It gives detailed information only 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. (The second argument is currently permitted only in the Multics implementation.) The information printed by baktrace is not the same as that obtained with evalframe; both should be used to get the maximum amount of debugging information.

baklist LSUBR 0 to 1 arg

baklist returns a list containing the information which baktrace would print. (This is available only on the PDP-10 implementations.)

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 (err *pdlptr message bcp*), where *pdlptr* is a number which describes the location in the pdl of the error, *message* is a list of from one to three things which, given to the error function, could have caused this same error, and *bcp* (binding context pointer) 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 pointer, in which case the stack is searched downward from the indicated position. Thus if there are recursive calls to the error handler, the second error back down the stack may be found by:

```
(errframe (cadr (errframe nil)))
```

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

If no error is found, errframe returns nil.

`errprint`　　LSUBR 1 to 2 args

`errprint` treats its argument the same as `errframe`. The second argument, in Newio implementation only, is the file(s) into which to print the information (see `print`). 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` pointer, 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 (*type pdlptr form bcp*), where *type* is `eval` or `apply`, *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 a list of the name of the function being applied and the arguments it was applied to, and *bcp* is a binding context 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, since no extra frame information is saved otherwise.

`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.

`evalhook`　　VARIABLE

If the value of `evalhook` is non-null, then special things happen in the evaluator. When a form (even an atom) is to be evaluated, `evalhook` is bound to `nil` and the functional form which was its value is applied to one argument - the form that was trying to be evaluated. The value it returns is then returned from the evaluator. This feature is used by the Stepper package described later in this section.

evalhook is bound to nil by break, and setq'ed to nil by errors that go back to top level and print *. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the LISP interpreter, several restrictions are imposed on evalhook. It only applies to evaluation - whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function eval. It *does not* have any effect on compiled function references, on use of the function apply, or on the "mapping" functions. Also, as a special case, the array reference which is the first argument to store is never seen by the evalhook function; however, the subexpressions of the array reference (the indices) will be seen. (This special treatment avoids a problem with the way store works.) Normally the evaluator does not check the value of evalhook, in order to save time. To make it check, you must both be in (*rset t) - debugging - mode, and have done (sstatus evalhook t). Not all implementations need both of those, but you should always do both to be sure. If you use the Stepper package, you need not worry; it does this automatically.

evalhook LSUBR 2 or 3 args

(evalhook *form hook*) is a function which helps exploit the evalhook feature. The *form* is evaluated with evalhook lambda-bound to the functional form hook. The checking of evalhook is bypassed in the evaluation of *form* itself, but not in any subsidiary evaluations, for instance of arguments in the *form*. This is like a "one-instruction proceed" in a machine-language debugger. If all three arguments are present, the second is a binding context pointer and is used as the second argument to eval, and the third argument is the hook.

Example:

```
(defun hook fexpr (x)           ;called as (hook <form>)
       ((lambda (*rset)
                (prog2 (sstatus evalhook t)
                                 ;magic sstatus
                       (evalhook (car x) 'hook-function)
                                 ;evaluate form
                       (sstatus evalhook nil)))
                                 ;more magic
        t))

(defun hook-function (f)
       ((lambda (terpri)
                (declare (special terpri))
                (princ '|form: | msgfiles)
                (prinl f msgfiles)
                ((lambda (v)
                         (terpri msgfiles)
                         (princ '|value: | msgfiles)
                         (prinl v msgfiles)
                         v)
                 (evalhook f 'hook-function)))
        t))
                                 ;this is how to eval the
                                 ; form so as to hook
                                 ; sub-forms
```

The following output might be seen from (hook (cons (car '(a . b)) 'c):

```
form: (cons (car (quote (a . b))) (quote c))
form: (car (quote (a . b)))
form: (quote (a . b))
value: (a . b)
value: a
form: (quote c)
value: c
value: (a . c)
(a . c)
```

The following functions only exist in the Multics implementation.

`baktrace1`   `LSUBR 0 to 2 args`

`baktrace1` is the same as `baktrace` except that binding context pointers 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 binding context pointers.

1.5.3 The Trace Package

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`. (See `autoload`.)

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 option1 option2 ...*)

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*. The user 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. (Then again, it might.) Note that *fn* may not be a compiled function.
`argpdl` *pdl*	specifies an atom *pdl* whose value `trace` initially sets to `nil`. A list of the current recursion level for the function, the function's name, and a list of the arguments is cons'ed 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 it 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`, and `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.

The trace specifications may be "factored." For example,

```
(trace ((foo bar) wherein baz value))
```

is equivalent to

```
(trace (foo wherein baz value) (bar wherein baz value))
```

All output printed by trace can be ground into an indented, readable format, by simply setting the variable sprinter to t. Setting sprinter to nil changes the output back to use the ordinary print function, which is faster and uses less storage but is less readable for large list structures.

Examples of the use of 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))
             both zxcvbnm (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.

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 is an error message. The error messages are:

(? 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 use of 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) evaluates to a list of all the 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 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.

1.5.4 The Stepper

The LISP "stepping" package is intended to give the LISP programmer a facility analogous to the Instruction Step mode of running a machine language program.

The Rich Stepper

The Rich stepper package provides a simple, small debugging capability. It is available as a library program in the ITS implementation.

How to Use the STEP Facility

The package contains two compiled functions which are loaded by

```
(fasload step fasl dsk liblsp)
```

The user interface is through the function (fexpr) step, which sets switches to put the LISP interpreter in and out of "stepping" mode. The basic commands are:

```
(step t)          ;Turn on stepping mode.
(step nil)        ;Turn off stepping mode.
```

These commands are usually typed at top level, and will take effect immediately (i.e. the next S-expression typed in will be evaluated in stepping mode). Also ^g, in addition to returning to top level, turns off stepping mode.

In stepping mode, the LISP evaluator will print out each S-expression to be evaluated before evaluation, and the returned value after evaluation, calling itself recursively to display the stepped evaulation of each argument, if the S-expression is a function call. After displaying each S-expression, the evaluator will wait before evaluation for a command character from the console:

<space>	Continue stepping recursively.
<rubout>	Show returned value from this level only, and continue stepping upward.
<cr> or <tab>	Turn off stepping mode. (but continue evaluation without stepping).

P	Redisplay current form in full (i.e. without `prinlevel` or `prinlength`)
B	Get breakpoint; proceed with $P
M	See advanced features under Stepping Macro Expansions.

More Advanced Features

Selectively turning on `step`:

```
(step foo1 foo2 ...)
```

If this command is typed at top level, stepping will not commence immediately, but rather when the evaluator first encounters a S-expression whose car is one of `foo1`, `foo2`, etc. This form will then display at the console, and the evaluator will be in stepping mode waiting for a command character.

Stepping Macro Expansions:

If the stepper is proceeded with a <space>, it will not step the execution of macro expansions, but will rather just show the result of the macro of expansion and wait for another command.

To see the execution of the macro expansion itself, proceed the stepper with an M instead of a <space>.

Using `step` with breakpoints:

The above description applies to turning stepping on and off globally at top level. More detail is necessary to use `step` flexibly in and out of breakpoints (e.g. together with `trace`).

If stepping is turned on by `(step t)` at top level, the evaluator will NOT be in stepping mode within a breakpoint loop. If you wish to use stepped evaluation within a break loop you must turn it on locally by `(step t)`. Conversely, if stepping was not turned on at top level and it is turned on by `(step t)` in a break loop, it will NOT be on when return is made from the break loop by $P.

However, executing `(step nil)` inside a break loop will turn off stepping globally, i.e. within the break loop, and after return has be made by $P.

The most useful feature is the following, however:

`(step)`	Command at top level has no immediate effect.

After `(step)` has been executed at top level, a subsequent `(step t)` inside of a break loop will have the effect of turning on stepping mode both inside the break loop and globally, i.e. the evaluator will start to step as soon as the return is made from the break loop by $P. Thus, for instance, one could set `trace` to break at some special place, and then use the break to turn on stepping.

`prinlevel` and `prinlength`:

In the present version, for convenience, `prinlevel` and `prinlength` are `lambda`-bound inside the hooking function to 3 and 5 respectively. These could be changed by editing the expr code and recompiling.

When the P command is used, `prinlevel` and `prinlength` are temporarily bound to `nil`, and the toplevel printer (the value of atom `prinl`) is used to redisplay the current form.

Overhead of Stepping:

If stepping mode has been turned off by ^g, the execution overhead of having the stepping packing in your LISP is exactly `nil`.

If stepping mode has been turned off by `(step nil`, every call to `eval` incurs a small overhead--several machine instructions, corresponding to the compiled code for a simple cond and one function pushdown.

From an overhead point of view, running with `(step)` entered at top level is the same as running with `(step nil)`.

Stopping stepping by responding <tab> incurs the same continued overhead as `(step nil)`.

Running with `(step fool foo2 ...)` can be more expensive, since a member of the car of the current form into the list `(fool foo2 ...)` is required at each call to `eval`.

In terms of memory requirements, the total compiled stepping package occupies about 423 words of binary program storage.

Interaction with debug and trace:

No special interactions of the **step** package with **debug**, **trace**, or any other system packages are known.

The Morgenstern Stepper

The Morgenstern Stepper package provides debugging capabilities for interpreted LISP programs that are comparable to the capabilities provided by DDT for assembler code. These capabilities include:

1) Single stepping through the evaluation of a function and over or into other interpreted functions, when called, on a selective basis as determined by the user. Each such form and its resulting value may be displayed.

2) Dynamic breakpointing on one or more of the following conditions: the form or atom about to be evaluated matches a pattern you provide; the form being evaluated involves a specified function; a given atomic symbol evaluates to a given value; a given atomic symbol is to be bound in a **prog**, either type of **do**, or an **eval'd** lambda-expression; or upon an arbitrary condition specified by a predicate written as LISP code.

3) Returning a different value for a given S-expression. This allows for changing the action that would be selected by conditionals in the program and/or by **go**'s in a **prog** or **do**. You can also go to any tag inside the current **prog**.

4) These capabilities may be requested when the program is initially started by a top-level form, or they may be initiated at any other point in the course of execution - either from the terminal while in a breakpoint, or directly by the program.

The stepper may be invoked initially by using the function **mev** as one would use eval of one argument; e.g. (mev '(fcn argl arg2)). From a breakpoint or in a program, the stepper may be turned on by invoking (hkstart) with no arguments. It may be turned off by the **q** command described below, or of course by a CTRL/g break. After mev evaluates its argument, it returns the value and turns off the stepper. Note that in the above example the form given as an argument to mev was quoted. If, say, the value of f was the S-expression (fcn argl arg2), then one could use (mev f) instead.

