

*Vary vs. constants  
which-hurt*

*the*

*to*

*bit-string  
→ bit-vector*

*Capitalize  
"Gaussian"?*

# CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

---

## Common Lisp Reference Manual

Guy L. Steele Jr.

26 February 1982

---

*FLET LABELS ?*

*Annotatious by  
Guy Steele  
(in multiple colors)*

*CACHEF ?*

**Flatiron Edition  
Still a Few Odd Wrinkles Left**

Spice Document S061

Keywords and index categories: PE Lisp & DS External

Location of machine-readable file: CLM.MSS.12 @ CMU-20C

Copyright © 1982 Guy L. Steele Jr.

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

### Acknowledgements

The many people who have contributed to the design of COMMON LISP are hereby gratefully acknowledged:

Alan Bawden  
 Rodney A. Brooks  
 Richard L. Bryan  
 Glenn S. Burke  
 Howard I. Cannon  
 George J. Carrette  
 David Dill  
 Scott E. Fahlman  
 Neal Feinberg

John Foderaro  
 Richard P. Gabriel  
 Joe Ginder  
 Richard Greenblatt  
 Martin Griss  
 Charles L. Hedrick  
 Earl A. Killian  
 John L. Kulp  
 John McCarthy  
 Don Morrison

David A. Moon  
 William L. Scherlis  
 Richard M. Stallman  
 Barbara K. Steele  
 William vanMelle  
 Allan C. Wechsler  
 Daniel L. Weinreb  
 Jon L. White  
 Richard Zippel

The organization, typography, and content of this document were inspired in large part by the *MacLISP Reference Manual* by David A. Moon and others, and by the *LISP Machine Manual* by Daniel Weinreb and David Moon, which in turn acknowledges the efforts of Richard Stallman, Mike McMahon, Alan Bawden, and "many people too numerous to list".

### Apology

For reasons unknown, Xerox has chosen not to provide any font for the Dover which faithfully reflects the ASCII standard. More precisely, there appears to be no simple way to get correct printed representations of the 95 standard ASCII printing characters. Many fonts do not have an accent grave; in other the accent grave is identical in appearance to the accent acute. Most fonts have a swung dash in place of the tilde, an uparrow or caret in place of the circumflex, and/or a leftarrow in place of the underscore.

This edition uses the SAILA family of Dover fonts for code. At CMU, at least, SAILA suffers from the swung-dash deficiency, and the accents acute and grave are not symmetric. The swung-dash problem has been compensated for by a SCRIBE macro.

*underscore !!*

For reference, here are the 95 ASCII printing characters (the first is the space character) as they appear in the code font in this edition:

! " # \$ % & ' ( ) \* + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?  
 @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ \_  
 ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~

*foo*

— Guy L. Steele Jr.

## Notes to the Swiss Cheese Edition

This edition is incredibly unpolished. It suffers from the following known deficiencies:

- ~~The chapter on macros and `defmacro` is not yet written.~~
- The chapter on the evaluator is not yet written.
- The chapter on how programs are expressed as S-expressions (which includes `defun`, `defvar`, `defconst`, and so on) is not yet written.
- ~~There is no coherent description of `setf` and related special forms.~~
- Nothing is yet written on packages and `intern`.
- No single entire pass has been made yet to catch inconsistencies.

Please send remarks, corrections, and criticisms to `Guy.Steele@CMUA`.

# Chapter 1

## INTRO

### 1.1. Purpose

This manual documents a dialect of LISP called "COMMON LISP", which is intended to meet these goals:

*Commonality.* At least four implementation groups already actively at work on LISP implementations for various machines are considering supporting this dialect. While the differing implementation environments will of necessity force incompatibilities among the implementations, nevertheless COMMON LISP can serve as a common dialect of which each implementation can be an upward-compatible superset.

*Portability.* COMMON LISP intentionally excludes features which cannot easily be implemented on a broad class of machines.

On the one hand, features which are difficult or expensive to implement on hardware without special microcode are avoided or provided in a more abstract and efficiently implementable form. (Examples of this are the forwarding (invisible) pointers and locatives of Lisp Machine LISP. Some of the problems which they solve are addressed in different ways in COMMON LISP.)

