MACLISP REFERENCE MANUAL
December 17, 1975

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 1565656565656565656565656565656565. 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 2-48. 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 2-54).

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 2-85 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 1-24).

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 2-48 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 1-24 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 2-10, 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 2-11.)

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 2-40 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 2-34 for the details of prog and do.

Defining Functions

(defun *name* (*arg1 arg2*...) *form1 form2*...) defines an (interpreted) function. See page 2-56 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 part 3.2 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 part 3.5 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.

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.

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 2-61).

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`.

`symeval` SUBR 1 arg

`symeval` is used to get the value of an atomic symbol, when the particular symbol which will be used is not known when the program is written, (for example in a language interpreter written in Lisp.) If the argument to `symeval` is not an atomic symbol, or is an atomic symbol but does not currently have a value, an error is signalled. The advantage of `symeval` over `eval` is that it is compiled into very efficient code (which will not detect the above-mentioned error, so watch out.)

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 part 3.5) 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 2-53).

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**.

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. *tags* are skipped over; *statements* 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 *init*s are evaluated, then the *var*s are saved, then the *var*s are setq'ed to the values of the *init*s. To put it another way, the *var*s are lambda-bound to the values of the *init*s. Note that the *init*s are evaluated *before* the *var*s are bound. At the beginning of each succeeding iteration those *var*s that have *repeat*s get setq'ed to the values of their respective *repeat*s. Note that all the *repeat*s are evaluated before any of the *var*s 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 *repeat*s, the *end-test* is evaluated. If the result is nil, execution proceeds with the body of the do. If the result is not nil, the *exit-forms* are evaluated from left to right and then do returns. The value of the do is the value of the last *exit-form*, or nil if there were no *exit-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 *repeat*s. 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 part 3.4) 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 (add1 x)))
```

This example may not work in compiled code if the compiler chooses to open-code the `add1` rather than calling the `add1` 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 .t a *value cell,* which is a piece of storage that can refer to one Lisp object. This obj.ct 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 1-13.) 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 2-81 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 2-87 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 part 3.6).

`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 2-87) 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
```

`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.

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 !7 (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 2-56), 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 2-54.

`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, uninterned, whose pname is given as a character string argument. Example:

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

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
	its own second argument	map	mapc
returns	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))) ))))
```

Function Index

*	2-67	LSUBR 0 or more args
*$	2-70	LSUBR 0 or more args
*array	2-88	LSUBR 3 or more args
*function	2-6	FSUBR
*rearray	2-88	LSUBR 1 or more args
+	2-67	LSUBR 0 or more args
+$	2-70	LSUBR 0 or more args
-	2-67	LSUBR 0 or more args
-$	2-70	LSUBR 0 or more args
/	2-67	LSUBR 0 or more args
/$	2-70	LSUBR 0 or more args
1+	2-68	SUBR 1 arg
1+$	2-71	SUBR 1 arg
1-	2-68	SUBR 1 arg
1-$	2-71	SUBR 1 arg
<	2-61	SUBR 2 args
=	2-61	SUBR 2 args
>	2-61	SUBR 2 args
abs	2-63	SUBR 1 arg
add1	2-65	SUBR 1 arg
alphalessp	2-52	SUBR 2 args
and	2-32	FSUBR
append	2-17	LSUBR 0 or more args
apply	2-5	LSUBR 2 or 3 args
args	2-57	LSUBR 1 or 2 args
array	2-88	FSUBR
arraycall	2-12	FSUBR
arraydims	2-89	SUBR 1 arg
ascii	2-79	SUBR 1 arg
assoc	2-26	SUBR 2 args
assq	2-27	SUBR 2 args
atan	2-73	SUBR 2 args
atom	2-1	SUBR 1 arg
bigp	2-1	SUBR 1 arg
boole	2-75	LSUBR 3 or more args
boundp	2-46	SUBR 1 arg
caaaar	2-14	SUBR 1 arg
caaadr	2-14	SUBR 1 arg
caaar	2-14	SUBR 1 arg
caadar	2-14	SUBR 1 arg

Atom Index

Concept Index