At any point during the stepping, one may inspect the values of other variables, and even reapply mev to any form. This may be done in either of three ways. Each command will be prompted for by //, usually following the last form printed out. Any S-expression that is typed which is not recognized as a command will be eval'd (within an errset to catch errors). Alternatively, the e command to eval any expression, or the h command to get a nice type of CTRL/h break. (This is really a CTRL/b break, but it used to be CTRL/h so the command happens to be called h.)

in the its implementation each command must be followed by a space (unless the command is a list). in the multics implementation each command must be followed by a newline. actually, this depends not on the implementation but on **(status linmode)**. each form and result which is printed out will be followed by *#number* indicating the relative level of evaluation (i.e. stack depth since invocation).

the primary commands are:

d (mnemonic for down) go down to the next deeper level of evaluation and display the first form there before evaluating it. e.g. if the form is a function call, this will display the first argument of the function if it has arguments in the call; otherwise it will display the first s-expression of the body of the function. it then prompts for the next command.

e (eval) can be used to evaluate an arbitrary expression. it starts a new line, waits for you to type the expression, then eval's it within an **errset**, and prints the result. this is comparable to just typing the expression or atom after the //, but cannot be confused with a command, and the format is nicer.

h (control-h) enters a break loop, and when $p'ed displays the current form. within the break, one can inspect the values of variables, etc., and even **reapply mev** to any form.

n (next) Display the next form at this level, without showing or inspecting the evaluation of the lower levels of the current form. The value of the current form is displayed first. If you wish a condition to be tested for at lower levels, use **nn** instead.

nn Like n but slower since it inspects the lower levels. Use instead of n when testing for a condition.

u (up) Go up to the next higher level of evaluation and show the next form at that level. The form(s) at the current and lower levels are evaluated without display. As an example of its use, after you have seen the evaluation of the arguments to a function, the next form to be evaluated, if the function is being interpreted, will be the first S-expression of the function; to avoid seeing how the function is evaluated internally, you can type u. Note that the lower levels are not inspected - thus if a condition is to be tested for at these levels, use uu.

(u *num*) If *num* is positive (or zero), forms are not inspected nor displayed until level *num* is returned to. If negative, it goes up (abs *num*) levels relative to the current level. Thus (u -1) is equivalent to u.

uu Like u but slower. Use if testing for a condition.

(uu *num*) Like (u *num*) but slower. Use if testing for a condition.

q (quit) Exit from the stepper.

s (show or display mode) For datapoints and other display terminals, this gives a nice easily read output of selected levels that constitute the context of the current evaluation. Specifically, it selects the current level for sprint'ing (pretty printing) as a "header", and as you go deeper, the local context is abbreviate-printed under this header, and the current output will be sprint'ed. s may be used as often as you like. Headers will automatically be popped when you return. The command (s *num*) selects a particular level as a header. It and the command sn and several user settable parameters are described in the more detailed section below.

(= *s-exp*) The S-expression is substituted for the current form and another command is prompted for (i.e. you can step into or over the new form if you want to). When the resulting value is returned it will be as if the original form had yielded that value. For example, you can change the apparent truth or falsity of predicates or bypass a (go *label*), as well as just returning different values for an S-expression.

(cond ...) Tests for conditions prior to evaluation of each future form, and when satisfied will print a message, display the form, and wait for another command (which may of course be h for a break). The argument to this cond is an arbitrary S-expression or symbol which is evaluated like a predicate. This is similar to the cond feature of the trace package.

In specifying the predicate, the form about to be evaluated may be obtained as the value of the variable %%form. The expression (hooklevel) returns the relative level of evaluation. More than one predicate may be given, in which case they are or'ed together, except when two arguments form a special test as described in the more detailed section below. The condition will remain active at all levels that are inspected by the stepper until explicitly turned off by (cond nil).

(matchf ...) is a function which will pattern match against the current form. It may be used in the predicate of the cond. (Also see its related use as a command.) The argument to matchf is compared to %%form element by element from left to right, and succeeds when each element of the pattern succeeds. Of importance, the pattern need not include the entire form. * matches anything. The procedure is applied recursively to sublists, unless the sublist is of the form (# ...) in which case # is bound to the current element of %%form and the cdr (not cadr) of the #-list is evaluated as the test on that element. Except in this case, atoms and lists should be given as in the original code since they are not evaluated. Some simple examples are:

(matchf xyz) succeeds if the atom xyz is about to be evaluated.

(matchf (setq alpha)) succeeds if the atom alpha is about to be setq'd.

(matchf (putprop name * ′source)) succeeds if the property source is about to be putprop'd on the atom pointed to by (i.e. the value of) name.

(matchf (setq (# member # ′(alpha beta s3)))) succeeds if either alpha, beta, or s3 is about to be setq'd.

(matchf (rplacd * ′(* 9))) matches (rplacd (last urlist) ′(2 9 4)).

(matchf ((# member # ′(foo bar)))) succeeds if a function call to either foo or bar is about to be evaluated (more precisely if the car of the form about to be evaluated is either foo or bar).

nil (cond nil) turns the condition off and saves the current non-nil condition.

(cond) When no argument is given, the last non-nil condition (which is the old property of %%cond) is established as the current condition (which is the value of %%cond). (If the previous condition was not nil then it is saved as the old property, thus allowing for alternation of two conditions.)

(matchf ...) is equivalent to (cond (matchf ...)), see above.

The following functions are useful in connection with the stepper.

(hkstart) will initiate stepping when encountered in a program or typed from a breakpoint. (hkstop) will act like the q command to turn off stepping. (Also see below for more info.)

(mbak) is a function to be used like the LISP system's (baklist). (mbak) strips out from the result of (baklist) those functions that have to do with the stepper.

The remainder of this section is a complete list of the Stepper commands, which can be used for reference.

Commands which are not lists must be followed by a space. You can use rubout before completing the command (and its space if necessary). Alternatively, you may abort the command before completing it by doing a CTRL/x break.

Any S-expression that you type which is not recognized as a command will be evaluated (within an errset to catch errors). Thus you can evaluate any atom or do any function call simply by typing it following the prompting // as long as it is not interpretable as one of the commands below (or nil). Note that you can actually go to a *tag* within your prog simply by typing (go *tag*) after the //. To evaluate a form which looks like a command, type (or *form*) to evaluate it, e.g. (or a) evaluates the atom a. If you want you can even write functions which know about the stepper and treat them as commands.

a (all) Automatically displays all forms and values seen by the stepper at all levels. Typing a space at any time thereafter will cause the stepper to leave this mode and prompt for a new command. If you want the stepper to wait for a command after each form, you can use the d command.

Commands a ad (a -) c and cc pause after each new form is displayed if %%ac-sleep is non-nil. Its value is used as the sleep time in seconds.

ad (all down) Automatically displays all forms and values encountered by the stepper in evaluating the current form (i.e. at deeper levels). Typing a space prior to completion will cause the stepper to leave this mode and prompt for a new command. (Also see d.) Sleeps after each form, as described under the a command.

(a *lev*) Automatically displays all forms and values at the indicated level and lower (deeper) levels, turning itself off when evaluation pops to a level with a smaller level number. Typing a space prior to completion will cause the stepper to leave this mode and prompt for a new command. (Also see d.) Sleeps after each form, as described under the a command.

b Sets a breakpoint to occur after evaluation of the current form. At the break, the value to be returned is the value of %%value, and may be changed by setq'ing this variable. The form that yielded this value is the value of %%form. Type $P to proceed from the breakpoint. If you prefer that the system wait rather than break see the `wtif` command. (b operates by adding the current hooklevel to `%%breaklist`.) You can get automatic breaking at all levels by using **`(retcond t)`** or conditional breaking as described below for the **`(retcond ...)`** command.

c (current) Automatically displays all forms and values at just the current level. Typing a space at any time during the display will cause the stepper to leave this mode and prompt for a new command. The stepper does not inspect the forms of lower levels - thus if a condition is to be tested for at these levels, use cc. Sleeps after each form, as described under the a command.

cc Like c, but inspects the lower levels.

`ctog` Flips the `%%condnotallow` toggle which is initially t, meaning do not allow c, m, n, or u commands if a condition is being tested for. `nil` means allow these anyway.

`(cond ...)` Tests for conditions prior to evaluation of each future form. For pattern matching against the form using the `matchf` function and for other information see the description of `(cond ...)` above.

special tests for `(cond ...)`:
To aid the specification of common tests, the following "flags" are provided - the same effects could be obtained by inspecting `%%form` in your own predicate given to cond. If the first argument to the cond is from the set **`(form formq bind bindq atomval atomvalq fcn fcnq and andq)`** then the second argument is used to derive a test. This process is repeated with the remaining arguments, if any. The resulting tests, together with any remaining arguments not satisfying this process, are effectively or'ed together to derive the overall condition (except for the `and` `andq` *flag* special tests which are and'ed). The arguments are not evaluated when typed but are evaluated each time the condition is tested. These flags each may be used more than once.

The meanings of these flags are:

`andq`	The next argument is and'ed with the remaining tests, and must yield a non-`nil` value for the remainder of the condition to succeed. (See the comments for cond in the "complete list of commands" below regarding the use of side effects)
`atomvalq`	The next argument should be a list of two elements, the first an

(unquoted) name of an atom, and the second the value of this atom for the test to succeed.

`bindq` Watch for the following (unquoted) atomic symbol to be bound in a prog, or in either type of do, or an explicitly evaluated `lambda` (as distinct from an applied `lambda` or function call).

`fcnq` Watch for the following (unquoted) function name to be seen by `eval` as the car of the form about to be evaluated. (This cannot check for applied or mapped function calls).

`formq` The following (unquoted) S-expression is to be watched for. E.g. used to check when a particular variable is about to be evaluated.

`and`
`bind`
`fcn`
`form`
`atomval` These evaluate their argument each time the condition is tested in order to get the desired S-expression or atom name, and then perform like their "q" counterparts. These are particularly useful if the flag's argument is the value of a variable. (Be sure not to change the variable's value accidentally while the condition remains in effect.)

As a simple example, `(cond fcnq rplacd)` will check and stop when the function `rplacd` is about to be used (i.e. when it is the the car of the form to be evaluated).

The commands c, m, n, and u do not inspect all levels, and thus the condition cannot be tested for at these levels. You can use cc, nn, mm, or uu instead, or use the `ctog` command. Naturally, condition testing slows the speed of execution at levels that are inspected by the stepper but which you do not have displayed.