On the other hand, features which are useful only on certain "ordinary" or "commercial" processors are avoided or made optional. (An example of this is the type declaration facility, which is useful in some implementations and completely ignored in others; such declarations are completely optional and affect only efficiency, never semantics.)

Moreover, attention has been paid to making it easy to write programs in such a way as to depend as little as possible on machine-specific characteristics such as word length, while allowing some variety of implementation techniques.

*Power.* COMMON LISP is a descendant of MACLISP, which has always placed emphasis on providing system-building tools. Such tools may in turn be used to build the user-level packages such as INTERLISP provides; these packages are not, however, part of the COMMON LISP core specification. It is expected such packages will be built on top of the COMMON LISP core.

*Expressiveness.* COMMON LISP culls not only from MACLISP but from INTERLISP, other LISP dialects, and other programming languages what we believe from experience to be the most useful and

understandable constructs. Constructs which have proved to be awkward or less useful have been eliminated (an example is the `store` construct of MACLISP).

*Compatibility.* Unless there is a good reason to the contrary, COMMON LISP strives to be compatible with Lisp Machine LISP, MACLISP, and INTERLISP, roughly in that order. Incompatibilities with various LISP dialects or other languages are noted here in the text in specially marked paragraphs.

*Efficiency.* COMMON LISP has a number of features designed to facilitate the production of high-quality compiled code in those implementations which care to invest effort in an optimizing compiler. At least one implementation of COMMON LISP will have such a compiler. This extends the work done in MACLISP to produce extremely efficient numerical code.

*Stability.* It is intended that COMMON LISP, once defined and agreed upon, will change only slowly and with due deliberation. The various dialects which are supersets of COMMON LISP may serve as laboratories within which to test language extensions, but such extensions will be added to COMMON LISP only after careful examination and experimentation.

The COMMON LISP documentation is divided into four parts, known for now as the blue pages, the white pages, the yellow pages, and the red pages. (This document is the white pages.)

- The *blue pages* constitute an implementation guide in the spirit of the INTERLISP Virtual Machine specification [1]. It specifies a subset of the white pages which an implementor must construct, and indicates a quantity of LISP code written in that subset which implements the remainder of the white pages. In principle there could be more than one set of blue pages, each with a companion file of LISP code. (For example, one might assume `if` to be primitive and define `cond` as a macro in terms of `if`, while another might do it the other way around.)
- The *white pages* (this document) is a language specification rather than an implementation specification. It defines a set of standard language concepts and constructs which may be used for communication of data structures and algorithms in the COMMON LISP dialect. This is sometimes referred to as the "core COMMON LISP language", because it contains conceptually necessary or important features. It is not necessarily implementationally minimal. While some features could be defined in terms of others by writing LISP code (and indeed may be implemented that way), it was felt that these features should be conceptually primitive so that there might be agreement among all users as to their usage. (For example, bignums and rational numbers could be implemented as LISP code given operations on fixnums. However, it is important to the conceptual integrity of the language that they be regarded by the user as primitive, and they are useful enough to warrant a standard definition.)
- The *yellow pages* is a program library document, containing documentation for assorted and relatively independent packages of code. While the white pages are to be relatively stable, the yellow pages are extensible; new programs of sufficient usefulness and quality will routinely be added from time to time. The primary advantage of the division into white and yellow pages is this relative stability; a package written solely in the white-pages language should not break if changes are made to the yellow-pages library.
- The *red pages* is implementation-dependent documentation; there will be one set for each

It? —  
lost

implementation. Here are specified such implementation-dependent parameters as word size, maximum array size, sizes of floating-point exponents and fractions, and so on, as well as implementation-dependent functions such as input/output primitives.

## 1.2. Notational Conventions

In COMMON LISP the empty list is written "()", which is not (necessarily) the same as the symbol named "nil". The empty list is, as in most LISP dialects, used to mean "false" in Boolean tests; therefore "false" is also written "()". Any non-() value is treated as being "true". *∴ "False" & "true"*

All numbers in this document are in decimal notation unless there is an explicit indication to the contrary.

Execution of code in LISP is called *evaluation*, because executing a piece of code normally results in a data object called the *value* produced by the code. The symbol "=>" will be used in examples to indicate evaluation. For example:

```
(+ 4 5) => 9
```

means "the result of evaluating the code (+ 4 5) is (or would be, or would have been) 9".