If you choose to, you can have your predicates produce side-effects such as recording information of value to you or setting states for use by the condition later. You can use the `and`, andq flags (more than once if you like) to have the expressions executed even upon success, so long as these flags appear first in the condition. Other conditions are evaluated in the order of appearance until the first success is found.

d (down) Displays the next level down. (as described above also). Note that if the form is an atom, the effect is the same as the n command. Hence if you want the stepper to display every form and value, but to wait for a command after each form, just keep using the d command.

e (eval) Can be used to evaluate an arbitrary S-expression. It starts a new line, waits for you to type the expression, evaluates it within an **errset**, and prints the result. Comparable to just typing the expression or atom after the //, but cannot be confused with a command, and the format is nicer.

(= *S-exp*) Replaces the current form with the given S-expression, and then prompts for another command, as described above. If two arguments are given, then this expression will not be treated as a stepper command, rather it will be evaluated (see comments at top of this section).

h CTRL/h break is executed. The current form is redisplayed when $P is typed. The form about to be evaluated is the value of %%form. Within the break, one can inspect the values of variables, etc., and even reapply mev to any form.

k (kill) Does not evaluate the current form nor display any value. This is good for avoiding side effects if restepping through a program again. Equivalent to (= nil) followed by m command.

lr (last result) A complete rather than abbreviated printout of the last result is given. (See (p - -) for further information.)

m Next, like n but the result of the current form is not displayed. If a condition is to be tested for at lower levels, use mm.

(matchf ...) is equivalent to (cond (matchf ...)), see the the description of (cond ...) above.

mm Next, like nn but the result of the current form is not displayed.

n (next) Displays the value of the current form and displays the next form, then awaits the next command. Does not inspect the lower levels. If a condition is to be tested for at lower levels, use nn instead.

nn Like n but inspects the lower levels.

o (old) Does (mev *'last form*). This is useful for seeing how a form produced an unexpected value when you went over it with n or nn. If reevaluating the form can produce side effects be careful. Can be exited from by the xx command. (The old form is the value of %%oldform.)

ol (old, at current level) Does (mev *'last form at this level*). Behaves like o. Useful to see the form (at this level) which produced the current value - rather than the last form printed out, as o would yield. (The old form used here can be obtained by (get %%hooklevel 'oldform).)

po(in)(print) Redisplays the current form. This is useful if you wish to clear the screen first with control-L. Gives typical abbreviated display (see (p - -)), except has somewhat different effect if in display mode (see s command). (For hackers of special data structures, e.g. "owl", printing will be done with the function which is the value of the atom prinl if non-nil - as also applies to top-level in LISP. This value of prinl is checked only in the mev function. Moreover, unless you request LISP not to "snap links" in compiled code, you may have to reload the stepper after changing prinl.

pp (full print) Gives a complete printout of the current form.

ppp (even better printout) Pretty-prints the current form using the sprint function. Uses a lot of screen in general, and so will turn on pagepause for you.

(p - -) Resets the parameters for the abbreviated printout used for results, forms and the p command. The first parameter is the prinlevel, the second is the prinlength; both must be given. If nil is given instead of a number no abbreviating is done with respect to that parameter; thus (p nil nil) turns off abbreviation. (The current settings are the value of %%hookprin.)

q (quit) Exits from the stepper. Previously requested breaks and conditions are disabled, and any non-nil conditions are saved on the old property of the condition name. (Control-G also exits as usual.)

s (show or display mode) For datapoints and other display terminals, this gives a nice easily read output of selected levels that constitute the context of the current evaluation. Specifically, it selects the current level for sprinting as a "header", and as you go deeper, the local context is abbreviate-printed under this header, and the current output will be sprinted. s may be used as often as you like. Headers will automatically be popped when you return. All sprinting is done with pagepause on. If control-X is typed during sprinting, that expression will be redisplayed using abbreviated-printing instead. When in this display mode, the **p** command will clear the screen from the last form down, unless preceeded by control-L (or if wrap-around occurred), in which case the screen is fully redisplayed. Also see `(s` *arg*`)` for more information and options.

`(s` *arg*`)` If *arg* is positive, this selects the form at that level as the "header" for s(how) mode. If negative, it uses the form at *arg* levels above the current one. If *arg* is `nil`, display mode is turned off (headers are remembered though). **`(s t)`** just turns display mode on if currently off using the previously remembered headers if still applicable; but if it is already on, this pops the stack of headers by one (normally headers are automatically popped when the level is returned from). All sprinting is done with pagepause on. If control-X is typed during sprinting, that expression will be redisplayed using abbreviated-printing instead. Also see the **sn** command.

Several parameters are user settable from their defaults. **`%%lowerdisplay`** and `%%lowerdisplay-min` control the maximum and minimum number of levels to display below the header (defaults of 5 and 2). This is done in abbreviate-printed form using `%%shortprin` which is a list of the prinlevel and prinlength (defaults 3 and 3). Sprinting of forms and results will be abbreviate-sprinted by the `msprint` function if the `flatsize` of the expression exceeds **`%%flatsize-max`** (default about 450). The prinlevel and prinlength for the latter are the list which is the value of `%%sprintabbr` (default is (7 8)). If **`%%flatsize`** is **`nil`**, full sprinting will always be used; (if negative, abbreviate-sprinting will always be used so that infinite printing circular structures will sprint and abbreviate-print finitely. To turn off sprinting of results setq **`%%result-sprint`** to **`nil`** (default t). If `%%mdistitle` is neither `nil` nor a number, it will be evaluated just after the screen is cleared, allowing printing of a title. If it is a number, that number of blank lines will be left at the top of the screen (also see **sviewmsg** function below). If the partial clearing of the screen bothers your eyes, **setq**'ing `%%eyestrainl` to a number of seconds (e.g. 0.5 to 2.0) will slow down the new display depending on the number of lines cleared.

`sn` Just for s(how) display mode. It prevents clearing of the screen after prompting for another command, but only until the next prompting // after that. Useful if you want a result to remain displayed a little longer. If you want to prevent clearing of the screen for more than a couple of times, use **`(s nil)`**, then do **`(s t)`** when you want to resume display mode.

`(retcond ...)` Tests for conditions just after each form is evaluated, and breaks when such condition is satisfied. At the break, the value to be returned is the value of %%value, and may be changed by setq'ing this variable. The form that yielded this value is the value of %%form. Type $P to proceed from the breakpoint. The conditions are specified as for (cond ...). Note that **(retcond t) will** give you a break as each level is popped (returned from), including levels above the one where the request was made. (retcond nil) disables the retcond. If you prefer waiting rather than breaking see the wtif command.

Two additional flags are available:

valueq The test (equal %%value *next-argument*) is performed as if it were *and*'ed with the remaining predicates in the condition.

value Like valueq but the test is (equal %%value (eval *next-argument*)). The overall condition is maintained on the value of the atom **%%retcond**, and the previous non-nil condition is on the old property of this atom. If you want both cond and retcond conditions to be the same you can **(setq %%retcond %%cond)**. The value and valueq predicates will be ignored in a (cond ...).

u (up) Go up to next higher level. Current and lower levels are executed without display. The lower levels are not inspected - thus if a condition is to be tested for at these levels, use uu. This can be used to skip the display of a function's internal evaluation after having seen the arguments, as described in the previous section.

(u *num*) If *num* is positive (or zero), forms are not inspected nor displayed until that level number is reached. If negative, it goes up this number (absolute value) of levels relative to the current level. Thus (u -1) is equivalent to u .

uu Like u, but also inspects lower levels. Use if you have a condition to be tested.

(uu *num*) Like (u *num*) but slower. Use if testing for a condition. Note that (uu -999) effectively means that you won't see any levels unless the condition in a cond or retcond is satisfied.

wtal (wait-all) Flips a toggle which when on causes a pause after the evaluation of every form, but before that value is returned. The system waits for an input character. Typing y(es), b(reak), or h (for control-h) followed by space will cause a break as would the b command. Typing just a space, or any other character followed by a space, will proceed from the pause. Default is off.

wtif (wait-if) Flips a toggle which when on causes requests by the b and (retcond ...) commands to result in a pause rather than a break. The pause is like that of the wtal command, and may be proceeded by a space; or a break initiated by typing y, b, or h followed by a space. Default is off.

xx Does a control-X type of LISP quit. (A control-X typed after the // prompt will be caught by an errset. The xx command is executed outside of that errset.)

The following other facilities exist:

(gethklevel *num*) This function returns the S-expression that is on the execution stack of the stepper at the given level number (see hkshow). Can be used to get an unsprinted unabbreviated display of the form or to record or process the form as you desire, including reapplication of mev to it in the current context.

(hkshow *num*) This function will display previous forms which are on the execution stack, as seen by the stepper while it has been activated. The previous *num* of levels are shown, with the current form last. If no argument is given, then all levels are shown. The display is done under the control of prinlevel and prinlength which are settable by the (p - -) command. Of course this function can also be used as if it were a command by typing it after the prompting //.

(hksprint *num*) This function will sprint the form on the level whose number is given as the argument. Can also be used as a command.

(hkstart) Use this function to invoke or reinvoke the stepper from a breakpoint or from a program as described above. If used within a break, type (hkstart) by itself rather than within another S-expression or function, as it has to climb the stack from the point of invocation. If an argument is given to this fexpr, it will be evaluated just prior to establishing stepping, with ^w bound to nil, so that you can print out information if called from a program.

(It is possible for the invocation of the stepper by this method to have limited scope under some circumstances. Such a boundary would be a second breakpoint higher on the stack or a previously terminated invocation of the stepper that is still on the stack. Also if the program was initially started without mev, and stepping is retained thoughout the rest of the execution, stepping may also remain for forms typed at top level - to stop this just do control-G (or use the q command) .)

(hkstop) This function turns off the stepper whenever executed - in the same manner as the q command would.

hooklist is an atom whose value is inspected before each attempt to read a command from the console. If hooklist is non-nil, it is assumed to be a list of commands to the stepper - each is printed out when used and treated as if it came from your typein. `hooklist` is also examined at each level that is inspected by the stepper even if no command reading is done (e.g. nn or uu modes).

`(mbak)` This function gives `(baklist)` but without the stepper functions, as described above.

`(mev` *top-form*`)` This function initiates stepping and otherwise acts like `eval` of one argument, as described above.

`(msprint` *form*`)` Gives abbreviated sprinting of the *form*. A second and third numeric argument specify the effective prinlevel and prinlength here, else a list of two numbers found as the value of *msprint* are used. The current implementation is somewhat slow as the regular sprint does not respond to standard abbreviating.