The symbol "==>" will be used in examples to indicate macro expansion. For example:

```
(push x v) ==> (setf v (cons x v))
```

means "the result of expanding the macro-call form (push x v) is (setf v (cons x v))". This implies that the two pieces of code do the same thing; the second piece of code is the definition of what the first does. *f*

The symbol "<=>" will be used in examples to indicate code equivalence. For example:

```
(- x y) <=> (+ x (- y))
```

means "the value and effects of (- x y) is always the same as the value and effects of (+ x (- y)) for any values of the variables x and y". This implies that the two pieces of code do the same thing; however, neither directly defines the other in the way macro-expansion does.

Functions, variables, special forms, and macros are described using a distinctive typographical format. Table 1-1 illustrates the manner in which COMMON LISP functions are documented. The first line specifies the name of the function, the manner in which it accepts arguments, and the fact that it is a function. Following indented paragraphs explain the definition and uses of the function and often present examples or related functions. *named constants*

In general, the ~~text of~~ actual code (including actual names of functions) appears in this typeface: (cons a b). Names which stand for pieces of code (meta-variables) are written in *italics*. In a function description, the names of the parameters appear in italics for expository purposes. The word "optional" in the list of parameters indicates that all arguments past that point are optional; the default values for the parameters are described in the text. Parameter lists may also contain "&rest", indicating that an indefinite number of *&key*

`sample-function arg1 arg2 &optional arg3 arg4` [Function]

The function `sample-function` adds together `arg1` and `arg2`, and then multiplies the result by `arg3`. If `arg3` is not provided or is `()`, the multiplication isn't done. `sample-function` then returns a list whose first element is this result and whose second element is `arg4` (which defaults to the symbol `foo`).

For example:

```
(function-name 3 4) => (7 foo)
(function-name 1 2 2 'bar) => (6 bar)
```

As a rule, `(sample-function x y) <=> (list (+ x y) 'foo)`.

Table 1-1: Sample Function Description

arguments may appear. (The `&optional` and `&rest` syntax is actually used in COMMON LISP function definitions for these purposes.)

`sample-variable` [Variable]

The variable `sample-variable` specifies how many times the special form `sample-special-form` should iterate. The value should always be a non-negative integer or `()` (which means iterate indefinitely many times). The initial value is 0.

Table 1-2: Sample Variable Description

Table 1-2 illustrates the manner in which a global variable is documented. The first line specifies the name of the variable and the fact that it is a variable.

*Symbolic Constants*

Tables `SAMPLE-SPECIAL-FORM-DESCRIPTION` and 1-4 illustrate the documentation of special forms and macros (which are closely related in purpose). Functions are called according to a single, specific, consistent syntax; the `&optional` and `&rest` syntax specifies how the function uses its arguments internally, but does not affect the syntax of a call. In contrast, each special form or macro can have its own idiosyncratic syntax, for it is by special forms and macros that the syntax of COMMON LISP is defined and extended.

*&key*

In the description of a special form or macro, an italicized word names a corresponding part of the form which invokes the special form or macro. Parentheses ("`(`" and "`)`") stand for themselves, and should be written as such when invoking the special form or macro. Square brackets ("`[`" and "`]`") indicate that what they enclose is optional (may appear zero times or one time in that place); the square brackets should not be

---

`sample-special-form` [*name*] (*{var}*\*) *{form}*+ [*Special form*]

This evaluates each form in sequence as an implicit `progn`, and does this as many times as specified by the global variable `sample-variable`. Each variable *var* is bound and initialized to 43 before the first iteration, and unbound after the last iteration. The name *name*, if supplied, may be used in a `return-from` (page 58) form to exit from the loop prematurely. If the loop ends normally, `sample-special-form` returns `()`.

For example:

```
(setq sample-variable 3)
(sample-special-form () form1 form2)
```

This evaluates *form1*, *form2*, *form1*, *form2*, *form1*, *form2* in that order.

Table 1-3: Sample Special Form Description

---

`sample-macro` *var* *{tag | statement}*\* [*Macro*]

This evaluates the statements as a `prog` body, with the variable *var* bound to 43.

```
(sample-macro x (+ x x)) => 86
(sample-macro var . body) ==> (prog ((var 43)) . body)
```

Table 1-4: Sample Macro Description

---

written in code. Curly braces (“{” and “}”) simply parenthesize what they enclose, but may be followed by a star (“\*”) or a plus sign (“+”); a star indicates that what the braces enclose may appear any number of times (including zero, that is, not at all), while a plus sign indicates that what the braces enclose may appear any non-zero number of times (that is, must appear at least once). Within braces or brackets, vertical bars (“|”) separate mutually exclusive choices.

In the last example in Table 1-4, notice the use of “dot notation”. The “.” in `(sample-macro var . body)` means that the name *body* stands for a list of forms, not just a single form, at the end of a list. This notation is often used in examples.

The term “LISP reader” refers not to you, the reader of this document, nor to some person reading LISP code, but specifically to a LISP program (the function `read` (page 197)) which reads characters from an input stream and interprets them by parsing as representations of LISP objects.

Certain characters are used in special ways in the syntax of COMMON LISP. The complete syntax is explained in ???, but a quick summary here may be useful:

- ' An accent acute ("single quote") followed by an expression *form* is an abbreviation for (quote *form*). Thus 'foo means (quote foo) and '(cons 'a 'b) means (quote (cons (quote a) (quote b))).
- ; Semicolon is the comment character. It and everything up to the end of the line is discarded.
- " Double quotes surround character strings. ~~It is always a string~~ see below
- \ Backslash is an escape character. As a rule, it causes the next character to be treated as a letter rather than for its usual syntactic purpose. For example, \A(B denotes a symbol whose name is "A(B", and "\\ " denotes a character string containing one character, a double-quote.
- # The number sign begins a more complex syntax. The next character designates the precise syntax to follow. For example, #o 105 means 105— (105 in octal notation); #\L denotes a character object for the character "L"; and #(a b c) denotes a vector of three elements a, b, and c. #
- | Vertical bars surround the name of a symbol which has special characters in it. — that
- ` Accent grave ("backquote") signals that the next expression is a template which may contain commas. The backquote syntax represents a program which will construct a data structure according to the template.
  - Commas are used within the backquote syntax.
- : Colon is used to indicate which package a symbol belongs to. For example, chaos:reset denotes the symbol named reset in the package named chaos. keywords ✓

All code in this manual is written in lower case. COMMON LISP is generally insensitive to the case in which code is written. Every symbol has a print name which specifies how it is to be capitalized, but the symbol will be recognized even if entered in the wrong case. You may write programs in whichever case you prefer; COMMON LISP will attempt to preserve the capitalization you use. There are ways to force case conversion on input or output. ↗

You will see various symbols that have the colon (:) character in their names. By convention, all "keyword" symbols have names starting with a colon. The colon character is not actually part of the print name, but is a package prefix indicating that the symbol belongs to the keyword package. This is all explained in ???; until you read that, just make believe that the colons are part of the names of the symbols.

"This is a ~~thirty~~ thirty-~~nine~~ nine character string"

internally  
upper case

## Chapter 2

# Data Types

COMMON LISP provides a variety of types of data objects. It is important to note that in LISP it is data objects which are typed, not variables. Any variable can have any LISP object as its value.

In COMMON LISP, a data type is a (possibly infinite) set of LISP objects. Many LISP objects belong to more than one such set, and so it doesn't always make sense to ask what *the* type of an object is; instead, one usually asks only whether an object belongs to a given type. The predicate `typep` (page 28) may be used to ask either of these questions.

The data types defined in COMMON LISP are arranged into an almost-hierarchy (a hierarchy with shared subtrees) defined by the subset relationship. Certain sets of objects are interesting enough to deserve labels (such as the set of numbers or the set of strings). Symbols are used for most such labels (here, and throughout this document, the word *symbol* refers to atomic symbols, one kind of LISP object). The root of the hierarchy, which is the set of all objects, is labelled by `t`. *Also, the empty data type is labelled by ().*

Objects may be roughly divided into the following categories (which are in fact types): number, character, symbol, list, array, structure, function, and random. Some of these categories have many subdivisions. There are also types which are the union of two or more of these categories. The categories listed above, while they are data types, are neither more nor less "real" than other data types; they simply constitute a particularly useful slice across the type hierarchy for expository purposes. *vector*

Each of these categories is described briefly below. Then one section of this chapter is devoted to each, going into more detail, and describing notations for objects of each type. Descriptions of LISP functions which operate on data objects are in later chapters. *global*