`(sviewmsg` *lineno toeval*`)` Useful in conjunction with s(how) mode. Puts the cursor at the *lineno* and evaluates the second argument, then returns the cursor to its original position. *lineno* = 0 means top; if negative counts from bottom, with -1 the bottom line. Typically have `%%mdistitle` (see `(s -)` command) be a number to skip lines on top, and use `sviewmsg` to display your debugging information up there.

If you really want specialized processing in particular situations, you can inspect and/or change `%%form` in a `(cond ...)` predicate, and `%%value` in a `(retcond ...)`. If `%%nohookflag` is `t`, form and value printout and command reading (except from a non-`nil` hooklist) is inhibited until it is reset to `nil`. Normal command processing is invoked by `(%%mhookcom)` with `%%nohookflag` bound to `nil`. Also described above are `%%breaklist`, `%%cond`, `%%retcond`, and `%%hookprin`.

1.5.5 The MAR Break Feature

This feature is currently available only in the ITS implementation.

The MAR break feature takes advantage of a hardware feature (the Memory Address Register break) which interrupts whenever a given memory location is accessed in a specified way. It allows the LISP user to specify an interrupt function to run whenever a variable or list cell is modified. The user must first "arm" the interrupt by saying `(sstatus mar` *cond loc*`)`. *cond* is the condition on which to interrupt:

0	Turn off the mar feature.
1	Interrupt on instruction fetch.

2	Interrupt on write (modification).
3	Interrupt on all references.
	(numbers are octal)

On a KL-10 processor, additional conditions are available:

10	Interrupt on data read.
11	Interrupt on data read or instruction fetch.
12	Interrupt on data read or write.
13	Interrupt on instruction fetch or write.

loc is any s-expression; that cell is the one monitored.

Example:

```
(setq foo (list 'a 'b))
(sstatus mar 2 foo)
```

will interrupt if the list cell in foo is ever rplaca'd or rplacd'd.

An example of the use of the mar-break interrupt:

```
(defun mar-tracer (x)
      ((lambda (val)
               (sstatus mar 2
                        (get the-mar-variable 'value))
               (nointerrupt nil)        ;let endpagfn interrupts in
         (terpri msgfiles)
         (princ '|Now the variable | msgfiles)
         (prinl the-mar-variable msgfiles)
         (princ '| has the value| msgfiles)
         (prinl val msgfiles))
     (symeval the-mar-variable)))

(setq mar-break mar-tracer)

(defun mar fexpr (x)
     (cond ((null x)(sstatus mar 0 nil))
          (t (setq the-mar-variable (car x))
                         ;make sure the variable has a value cell
                (or (boundp the-mar-variable)
                    (set the-mar-variable nil))
                (sstatus mar 2 (get the-mar-variable 'value)))))
(mar quux)
(setq quux 5)
;Now the variable quux has the value 5
5
(do ((quux 0 (+ quux 1))) ((= quux 2))
    (hack quux))
;Now the variable quux has the value 0
;Now the variable quux has the value 1
;Now the variable quux has the value 2
;Now the variable quux has the value 5
nil
```

Notice that quux is altered by the do loop, and also by the restoration of the old value 5. This example is for a KA-10 processor. On a KA-10, the user interrupt is always run after the location has been changed. On a KL-10 processor, the interrupt occurs just *before* a modification or access rather than just after.

The mar break feature is sometimes used by DDT to debug the LISP system. As long as DDT and LISP do not both try to use the mar break feature on the same LISP at the same time, there should be no problem. **(sstatus mar 0 nil)** releases the mar break feature entirely for use by DDT.

The suspend function will attempt to save and restore the state of the mar break feature. If you don't want an armed mar break to persist beyond a call to suspend, turn it off first with `(sstatus mar 0 nil)`.

When a CTRL/g quit (or the ^g function) forces a quit back to top level, it disables the mar break before unwinding variable bindings and re-enables it afterwards. This is because during a CTRL/g quit LISP may not be in a good state for running user interrupt functions.

1.6 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.

1.6.1 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 internal 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 for creating new S-expressions.

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; the decimal values are 0, 1, 8, 9.)

^d SWITCH

If the value of ^d is non-nil, the garbage collector prints an informative message each time garbage collection occurs. (In a Newio implementation, this message is output to the files in msgfiles, see page 3-15) In the PDP-10 implementation, it also prints a message when a space is expanded without first doing a complete garbage collection, or when a file or inferior job is closed because a file object was garbage collected. See also (status gcwho).

See also the user interrupts gc-daemon, gc-overflow, and gc-lossage.

1.6.2 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 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.
HUNK4	Hunks of various sizes. PDP-10 implementations without hunks do not have these spaces.
HUNK8	
HUNK16	
...	
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, pure HUNG4, ...
These spaces are used to store "pure" (read-only) data of the indicated types. This is a feature used to make subsystems more efficient. See page 3-67.

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.

The precise spaces available in a given implementation can be determined by using `(status spcnames)`, `(status purspcnames)`, and `(status pdlnames)`.

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 recursion.
`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 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 collection 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 may 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 collection. 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 PDP-10 implementation presently allocates memory to spaces in blocks of 512. words. The Multics implementation allocates at least 16384. words between garbage collections and presently controls the size of pushdown lists in blocks of 16. words.

Some spaces, such as Binary Program Space in the PDP-10 implementation or Static storage 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.

1.6.3 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 pdlmax (for a pdl space) or the gcsize and gcmax (for other spaces.) A 3-list 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. A 3-list cannot be used with a pdl space.

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

```
(alloc '(list (30000. 5000. 0.25)
         fixnum (4000. 7000. nil)
         symbol 6000.
         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.

See page 3-87 for some status functions which are related to the topic of storage spaces.

1.6.4 Dynamic Space and Pdl Expansion

There are several user interrupts generated by the storage management. See section page 3-16 for a description of user interrupts. The gc-daemon interrupt occurs after each garbage collection. The argument passed to the gc-daemon interrupt handler is a list of spaces and their sizes. In the PDP-10 implementation, the items on the list are of the form: (*space-name free-before free-after size-before size-after*), where *space* is the name of a space, and *free-before* indicates the number of cells free before the garbage collection and *free-after* indicates the number of cells free afterwards. The last two numbers are the size of the space (see (status spcsize)) before and after the GC. (The sizes are in PDP-10 words.) In the Multics implementation, the items are of the form: (*space before* . *after*). 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 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. If the interrupt handler returns, the value is ignored, and another garbage collection is attempted.

The pdl-overflow interrupt 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, for example by doing (^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 occurs when some space (other than a pdl) exceeds its gcmax. This gives the user a chance to decide that 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 in that if the interrupt handler function returns at all, even if it returns nil, the interrupted computation proceeds. To terminate the computation an explicit (^g) or throw must be done.

1.6.5 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 the DEC-10 implementations, which cannot 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 for most purposes. ("?" causes a list of suitable responses to be printed out.) "N" means that you do not want to specify allocation. You will get the default. CTRL/q means to read your initialization file (see below.) "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. If the name of the space is preceded by "*", then it cannot be expanded once allocated by this dialogue. 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. CTRL/g restarts the dialogue with the "ALLOC?" question.

If you reply with a CTRL/q, it means to read your initialization file. In the ITS implementation, this is either *udir*; .LISP. (INIT) or (INIT); *udir* .LISP., where *udir* is your master sname. In the TOPS-10 implementation, this is LISP.INI in the directory you are logged in to. In the Multics implementation, this is *hd*>start_up.LISP, where *hd* is your home directory. Since the Multics implementation doesn't have the allocation dialogue, this file is always read when the LISP command is given with no arguments.

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 different implementations. An example of the form of this comment is:

```
(comment fixnum 5000 symbol 4000 flonum 2000
         bignum 1400 regpdl 5000 hunk8 30000)
```

The remaining forms in the file are simply read and evaluated using the standard read-eval-print loop.

1.7 Implementing Subsystems with Maclisp

1.7.1 Entering LISP

A subsystem is an entity that exists in most time-sharing systems. It is normally a complete world which the user enters by typing a command. He then has whatever facilities the subsystem offers. A subsystem can be oriented toward programming, as the Maclisp subsystem itself is, or it can be oriented toward a particular application, for instance compiling LISP programs, operating machinery, or solving differential equations.

Maclisp is frequently used as a base on which to build subsystems. Consequently it has been equipped with a number of mechanisms which allow the subsystem writer to gain complete control over the operation of Maclisp, make it possible to hide the vagaries of Maclisp from the naive user of a different subsystem, and provide increased efficiency in memory and processor usage for heavily-used subsystems.

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 now goes through an allocation dialogue and optionally reads your .LISP. (INIT) file. See page 3-65 for information on this.

LISP may be entered by the command `:LISP name1 name2 dev dir` (or `:LISP dev:dir;name1 name2`). In this case the file `dev:dir;name1 name2` is read in the same way as a `.LISP. (INIT)` file. This can be used to start up a subsystem. The device defaults to DSK, the directory defaults to your current DDT master sname (`..MSNAME`), and the file names default to `.LISP. (INIT)`.

It is also possible to build a subsystem in a LISP, then save it as TS FOO. The :FOO (or FOO^K) command will then enter the subsystem, bypassing the allocation dialogue.

in the DEC-10 implementation

LISP may be entered by the monitor command R LISP or RUN LISP. The allocation dialogue (see page 3-65) is entered. Optionally a `LISP.INI` in the user's directory may be read. If the DEC-10 monitor allows the use of additional arguments on the R command line, then the particular name of a file may be specified (but not a directory). The extension defaults to INI. This file is read in the same way as a LISP.INI file. This can be used to start up a subsystem.

As in the ITS implementation, a subsystem can be saved and then invoked by the appropriate R or RUN command.

in the Multics implementation

The Maclisp subsystem is entered by issuing the LISP command at Multics command level. 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 which is copied into the current environment. This saved environment would contain some subsystem, which will receive control. Additional arguments to the LISP command in this case are actually arguments to the subsystem.

Often one constructs a trivial command for getting into a subsystem, which simply calls the LISP command with the right arguments.

For instance, the LISP compiler subsystem may be entered through the LISP_compiler command, which calls LISP with the pathname of the saved environment containing the LISP compiler as the first argument, and the arguments to the LISP_compiler command as the remaining arguments.

When the standard initial environment (i.e. the ordinary Maclisp subsystem) is entered, it 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.

1.7.2 Saving an Environment

A subsystem is constructed by the following procedure. One starts with the ordinary Maclisp subsystem, and defines a number of function definitions and variable values. This creates an *environment* which is capable of implementing the desired subsystem. This environment is then saved in a file, and necessary mechanisms are set up so that an operating-system command can invoke LISP and cause it to set up the environment saved in the file. When the saved environment is invoked control is passed to the functions in it which then proceed to do the business of the subsystem.

The exact way of saving the environment differs from implementation to implementation. In the Multics implementation there is a function called **save**:

`save`   `FSUBR`

(save foo) saves the current LISP environment in a file named foo.sv.LISP in the working directory. foo is not evaluated. The saving operation destroys the working copy of the environment, so when the save is complete LISP returns to Multics command level.

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 section on Gaining and Keeping Control, below)

In the PDP-10 implementations there is a function called suspend:

`suspend`   `LSUBR 0 to 2 args`

suspend puts LISP in a state such that it can be :PDUMP'ed (ITS) or SSAVE'd (DEC-10) and later restarted. When the saved core image is restarted, everything will be the same as it was when suspended, and control will return from the invocation of suspend.

suspend may be used at any point in a computation, with the restriction that no I/O devices other than the terminal may be in use. If an I/O device other than the terminal is in use, a `fail-act` correctable error occurs, indicating the offending device(s).

In the ITS implementation, care is taken so that all subsystems saved with suspend from the same version of LISP will share the pure pages of LISP. In addition, all invocations of a particular subsystem will share the pure pages peculiar to that subsystem. Declaration of data to be placed in pure pages is described in a later section.

In the TOPS-10 implementation, care is taken so that all LISPs and subsystems saved from LISP will share the same high segment.

After suspend has prepared the LISP core-image for dumping, it returns control to the operating system so that it can be dumped.

If suspend is given an argument, that argument is explodec'ed and the resulting character string is passed back to the operating system as a command. (See cline and valret.) (On ITS this is done with a .VALUE; on SAIL this is done with a PTLOAD. On TOPS-10 the argument is ignored.)

The SAIL version uses the second argument, if present, as a file name. The high segment is dumped under this name. The low segment should then be saved with SAVE, not SSAVE. This is used to create subsystems that can share the same high segment containing code specific to the subsystem.

Commonly one will write a setup routine for a subsystem like this:

```
(progn
        (terpri)
        (princ 'options:)
        ... read in options ...
        (terpri)
        (princ 'loading)
        ... load in files of functions ...
        (gc)
        (sstatus gctime 0)
        (suspend)
        (start-the-subsystem)
))
```

The subsystem's environment is now ready for dumping. Alternatively, one might write (on ITS)

```
(suspend '|:PDUMP DSK:FOODIR;'TS NSUBSYS/
:$ALL DONE$/
|)
```

which will do the dump itself and print a message when done. On SAIL one might write

```
(suspend '|SAVE SYS:FOO.SAV|
         '(FOO SHR SYS))
```

See also the valret function.

1.7.3 Gaining and Keeping Control

In the Multics implementation, when a saved environment is restarted it looks like an error that returns to top level. The forms in the **errlist** are evaluated. These forms should do whatever is necessary to start up the subsystem. The arguments to the command which invoked the subsystem may be obtained via **(status arg)** or **(status jcl)**.

In the PDP-10 implementations, when a saved environment is restarted execution continues from the point where suspend was called. The next form evaluated should do whatever is necessary to start up the subsystem. (It is also possible to cause the same return to top level on startup as in the Multics implementation by using **valret** instead of suspend.) The arguments of the command line which invoked the subsystem may be obtained via **(status jcl)**.

If the subsystem wants to hide the underlying Maclisp from the user, it has a number of facilities available. By setting up its own user-interrupt handlers it can handle any LISP errors which occur itself. In Newio implementations, it can alter, augment, or abolish standard interrupt control characters. It can replace the Maclisp interpretive interaction loop with its own by using **(sstatus toplevel)** and **(sstatus breaklevel)**. It can also provide a totally different interaction loop by not returning control to the LISP top level when it is started, but instead retaining control in its own functions which read and respond to user input.

It is possible for a subsystem to retain the trappings of Maclisp but change the way things read and print. Macro characters and the readtable can be used to change the way input is parsed; alternatively, setq'ing the variable **read** will redefine the system reader function (in the PDP-10 implementations). All output by Maclisp (with the exception of character-string messages) is done through the function **prin1**, and the subsystem may redefine this function. In the Multics implementation one simply redefines it, but in the PDP-10 implementations the variable **prin1** must be bound to the function which is to substitute for **prin1**.

Some subsystems don't do any of this, but simply consist of standard Maclisp augmented by some additional functions which may be used in forms typed in at top level.

1.7.4 Purity

In the PDP-10 implementations, there are some facilities which allow subsystems to put their non-changing data, function definitions, binary code, etc. into pure pages. This decreases the load on memory by sharing pages between multiple users of the same subsystem.

There are some extra storage spaces which are used to store pure (unchanging) LISP objects. These are the pure list, pure fixnum, pure flonum, pure bignum, and pure hunk spaces.

`purcopy` SUBR 1 arg

> This function makes and returns a copy of its argument in pure storage. This is primarily of use in the creation of large sharable systems like MACSYMA. In implementations other than PDP-10 implementations with pure spaces, purcopy simply returns its argument.

> There are a number of features which control how binary code and constants are purified when a compiled program is loaded into LISP.

`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 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`.

`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.

`getsp` LSUBR 1 to 2 args

(`getsp` *n flag*) ensures that (`- bpend bporg`) is at least *n*, allocating more memory if necessary. If *flag* is non-`nil`, then the memory added is marked as being potentially purifiable by `purify`. If *flag* is omitted, the value of **pure** is used. This is generally used by clever subsystems loaders to expand binary program space quickly so that `fasload` will not require several garbage collections to do the same thing. It can also be used by programs which create and destroy many arrays. See also `noret`.

`noret` VARIABLE

Normally the garbage collector will return memory to the time-sharing system if (`- bpend bporg`) is very large, but setting **noret** non-`nil` prevents this. This is useful in conjunction with `getsp`.

`purify` SUBR 3 args

The first two arguments to `purify` should be fixnums, delimiting 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 made pure, but in addition some work is done to make sure that no UUO on those pages may ever be "clobbered". (See **pure** and `purclobrl`) This option should always be used if the pages involved contain binary code loaded by `lap` or `fasload`. Presently `purify` does nothing in the TOPS-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 function might be used to produce a sharable system on ITS:

```
(defun superdump ()
       (setq lopage (pagebporg))      ;save low page address
       (setq pure 3)                  ;specifies pure code
       (setq lopage (+ lopage 6000)) ;allow for area 1
                             ; (ITS page = 2000 words)
       (fasload funny fasl)           ;load up system
       (fasload weird fasl)
       (uread some lap)
            ...
       (sstatus toplevel              ;set up top level for system
               '(top-handler))
       (setq hipage (pagebporg))      ;save high page address
       (purify lopage (1- hipage) 'bporg)    ;purify pages
       (suspend '|:pdump sys:ts super^M|)  ;tell ddt to dump
       (terpri)                       ;stuff for system startup
       (princ 'welcome/ to/ supersystem/!)
       (terpri))
```

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. pure should be the desired size in words of the "uuolinks area"; this is rounded up to a whole number of pages. (If pure is between 1 and 8, it is the number of pages rather than the number of words.) Let the number of pages be *n*. First fasload calls pagebporg, and then reserves 2*n* 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 *n* 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), provided, of course, that the value of nouuo is nil, 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. This allows the called functions to be traced again, for example, or redefined as expr code. Naturally, an area large enough to contain all the UUO's should be reserved; (status uuolinks) (q.v.) yields information relevant to this. As a rule of thumb, the area hsould be 20% larger than the number of functions called by the loaded code (including LISP functions such as equal). For the DEC-10 version, pure may be set to a negative value. the magnitude is used as above, while the sign controls which segment to load into (positive = low segments, negative = high segment). A negative value also causes uuolinks area 1 to go in the low segment, and area 2 in the high segment. For compatibility, a negative value means the same as a positive value to the ITS implementation.

purclobrl   VARIABLE

Used by fasload to keep track of UUO's which are potentially but not immediately clobberable.

*pure   VARIABLE

This variable controls automatic purification of S-expressions and atomic symbols. If it is set non-nil (the initial value is nil), then the following are placed in pure storage spaces instead of regular storage spaces: pnames of atomic symbols; list, fixnum, flonum, bignum, and hunk constants used by code loaded with fasload; properties whose indicators are in the list which is the value of the variable putprop (initially subr, fsubr, and lsubr). In the SAIL implementation, if *pure is a fixnum, it should be an estimate of the *total* number of pure data structures needed, including all files previously loaded and the initial LISP system pure data structures (currently about 6000. words). This causes purcopy to use the high segment for pure data. Making the estimate in *pure too large merely wastes space in the high segment; making it too small causes purcopy to make copies in the low segment when it runs out of room in the high segment. This whole feature only works if pure is a negative fixnum.

putprop VARIABLE

If the value of *pure is non-nil and the third argument to the **putprop** function is in the list of indicators which is the value of the variable **putprop**, then the second argument is passed through **purcopy** to purify the structure. Furthermore, the two cells of the property list are **cons**'ed from pure list space. Since impure cells must precede pure cells in the property list, **putprop** may not put a new property at the front of the property list in this case.

The putprop and remprop functions know about purified property lists. If necessary, they will copy the property list (but not the properties themselves) into non-pure storage so that it can be modified. This is true regardless of the value of *pure. Recall also that defprop and defun use putprop.

1.8 Miscellaneous Functions

1.8.1 The Status Functions

`status`　　　FSUBR

The **status** special form is used to get the value of various system parameters. 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. These arguments may or may not be evaluated, depending on the first argument. If certain additional arguments are omitted, a default value is supplied, again depending on what the first argument is. The various status functions are listed below.

`sstatus`　　　FSUBR

The **sstatus** special form is used to set the value of various system parameters. Its arguments are similar to those of **status**.

These are the things that you can do with **status** and **sstatus**:

STATUS FUNCTIONS FOR I/O

`tabsize` `(status tabsize)` returns the number of character positions assumed between tab stops, which depends on the implementation. Currently this is 8 in the PDP-10 implementation and 10. in the Multics implementation.

`newline` `(status newline)` returns a fixnum which is the ascii code for the character which marks the end of a line of input. For example, one might say `(= (setq ch (tyi)) (status newline))` . Cureently this is 15 (octal) in the PDP-10 implementation and 12 (octal) in the Multics implementation.

`charmode` `(status charmode` *f*`)` returns the value of the character-mode switch for the file *f*. If *f* is **t** or omitted the value of **tyo** (the default output terminal) is assumed. If the character-mode 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. (In the Multics implementation, you can also cause the buffer to be sent by using the `force-output` function.) This provides increased efficiency at the cost of not immediately seeing all output in some cases.

`(sstatus charmode` *x f*`)` sets the character-mode switch of the file *f* (*f* may be `t` or omitted to signify the output terminal in the `tyo` variable) to *x*, which may be `nil` or `t`. *x* and *f* are evaluated. Currently in the PDP-10 implementation it is not possible to set the character-mode switch; one must specify it initially to open. See also `(status linmode)` and `(status filemode)`.

`linmode` `(status linmode)` reads the "line mode" and `(sstatus linmode` *x*`)` sets the "line mode" to *x* (`t` or `nil`.) These functions take an optional extra argument, which is the file whose line mode is being discussed. This defaults to the value of `tyi` (the default input terminal). In any case, this file must be a terminal. In some implementations the "line mode" may not be changed. If the "line mode" is `t`, user input is buffered up a line at a time before being sent to LISP. The input-editing conventions of the host operating system are used. If the "line mode" is `nil`, LISP sees each character as it is typed and applies its own input editing conventions. This mode can provide input facilities more suited to LISP and possibly better handling of the terminal, if it is a type that LISP knows a great deal about. However, it uses more machine resources. It is possible for a user program to take direct control of the terminal when the "line mode" is `nil`; however, this may require knowledge of the `(sstatus tty)`. The Multics implementation always operates with a "line mode" of `t`. See also the `(sstatus ttyscan)` function below.

`ttyint` `(sstatus ttyint` *char func file*`)` turns on a tty interrupt character. When the character *char* is typed on the input tty *file*, the LISP program will be interrupted and the function *func* will be applied to two arguments - *file* and *char*. If *file* is omitted, the value of `tyi` (the default input file) is assumed. The *char* may be either a character object or a fixnum (ascii code). Any of the 128. ascii characters may be used. The *func* may be either an ordinary functional form or a fixnum, which means the default system action for the character with that ascii value. For instance a *func* of 7 means quit back to the top level of LISP (control-G). If *func* is `nil`, the *char* is made non-interrupting. All three arguments are evaluated.

In the ITS implementation, it may be necessary to use `(sstatus tty)` to inform ITS about non-standard interrupt characters.

In PDP-10 implementations which support fixnum tty input mode (more than 128. characters), then when an interrupt occurs the character is folded down to 7 bits before selecting an interrupt function. However, the unfolded character is passed as the second argument to the interrupt function, so that the function can filter out unwanted characters. If a fixnum is used for the function, then the fixnum may specify required or forbidden supra-ascii bits as follows: adding bit n to the fixnum requires the bit, and adding (lsh n 18.) forbids it. For example, in the ITS implementation using 400000207 (octal) as a function allows CTRL/g to perform a quit, but not the "pi" character or control-meta-g, since 200 requires the control bit and 400000000 forbids the meta bit.

Example: a standard subsystem convention (used by NCOMPLR, MACSYMA, and SCHEME, for example) is to use ^^ (control-uparrow, ascii 036) as a quit character which restarts the subsystem. This leaves ^g with its normal meaning of "exit to LISP toplevel," which is occasionally also useful.