- *Numbers* are provided in various forms and representations. COMMON LISP provides a true integer data type: any integer, positive or negative, has in principle a representation as a COMMON LISP data object, subject only to ~~total~~ memory limitations (rather than machine word width). A true rational data type is provided: the quotient of two integers, if not an integer, is a ratio. Floating-point numbers of various ranges and precisions are also provided. Some implementations may choose to provide Cartesian complex numbers. *of the impl*
- *Characters* represent printed glyphs (such as letters) or text formatting operations. Strings are one-dimensional ~~arrays~~ of characters. COMMON LISP provides a rich character set, including ways *(/)*

*vectors*

to represent characters of various type styles.

- *Symbols* (sometimes called *atomic symbols* for emphasis or clarity) are named data objects. LISP provides machinery for locating a symbol object, given its name (in the form of a string). Symbols have *property lists*, which in effect allow symbols to be treated as record structures with an extensible set of named components, each of which may be any LISP object.
- *Lists* are sequences represented in the form of linked cells called *conses*. There is a special object which is the empty list. All other lists are built recursively by adding a new element to the front of an existing list. This is done by creating a new *cons*, which is an object having two components called the *car* and the *cdr*. The *car* may hold anything, and the *cdr* is made to point to the previously existing list. (Conses may actually be used completely generally as two-element record structures, but their most important use is to represent lists.) *there*
- *Arrays* are dimensioned collections of objects. An array can have any non-negative number of dimensions, and is indexed by a sequence of integers. General arrays can have any LISP object as a component; others are specialized for efficiency or for other reasons, and can hold only certain types of LISP objects. Two important special cases of arrays are *strings*, which are one-dimensional arrays of characters, and *bit-strings*, which are one-dimensional arrays that can contain only the integers 0 and 1. *and may be linked together to form graphs,*
- *Structures* are user-defined record structures, objects which have named components. The *defstruct* (page 167) facility is used to define new structure types. Some COMMON LISP implementations may choose to implement certain system-supplied data types as structures; these might include *bignums*, *readtables*, and *streams*. *an ordered Vectors*
- *Functions* are objects which can be invoked as procedures; these may take arguments, and return values. (All LISP procedures can be construed to return a value, and therefore treated as functions. Those which have nothing better to return usually return ( ).) *there* *can be*
- *Random* objects are those which do not fit into any other category. This is a catch-all data type which primarily covers implementation-dependent objects for internal use. *there*

## 2.1. Numbers

There are several kinds of numbers defined in COMMON LISP. Table 2-1 shows the hierarchy of number types.

### 2.1.1. Integers

The *integer* data type is intended to represent mathematical integers. Unlike most programming languages, COMMON LISP in principle imposes no limit on the magnitude of an integer; storage is automatically allocated as necessary to represent large integers.

In every COMMON LISP implementation there is a range of integers which are represented more efficiently than others; each such integer is called a *fixnum*, and an integer which is not a *fixnum* is called a *bignum*. The distinction between *fixnums* and *bignums* is visible to the user in only a few places where the efficiency of

```

number
  rational
    integer
      fixnum
      bignum
    ratio
  float
    short-float
    single-float
    double-float
    long-float
  complex

```

Table 2-1: Hierarchy of Numeric Types

representation is important; in particular, it is guaranteed that any dimension of an array (and therefore any index into an array) can be represented as a fixnum. Exactly which integers are fixnums is implementation-dependent; typically they will be those integers in the range  $-2^n$  to  $2^n - 1$ , inclusive, for some  $n$  not less than 15.

**Implementation note:** In the PERQ implementation of COMMON LISP, fixnums are those integers in the range  $[-2^{27}, 2^{27} - 1]$ . In the S-1 implementation, fixnums are those integers in the range  $[-2^{31}, 2^{31} - 1]$ . In the VAX implementation, fixnums are those integers in the range  $[-2^{29}, 2^{29} - 1]$ .

flush

Integers are ordinarily written in decimal notation, as a sequence of decimal digits, optionally preceded by a sign and optionally followed by a decimal point.

For example:

```

0      ; Zero.
-0     ; This always means the same as 0.
+6     ; The first perfect number.
28     ; The second perfect number.
1024.  ; Two to the tenth power.
-1     ;  $e^{\pi i}$ 
1551121004333098598400000. ; 25 factorial (25!). Probably a bignum.

```

**Compatibility note:** MACLISP and Lisp Machine LISP normally assume that integers are written in *octal* (radix-8) notation unless a decimal point is present. INTERLISP assumes integers are written in decimal notation, and uses a trailing Q to indicate octal radix; however, a decimal point, even in trailing position, *always* indicates a floating-point number. This is of course consistent with FORTRAN; ADA does not permit trailing decimal points, but instead requires them to be embedded. In COMMON LISP, integers written as described above are always construed to be in decimal notation, whether or not the decimal point is present; allowing the decimal point to be present permits compatibility with MACLISP.

There are special ways to notate integers in radices other than ten. The notation

`#nr dddd`

means the integer in radix- $n$  notation denoted by the digits `dddd`. More precisely, one may write “#”, a non-empty sequence of decimal digits representing an unsigned decimal integer  $n$ , “r” (or “R”), an optional sign, and a sequence of radix- $n$  digits, to indicate an integer written in radix  $n$ . Only legal digits for the specified radix may be used; for example, an octal number may contain only the digits 0 through 7. Letters of

the alphabet of either case may be used in order for digits above 9. Binary, octal, and hexadecimal radices are useful enough to warrant the special abbreviations “#b” for “#2r”, “#o” for “#8r”, and “#x” for “#16r”.

For example:

*Safe: lower case #b, #o, #x to prevent confusion.*

```
#2r11010101 ; Another way of writing 213 decimal.
#b11010101  ; Ditto.
#b+11010101 ; Ditto.
#o325       ; Ditto, in octal radix.
#xD5        ; Ditto, in hexadecimal radix.
#16r+D5     ; Ditto.
#o-300      ; Decimal -192, written in base 8.
#3r-12010   ; Same thing in base 3.
#25R-7H     ; Same thing in base 25.
```

*Vars [ max-pos-fixnum min-neg-fixnum ]*

## 2.1.2. Floating-point Numbers

Generally speaking, a floating-point number is a (mathematical) rational number of the form  $(-1)^s f 2^{e-p}$ , where  $s$  is a bit (0 or 1), the *sign*;  $p$  is a positive integer, the *precision* (in bits) of the floating-point number;  $f$  is a positive integer between  $2^{p-1}$  and  $2^p-1$  (inclusive), the *fraction* (properly speaking, the fraction is actually  $f/2^p$ ); and  $e$  is an integer, the *exponent*. In addition, there is a floating-point zero. The value of  $p$  and the range of  $e$  depends on the implementation and on the type of floating-point number within that implementation.

Floating-point numbers are provided in a variety of precisions and sizes, depending on the implementation. High-quality floating-point software tends to depend critically on the precise nature of the floating-point arithmetic, and so may not always be completely portable. To aid in writing programs which are moderately portable, however, certain definitions are made here:

- A *short* floating-point number is the representation of smallest precision provided by an implementation.
- A *long* floating-point number is the representation of the largest fixed precision provided by an implementation.
- Intermediate between short and long sizes are two others, arbitrarily called *single* and *double*.

The precise definition of these categories is implementation-dependent. However, the rough intent is that short floating-point numbers be precise at least to about five decimal places; single floating-point numbers, at least to about seven decimal places; and double floating-point numbers, at least to about fourteen decimal places. Therefore the following minimum requirements are imposed on these formats: the precision (in bits) and the exponent size (in bits; that is, the base-2 logarithm of one plus the maximum exponent value) must be at least as great as the values in Table FLOATING-FORMAT-REQUIREMENTS-TABLE.

In any given implementation the categories may overlap or coincide. For example, short might mean the same as single, and long might mean the same as double.

**Implementation note:** Where it is feasible, it is recommended that an implementation provide at least two types of floating-point number, and preferably three. Ideally, short-format floating-point numbers should have an “immediate”

Format	Minimum Precision	Minimum Exponent Size
Short	7 bits	20 bits
Single	8 bits	24 bits
Double	8 bits	50 bits

Table 2-2: Minimum Floating-Point Precision and Exponent Size Requirements

representation that does not require consing, single-format floating-point numbers should approximate IEEE standard single-format floating-point numbers, and double-format floating-point numbers should approximate IEEE standard double-format floating-point numbers.