```
(sstatus ttyint 36
        (function (lambda
                   (f c)
                   (sstatus toplevel '(restart-subsystem))
                   (do nil ((or (= (listen f) 0)
                                (= (tyi f) 36))))
                   (^g))))
(defun restart-subsystem ()
       (setq errlist nil)
       (sstatus toplevel nil)
       (nointerrupt nil)
       (initialize-the-world)
       (enter-subsystem))
```

(status ttyint *char file*) returns *func*, the interrupt function for the character *char* on the tty *file*. *file* may be omitted. It defaults to **t**.

ttycons (sstatus ttycons *tty1 tty2*) binds two tty files into a console. One should be an input tty and the other an output tty. If *tty1* or *tty2* is **t**, it will be taken as the appropriate direction of the default terminal (determined by the variables tyi and tyo). The binding into a console is primarily used for purposes of echoing. In addition, interrupt characters typed on an input tty file should affect the output on its corresponding output tty file, not on some other tty. If **tty1** or **tty2** already has a ttycons relationship, that relationship is broken before the new one is established.

If one argument is a tty and the other is nil, a ttycons relationship involving that tty will be broken. Closing a tty file automatically breaks any ttycons relationship it may have. (See close.)

To put echoing at the bottom of the screen, distinct from output,

```
(sstatus ttycons t
                  (setq echotty (open 'tty: '(echo out tty
                        echotty))
```

which conses tty input with a new tty output channel which is set to go to the echo area at the bottom of the screen.

(status ttycons *ttyl*) returns the other tty file which is bound into a console with *ttyl*, or nil if there is none.

filemode (status filemode *file*) => (open-mode-list . internal-cruft). *open-mode-list* is a suitable second argument for the open function. *internal-cruft* is a list of implementation-dependent information which may sometimes be needed by special programs.

The following symbols may appear in *internal-cruft*. (These are the standardized ones; additional symbols may appear at the discretion of the implementation.)

cursorpos	This file (an output tty) has the ability to position its cursor anywhere on the screen. See cursorpos.
filepos	The filepos function can be used to access randomly within the file. See the filepos function.
rubout	This file (an output tty) has selective erase capability. (cursorpos 'x) will work. See also the rubout function, page .
sail	This file (an output tty) has the so-called SAIL character set. This is an extension of ascii which is related to but not necessarily identical to the actual character set used at SAIL.

If *file* is a closed file object, `(status filemode)` merely returns `nil` instead of giving an error.

`ttyscan` `(sstatus ttyscan` *func file*`)` allows the user to supply a function which performs initial processing of terminal input. *func* is a functional form, and *file* must be a tty input file. If it is omitted, the value of `tyi` (the default input terminal) is assumed. Both arguments are evaluated.

When LISP wants to take input from *file*, it first calls the prescan function, which is supposed to gobble down a complete unit of input (for instance an S-expression) and return a list of characters. The prescanner supplied automatically by LISP when a tty input file is first opened counts parentheses and does fancy rubout processing on display terminals. It also implements the CTRL/k, CTRL/l, and CTRL/u characters (as appropriate to the implementation) which allow the complete input to be redisplayed or cancelled, and takes care of force-feed characters. A user-written prescanner might provide additional features such as super-parentheses, name recognition and completion, or fancy editing.

The prescan function *func* is applied to three arguments: the *file*, the name of the input function on whose behalf it is acting (`read`, `readch`, or `readline`), and a fixnum which, in the case of `read`, is the count of the number of unmatched left-parentheses. It is supposed to return a list of fixnums, which represent characters. The prescan function should read the input with `tyi`, since it and `tyipeek` are the only input functions which don't call the prescan.

If the prescan function returns `nil`, an eof condition occurs for the input file. This is the standard way to signal over-rubout.

There is a function called `rubout` to assist the pre-scanner in processing rubouts. It is described on page .

It is a good idea for the prescan function to `lambda-bind echofiles` to `nil` so that characters do not appear in the echo files twice, and so that the echo files reflect the "clean" input after rubout processing. The system-supplied prescan function does this.

`(status ttyscan` *file*`)` returns the *file*'s *func*.

This feature does not presently exist in the Multics implementation.

The following status functions exist only in the ITS implementation

`ttysize` (`status ttysize` *f*) returns the height and width of the terminal open as the file *f*. If *f* is omitted, the value of `tyo` (the default output terminal) is used. The result is a dotted pair of fixnums "(*ttyheight* . *ttywidth*)" (cf. `cursorpos`). If the terminal is a printing console instead of a display, then the height will be a very large number such as 200000000000 or so.

`ttytype` (`status ttytype` *f*) returns the type of the terminal open as the file *f*. If *f* is omitted, the value of `tyo` (the default output terminal) is used. The result is a fixnum taken from the TCTYP (not TTYTYP!) variable in ITS:

0	Printing terminal.
1	Good Datapoint.
2	Bad Datapoint ("loser").
3	Imlac.
4	Tektronix.
5	PDP-11 ("Knight") TV.
6	Memorex.
7	Software terminal.
10	Terminet.
11	Display using standard ascii display codes.
12	Datamedia.

This status call is meant only for esoteric purposes. To find out whether you can position the cursor or erase characters, use `(status filemode)`.

`tty` (`status tty` *f*) returns a list of three fixnums which control the behavior of the terminal open as file *f*. If *f* is omitted, the value of `tyi` (the default input terminal) is used. (`sstatus tty` *x y z f*) sets these three fixnums to *x*, *y*, and *z*. If *z* is omitted, it is not changed. If *f* is omitted, the value of `tyi` (the default input terminal) is used. The three fixnums are the ITS variables `TTYST1`, `TTYST2`, and `TTYSTS`. Their meaning is as follows:

`TTYST1` and `TTYST2` are divided into twelve groups of six bits. Each group controls a certain subset of the ascii characters. These groups, in left-to-right order within `TTYST1` and `TTYST2`, are:

```
(0) ^@-^F ^K ^L ^N-^R ^T-^Z ^\-^_
(1) A-Z a-z (letters)
(2) 0-9     (digits)
(3) ! " # $ % & ´ , . : ; ? @ \ ` | ~
(4) + * - / = ^ _ (arithmetic operators)
```

```
(5) < > ( ) [ ]  (parentheses)
(6) ^G ^S
(7) ^I ^J (tab, linefeed)
(8) altmode (ascii 33)
(9) ^M (carriage return)
(10) rubout (ascii 177)
(11) space, ^H (backspace)
```

The meanings of the six bits in each group are:

40	Echo the character when read by LISP (normal mode).
20	Echo the character when typed (this may disappear from ITS soon!).
10	Echo in "image mode" rather than "ascii mode"
4	Convert lower case to upper case (applicable only to letters; not normally used by LISP - the **readtable** accomplishes case conversion).
2	Activation characters. If this bit is not set, then ITS will not schedule LISP to run when you type a character within the group even if LISP was waiting for terminal input. The only purpose of this is to increase the efficiency of the time-sharing system. For example, letters and digits normally have this bit off, since typing one cannot terminate an S-expression, and there is no point in "activating" LISP until an S-expression terminator like space or ")" has been typed. An exception to this rule is that for **tyi** and **readch** LISP asks ITS to activate on any character. Normally only groups (0), (5), (6), (7), (9), (10), and (11) are activators. If you alter the readtable, you may wish to change the activator groups.
1	Interrupt characters. If this bit is not set, then even if you have used **(sstatus ttyint)** to set up an interrupt function you will not get the interrupt since ITS won't tell LISP about it. Initially only groups (0) and (6) cause interrupts (these include all control-characters except altmode and format effectors). If you give **@** an interrupt function, for example, you should set this bit for group (3).

On a keyboard with supra-ascii characters, one may wish to use meta-characters for interrupts. Suppose we want meta-D to run the function **foo**. Then we would turn on the interrupt bit for group 1 (letters) and say:

```
(sstatus ttyint 104      ;letter D
        (function (lambda (f c)
                    (and (= c 504)  ;meta bit
                         (foo))))   ; is 400
                  - filter out plain "D"
```

Notice that this only handles meta-D, not meta-d. To handle the latter, one must also do (sstatus ttyint 144 ...).

Another way to filter out plain "D" is

```
   (sstatus ttyint 504    ;require meta bit
            '(lambda (f c) (foo)))
```

which is somewhat more efficient. On the other hand, to use both meta-D and control-meta-D, we must do a dispatch on the control bit:

```
   (sstatus ttyint 4
            '(lambda (f c)
                    (cond         ;ignore
                     ((or (= c 304)(= c 344))
                      (setq ^d t))
                     ((or (= c 504)(= c 544))
                      (meta-d-foo))
                     ((or (= c 704)(= c 744))
                      (control-meta-d-foo)))))
```

The third variable, TTYSTS, has these bits of interrest:

40000_22 (%TSFCO) If set, output on this terminal uses the supra-ascii convention of an "alpha" prefix for "control" and "beta" for "meta".

20000_22 (%TSALT) If set, ascii codes 175 and 176 are not converted to 33 on input.

10000_22 (%TSROL) If set, terminal is in scroll mode.

4000_22 (%TSSAI) If set, echoing and ascii output use the SAIL character set; otherwise, control characters are output in the form "^X".

10_22 (%TSNOE) If set, suppress echoing of typed characters.

2_22 (%TSSII) If set, terminal uses "super-image input" mode. Not even ^Z and ^_ can take their usual effect for ITS.

Notice that some of these bits affect output even though the argument is an input file; they affect the output terminal which is logically related by ITS to the input terminal. Normally LISP or the user will arrange for (status ttycons) to reflect this relationship.

STATUS FUNCTIONS FOR THE OLD I/O SYSTEM

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

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

`uwrite` (status uwrite) returns the 2-list for the current **uwrite** output destination.

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

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

(sstatus crunit device directory) sets the current default device and directory for uread, etc. The arguments are not evaluated.

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

(sstatus crfile name1 name2) sets the current default file names for uread, etc. The arguments are not evaluated.

STATUS FUNCTIONS FOR THE READER

See section 13.6.2 for a description of how the parameters controlled by these functions are used. All of these parameters are kept in the readtables, and the status functions below deal with the readtable which is the value of the variable **readtable**. 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.

`chtran` (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 (not preceded by slash) in a pname being read in. This feature is used in the PDP-10 implementations to translate lower-case input to upper case.

(sstatus chtran *c* *k*) sets *c*'s character translation to *k*. *k* is always evaluated. See the setsyntax function.

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

(sstatus syntax *c m*) sets *c*'s syntax bits to *m*. *m* is evaluated and returned. The setsyntax function is usually a better way to do this, however.

Note that in the above two sstatus 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 ; and |), instead of being changed to its standard syntax and chtran its syntax is set to 502 (slashified extended alphabetic) and its chtran is set to itself.

macro (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 splicing for splicing macros.

(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. The setsyntax function is generally a better way to do this, however.

+ (status +) gets the value of the + switch (t or nil) in the readtable. 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. See ibase.

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

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

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

STATUS FUNCTIONS FOR THE PRINTER

`terpri` (status terpri *file*) returns the value (t or nil) of the terpri switch for the *file*, which defaults to t. 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. See also the **terpri** variable.

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

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

(sstatus _ *x file*) sets the _ switch to the value of *x*, t or nil.

`abbreviate`(status abbreviate) returns the value of the abbreviation control. See section 13.7 for a description of the abbreviation control. Note that the abbreviation control is kept in the current readtable.

(sstatus abbreviate *n*) sets the abbreviation control to *n*.

(sstatus abbreviate nil) turns off abbreviation.

(sstatus abbreviate t) turns on maximal amount of abbreviation.

STATUS FUNCTIONS FOR THE GARBAGE COLLECTOR

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

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

It is a good idea to do (gc) and then **(sstatus gctime 0)** just before dumping out a subsystem. See also **(status gcwho)**.

`spcnames` (status spcnames) returns a list of the names of all the spaces available in the LISP being used. These are the names acceptable to the **alloc** function and to the following status functions which require a space argument.

spcsize	(status spcsize *space*) returns the actual current size of *space*, in words. *space* is evaluated.
gcmax	(status gcmax *space*) returns the gcmax parameter for *space*. sstatus gcmax *space n*) sets the gcmax parameter for *space* to *n*. See the alloc function.
gcmin	(status gcmin *space*) returns the gcmin parameter for *space*. sstatus gcmin *space n*) sets the gcmin parameter for *space* to *n*. See the alloc function.
gcsize	(status gcsize *space*) returns the gcsize parameter for *space*. This is not the actual size of the space - see (status spcsize). sstatus gcsize *space n*) sets the gcsize parameter for *space* to *n*. See the alloc function.
purspcnames	(status purspcnames) returns a list of names of spaces which have pure versions. See purcopy.
pursize	(status pursize *space*) returns the actual current size of the pure version of *space*. Only spaces returned by **(status purspcnames) have** pure versions.
pdlnames	(status pdlnames) returns a list of the names of all the pdls available in the LISP being used. These are the names acceptable to the following status functions which require a pdl argument.
pdlsize	(status pdlsize *pdl*) returns the current number of words on a pdl.
pdlroom	(status pdlroom *pdl*) returns the "pdlroom" of a pdl, i.e. the maximum size to which it may ever grow.
pdlmax	(status pdlmax *pdl*) returns the current value of the pdlmax parameter of a pdl. (sstatus pdlmax *pdlsize*) sets the pdlmax parameter for the pdl *space* to *size*. Both arguments are evaluated. See also the alloc function.

The following status function exists only in the PDP-10 implementation

memfree — (status memfree) returns the number of words of address space not yet allocated for any purpose (i.e. still available for allocation to various spaces).

ENVIRONMENT ENQUIRIES

Note: the various enquiries related to time may return nil in the rare circumstance that LISP cannot determine what time it is.

date — (status date) returns a 3-list of fixnums indicating the current date as (last-two-digits-of-the-year month-number day).

dow — (status dow) returns an interned atomic symbol which is the name of the current day of the week.

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

system — (status system *x*) returns a list of the system properties (that is, properties whose values are supplied in the initial LISP system) of the atomic symbol *x*, which is an evaluated argument. This list may contain subr, fsubr, lsubr, macro, autoload, or array if *x* is a system function, and value if this atomic symbol is a system variable.

LISPversion

(status LISPversion) returns the version identification of LISP. This is usually an atomic symbol.

jcl — (status jcl) returns the "job command line" from DDT in the PDP-10 implementation. This is used to specify the init file, but interpretation of the file name is terminated by altmode, and (in the ITS implementation at least) additional information may follow the altmode. The init file may wish to examine this information. The DEC-10 implementation supports two syntaxes for jcl:

```
     .R LISP; -jcl-
and  .R LISP(-jcl-)
```

Not all DEC-10 operating systems support both syntaxes, however, and some support neither. In the Multics implementation this returns the explodec'd second argument of the LISP command, or else nil if the LISP command did not have two arguments. (See **(status arg)**.) If LISP was invoked by

LISP environment_name "foo bar"

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

This function is also used by subsystems implemented in Maclisp to pick up the arguments from the command which invoked them.

udir	(status udir) returns the name of the user's directory. In the ITS implementation this is the user's "master sname", which is usually 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 TOPS-10 implementation this is a list (proj prog).
uname	(status uname) returns an interned atomic symbol whose pname is the user's login name. In the ITS implementation this is just the .UNAME . In the Multics implementation this is in the format User.Project.Instance; the dot will be slashified if print is used to display this. In the TOPS-10 implementation this is actually a list (proj prog) representing the project-programmer number rather than a symbol. See also **(status userid)**.

The following status functions exist only in the PDP-10 implementation.

userid	(status userid) tries to return an identification of the user (as opposed to an identification of an instance of his being logged in). Thus if a user is logged in on several terminals, (status uname) may differ among them, but (status userid) should be the same. In the ITS implementation this returns the .XUNAME as an interned atomic symbol. Normally the .XUNAME is the same as .UNAME with trailing digits stripped off. It is possible, however, to cause a job to be run with a different .XUNAME. Subsystem initializations should check the userid to determine which init files to use. In the TOPs-10 implementation this the the .GTNM1 and .GTNM2 GETTAB variables as an atomic symbol, if they exist, and otherwise is the same as (status uname). In the SAIL implementation this is the same as (status uname).
jname	(status jname) returns the job name as an interned atomic symbol. The jname of a job is a unique identifier for it among all of the user's jobs. In the ITS implementation this is the job's .JNAME . In the DEC-10 implementation this is *nnn*LISP, where *nnn* is the job number in decimal.

xjname (status xjname) in the ITS implementation returns the .XJNAME as an interned atomic symbol. This is normally the name of the program that the user invoked to create this job. Usually this will be the same as the jname, but sa user can invoke several copies of a program at once, each having a unique jname but all having the same xjname. In the DEC-10 implementation this will return the job's "program name". On Multics, it should return (status arg 0).

seglog (status seglog) returns the logarithm (base 2) of the size of a "segment", the unit of space allocation. (The size of a segment need not be the same as the size of a memory page.)

The following status functions only exist in the ITS implementation.

its (status its) returns a list of five items (obtained from the SSTATU symbolic system call):

(1) The amount of time in seconds until ITS is going down, as a flonum, -1.0 if ITS doesn't plan to go down, or -2.0 if ITS already is down.

(2) a fixnum which is nonzero if ITS is being debugged.

(3) the number of users logged in, as a fixnum.

(4) the number of memory errors the system has survived, as a fixnum.

(5) the time in seconds the system has been up, as a flonum.

Some of this information may be of use to a user handler for the **sys-death** interrupt.

hactrn (status hactrn) returns an atomic symbol indicating what kind of job is superior to the LISP. Current possibilities are **ddt**, **LISP**, **t** (a superior of unknown type), and nil (no superior).

The following status functions exist only in the Multics implementation.

paging (status paging) returns a list of the paging-device page reads and total page reads that have been caused by this process.

arg (status arg *n*) returns the *n*+1'th argument of the command which invoked the subsystem, as an interned atomic symbol. (The first argument, (status arg 0), is the name of the subsystem.) nil is returned if *n* is greater than the number of arguments to the command.

STATUS FUNCTIONS FOR THE WHO-LINE

The following status functions exist only in the ITS implementation.

`who1` (sstatus who1 *a b c d*) sets four user parameters for the "who-line" which appears at the bottom of some display terminals. These control the print format of information specified by (sstatus who2), (sstatus who3), and (sstatus gcwho). Each of the four parameters specified eight bits of information. Parameters *a* and *c* must be fixnums; *b* and *d* may be fixnums or atomic symbols (which represent the ascii value of the first character in their pnames). Their meaning is as follows:

a 200 If 1, suppress display of entire who-line.
100 If 1, suppress space between halves of who2.
70 Mode for printing high 18. bits of who2.
7 Mode for printing low 18. bits of who2.

b 177 If non-zero, print between halves of who2 as an ascii character.
200 If 1, print character twice.

c 200 If 1, suppress space between who2 and who3.
177 As for *a*, but affects who3.

d 377 As for *b*, but affects who3.

That is, if the who-line is displayed at all, the user information appears in the form:

PPP**-QQQQ=RRRR@@+SSSS

where:

PPPPis the result of printing the high 18. bits of who2 as specified by *a*'s 70 bits.

QQQQlow 18. bits of who2, by *c*'s 7 bits.

RRRRhigh 18. bits of who3, by *c*'s 70 bits.

SSSS low 18. bits of who3, by *c*'s 7 bits.

** zero to two characters, specified by *d*.

- space, unless *a*'s 100 bit is set.

=	space, unless *c*'s 200 bit is set.
+	space, unless *c*'s 100 bit is set.

The possible print modes are:

0 Do not print.

1 Date in packed form:

774000 Year mod 100.

3600 Month (January = 1)

174 Day of month.

Printed as mm/dd/yy.

2 Time in fortieths of a second. Printed as hh:mm:ss.t.

3 Time in half-seconds, printed as hh:mm:ss.

4 Octal number.

5 Decimal number (no "." is printed).

6 Three sixbit characters.

7 Unused.

`(status who1)` returns a list of four fixnums (*a b c d*).

who2 `(sstatus who2 x)` sets the who2 variable to *x* if it is a fixnum, or to the sixbit value of the first six characters of the **pname** of *x* if it is a symbol. `(status who2 x)` returns who2 as a fixnum.

who3 `(sstatus who3 x)` and `(status who3)` are analogous to those for who2.

Example:

```
(all numbers in octal)
(sstatus who1 166 0 144 254)
(sstatus who2 'FOOBAR)
(sstatus who3 (+ (LSH 1234 22) 3456))
```

would cause "FOOBAR 1234,,3456" to appear in the who-line. The general idea is that the user should set up who1 just once and then keep updating who2 and who3 as appropriate. For example, a routing that processed functions in files might do

```
                (sstatus who1 166 0 166 0)
```

and on opening a new file do

```
                (sstatus who2 (cadr (truname infile)))
                (sstatus who3 0)
```

and on encountering a function do

```
                (sstatus who3 fnname)
```

and so continuously display the filename and function name being processed. One advantage of doing this over printing the information is that LISP can update who-line information even though it doesn't have control of the console.

gcwho (sstatus gcwho *n*) sets the gcwho switch to *n*. Currently only the low two bits are significant.

The 1 bit means that during a garbage collection the who-line should be usurped to display the message "GC:XXXX", where "XXXX" is the reason for the garbage collection. On completion of the garbage collection the original user who-line information is restored.

The 2 bit means that at the end of the garbage collection the who2 word should be altered so that the low 18. bits get the result of (status gctime), converted to fortieths of a second, and the high 18 bits get the result of (// (* (status gctime) 100.)(runtime)), which is the percentage of time spent in gc. If the user gives 52 octal and '% as the first two arguments to (sstatus who1), then these quantities will be displaced in the form "nn% hh:mm:ss.t", just like the standard system statistics for the LISP job. The user can still use who3 for his own purposes.

Subsystems which use (sstatus gcwho) should perform

```
    (gc)
    (sstatus gctime 0)
```

before dumping out the subsystem, so that the gc time percentages will be accurate .

(status gcwho) returns the value of the gcwho switch.

MISCELLANEOUS STATUS FUNCTIONS

evalhook (sstatus evalhook t) enables the evalhook feature; (sstatus evalhook nil) disables it. (status evalhook) returns the state of the switch. See page 3-31 for the details of this feature.

toplevel (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.

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

For example, to make Maclisp have an *evalquote* top level similar to LISP 1.5:

```
    (sstatus toplevel
             '(progn (print *)
                     (apply (read) (read)) ))
```

See section 12.1 for further details.

breaklevel (status breaklevel) returns the break-loop form. (sstatus breaklevel *x*) sets this form. See page 3-6 for how this is used.

uuolinks (status uuolinks) returns a number which represents the number of available slots for linking between compiled functions. (status uuolinks) returns nil if no uuolinks pages have been set up yet (see pure). Otherwise it returns a list of two items. The first is t if area 2 has been purified (and so no new calls may be put in the area), and nil otherwise. The second is the number of slots not yet used in the uuolinks areas.

(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.

divov (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.

(sstatus divov *x*) sets the "divide overflow" switch to *x*.

In the PDP-10 implementation, divov applies only to quotient. // and //$ do not detect division by zero.

mar (sstatus mar *cond loc*) arms the mar (memory address register) interrupt (currently available only in the ITS implementation). (status mar) returns a list of *cond* and *loc*, or nil if the mar feature is not in use. See the mar break feature for more details (page 3-55).

features (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 loaded
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
hunk	the hunks data type and its associated functions are present.
funarg	the "fake funarg" feature (second argument to eval, third to apply, etc.) is present.
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.
roman	this LISP can read and print roman numerals (see base and ibase.
trace	the trace package (chapter 15) is present.

grindef	the function definition formatter (chapter 16) is present.
grind	the file formatter (chapter 16) is present.
compiler	this is the LISP compiler (chapter 14) rather than the interpreter (see also `compiler-state`).
`fastarith`	`the fast-arithmetic features of the compiler are present`
`ml`	`this LISP is on the MathLab machine at MIT`
`ai`	`this LISP is on the AI machine at MIT`
`mc`	`this LISP is on the MC machine at MIT`
`SAIL`	`this LISP is running at SAIL`
`H6180`	`this LISP is on an H6180 Multics machine or a compatible machine such as a 68/60 or a 68/80.`
`its`	`this LISP is on some ITS system`
`Multics`	`this LISP is on some Multics system`
`TOPS-10`	`this LISP is on some DEC TOPS-10 system; or on some TENEX system since the TENEX implementation runs under a TOPS-10 emulator.`

`A package being "present" means that it has been loaded into the environment.` If `(status features)` claims it is not present, it may still be available because it may be automatically loaded when required. This does not apply to the compiler.

`(car (last (status features)))` is an implementation name, such as ITS or TOPS-10 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.

feature (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.

(sstatus feature *foo*) makes *foo* a feature. *foo* is not evaluated. For example, the trace package does (sstatus feature trace) when it is loaded.

Example:

```
(cond ((status feature bignum)
       (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)))) ))
```

nofeature (sstatus nofeature *foo*) makes *foo* not be a feature. *foo* is not evaluated.

(status nofeature *foo*) is equivalant to (not (status feature *foo*)).

status (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.

sstatus (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.

1.8.2 Time

runtime SUBR no args

(runtime) returns as a fixnum 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 in seconds that the system has been up, as a flonum. The difference between the results of two calls to time indicates the amount of elapsed real time. (In the ITS implementation, time that elapses while the system is stopped due to memory errors is not considered "real" and not counted. Use (status daytime) to measure true "real world" time.)

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 1.4.3, and the date, daytime, and dow functions of the status special form, described in the preceding section.

1.8.3 Escaping from LISP

It is possible to escape temporarily from LISP to execute a command in the host operating system. Of course, the program (or user) that supplies the command has to know which operating system it is running under. It is also possible for LISP to return permanently to the host operating system. This discards the LISP environment and gives back whatever resources, such as memory, it was using.

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")
```

`quit` SUBR no args

(`quit`) returns from the LISP command, freeing up the temporary segments that were used to hold the LISP environment.

in the ITS implementation

`valret` LSUBR 0 or 1 args

(`valret`) 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 `|:PROCEED :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 '|:PROCEED :DISOWN |)
```

starts the LISP running on its own without a terminal.

```
(valret '| :KILL :TECO^M|)
```

kills the LISP and starts up a TECO.

```
(valret '0$N$P)
```

causes DDT to print out the contents of all non-zero locations in LISP and then return to LISP.

If the valret command leaves you in DDT (or anywhere other than the LISP), a :CONTINUE or $P command to DDT will return to the LISP, which will then return from the call to valret. However, valret differs from suspend in two ways: it does not check for open files, and it does not alter the starting address of the LISP. If you use valret to dump a subsystem, then starting up the subsystem causes a return to top level and the consequent evaluation of the top-level errlist (not the binding of errlist in effect at the time of the valret!). Using suspend, the subsystem startup merely causes a return from the suspend function.

in the TOPS-10 implementation

There is currently no way for LISP to return a command string to the Monitor in the TOPS-10 implementation. However, (valret) will return control to the monitor so that a command may be manually typed. Then type CONTINUE to resume LISP.

The remarks above about valret and suspend hold for the TOPS-10 implementation also.

1.8.4 Additional Functions

subr   SUBR 1 arg

The argument, a fixnum, is taken to be the address of a location within a compiled or system function. subr attempts to determine the name of the function by groveling through the current obarray looking at all the compiled function properties. If it fails it returns "?". This is used by baktrace, for example. This function may be useful to user handlers for the machine-error interrupt, for example.

The following function exists only in the ITS implementation

`syscall` LSUBR 2 to 10 args

This function allows the LISP user to directly issue ITS symbolic system calls. The first argument should be

```
(+ (lsh c 18.) n)
```

where *n* is the desired number of output values and *c* is the control bits for the call. The second argument should be an atomic symbol whose pname is the name of the ITS system call. The rest of the arguments should be the input arguments for the call. An argument may be a fixnum, or a file object (for which the file's channel number is used). If the call succeeds, a list of *n* fixnums is returned. If an error occurs, the ITS error code is returned as a fixnum.

Example: `(syscall 0 'scml tyi 3)`
sets the number of echo lines for the tty to 3 and returns `nil` on success.