*Long may be the same as double, or if a yet larger size.*

In the PERQ SPICE LISP implementation of COMMON LISP, two types are to be provided:

- For the small size (28 bits),  $p=20$  and  $e$  is in  $[-128, 127]$ . Short format maps to this.
- For the large size (96 bits),  $p=63$  and  $e$  is in  $[-2^{31}, 2^{31}-1]$ . Single, double, and long formats map to this.

On the S-1, three types are provided:

- For halfword size (18 bits),  $p=13$  and  $e$  is in  $[-15, 16]$ . Short format maps to this.
- For singleword size (36 bits),  $p=27$  and  $e$  is in  $[-255, 256]$ . Single format maps to this.
- For doubleword size (72 bits),  $p=57$  and  $e$  is in  $[-2^{14}+1, 2^{14}]$ . Double and long formats map to this.

The VAX architecture provides four floating-point formats:

- F-floating: 32 bits,  $p=24$ ,  $e$  in  $[-127, 127]$ .
- D-floating: 64 bits,  $p=56$ ,  $e$  in  $[-127, 127]$ .
- G-floating: 64 bits,  $p=53$ ,  $e$  in  $[-1023, 1023]$ .
- H-floating: 128 bits,  $p=113$ ,  $e$  in  $[-16383, 16383]$ .

Probably D-floating format should not be used. If so, then *short* and *single* might refer to F-floating format, *double* to G-floating format, and *long* to H-floating format (if that is supported; if not, then G-floating format). Alternatively, *short* format might be a 28-bit format consisting F-format with the four lowest-order fraction bits removed.

*flush*

Floating point numbers are written in either decimal fraction or "computerized scientific" notation: an optional sign, then a non-empty sequence of digits with an embedded decimal point, then an optional decimal exponent specification. The decimal point is required, and there must be digits either before or after it; moreover, digits are required after the decimal point if there is no exponent specifier. The exponent specifier consists of an exponent marker, an optional sign, and a non-empty sequence of digits. For preciseness, here is an extended-BNF description of floating-point notation. The notation " $\langle x \rangle^*$ " means zero or more occurrences of " $x$ ", the notation " $\langle x \rangle^+$ " means one or more occurrences of " $x$ ", and the notation " $\langle x \rangle^?$ " means zero or one occurrences of " $x$ ".

$\langle \text{floating-point number} \rangle ::= \langle \text{sign} \rangle^? \langle \text{digit} \rangle^* \langle \text{digit} \rangle^+ \langle \text{exponent} \rangle^? | \langle \text{sign} \rangle^? \langle \text{digit} \rangle^+ . \langle \text{digit} \rangle^* \langle \text{exponent} \rangle$   
 $\langle \text{sign} \rangle ::= + | -$   
 $\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$   
 $\langle \text{exponent} \rangle ::= \langle \text{exponent marker} \rangle \langle \text{sign} \rangle^? \langle \text{digit} \rangle^+$   
 $\langle \text{exponent marker} \rangle ::= e | s | f | d | 1 | E | F | D | S | L$

*Use { }, [ ]*

If no exponent specifier is present, or if the exponent marker  $e$  (or  $E$ ) is used, then the precise format to be used is not specified. When such a floating-point number representation is read and converted to an internal floating-point data object, the format specified by the variable `read-default-float-format` (page `READ-DEFAULT-FLOAT-FORMAT-VAR`) is used; the initial value of this variable is `single`.

*why?*

The letters *s*, *f*, *d*, and *l* (or their respective upper-case equivalents) specify explicitly the use of *short*, *single*, *double*, and *long* format, respectively.

??? Query: There has been some objection to the use of the words *single* and *double*, as they may be misleading to the user or too confining for the implementor. Any suggestions?

For example:

0.0	; Floating-point zero in default format.
- .0	; Also a floating-point zero.
0.	; The <i>integer</i> zero, not a floating-point number!
0.0s0	; A floating-point zero in <i>short</i> format.
3.1415926535897932384d0	; A <i>double</i> -format approximation to $\pi$ .
3.1415926535897932384B0	; A <i>big</i> -format approximation to $\pi$ .
6.02E+23	; Avogadro's number, in default format.
3.1010299957f-1	; $\log_2 2$ , in <i>single</i> format.
-0.000000001s9	; $e^{\pi}$ in <i>short</i> format, the hard way.

### 2.1.3. Ratios

The rationals include the integers, and also *ratios* of two integers. The canonical printed representation of a rational number is as an integer if its value is integral, and otherwise as the ratio of two integers, the *numerator* and *denominator*, whose greatest common divisor is one, and of which the denominator is positive (and in fact greater than 1, or else the value would be integral), written with "/" as a separator thus: "3/5". It is possible to notate ratios in non-canonical (unreduced) forms, such as "4/6", but the LISP function `prin1` (page 201) always prints the canonical form for a ratio.

Implementation note: While ~~some~~ implementation<sup>s</sup> of COMMON LISP will probably choose to maintain all ratios in reduced form, there is no requirement for this as long as its effects are not visible to the user. Note that while it may at first glance appear to save computation for the reader and various arithmetic operations not to have to produce reduced forms, this savings is likely to be counteracted by the increased cost of operating on larger numerators and denominators.

Rational numbers may be written as the possibly signed quotient of decimal numerals: an optional sign followed by two non-empty sequences of digits separated by a "/". The second sequence may not consist entirely of zeros.

For example:

2/3	; This is in canonical form.
4/6	; A non-canonical form for the same number.
-17/23	
-30517578125/32768	; This is $(-5/2)^{15}$ .
10/5	; The canonical form for this is 2.

There are ways to notate rational numbers in radices other than ten; one uses the same radix specifiers (one of #nnR, #O, #B, or #X) as for integers.

For example:

#o-101/75	; Octal notation for -65/61.
#3r120/21	; Ternary notation for 15/7.
#Xbc/ad	; Hexadecimal notation for 188/173.

## 2.1.4. Complex Numbers

Complex numbers may or may not be supported by a COMMON LISP implementation. They are represented in Cartesian form, with a real part and an imaginary part each of which is a non-complex number (integer, floating-point number, or ratio). It should be emphasized that the parts of a complex number are not necessarily floating-point numbers; in this COMMON LISP is like PL/I and differs from FORTRAN. In general, these identities hold:

```
(eql (realpart (complex x y)) x)
(eql (imagpart (complex x y)) y)
```

Complex numbers may be notated by writing the characters "#C" followed by a list of the real and imaginary parts. (Indeed, "#C(*a b*)" is equivalent to "#,(complex *a b*)"; see the description of the function `complex` (page 130).)

For example:

```
#C(3.0s1 2.0s-1)
#C(5 -3)
#C(5/3 7.0)
```

*Query J notation.* —

; A Gaussian integer.

Some implementations furthermore provide specialized representations of complex numbers for efficiency. In such representations the real part and imaginary part are of the same specialized numeric type. The "#C" construct will produce the most specialized representation which will correctly represent the two notated parts. The type of a specialized complex number is indicated by a list of the word `complex` and the type of the components; for example, a specialized representation for complex numbers with short floating-point parts would be of type `(complex short-float)`. The type `complex` encompasses all complex representations; the particular representation which allows parts of any numeric type is referred to as type `(complex t)`.

## 2.2. Characters

Every character object has three attributes: *code*, *bits*, and *font*. The code attribute is intended to distinguish among the printed glyphs and formatting functions for characters. The bits attribute allows extra flags to be associated with a character. The font attribute permits a specification of the style of the glyphs (such as italics). Each of these attributes may be understood to be a non-negative integer.

A character object can be notated by writing "#\*\*" followed by the character itself. For example, "#\*g*" means the character object for a lower-case "g". This works well enough for "printing characters". Non-printing characters have names, and can be notated by writing "#\*\*" and then the name; for example, "#\*\return*" (or "#\*\RETURN*" or "#\*\Return*", for example) means the <return> character. The syntax for character names after "#\*\*" is the same as that for symbols.

The font attribute may be notated in unsigned decimal notation between the "#" and the "\". For example, #3\*\A* means the letter "A" in font 3. Note that not all COMMON LISP implementations provide for non-zero font attributes; see `char-font-limit` (page 139).

*Tie this to a FEATURE?*

*Query J notation.* —

*Query: float font.* —

The bits attribute may be notated by preceding the name of the character by the names or initials of the bits, separated by hyphens. The character itself may be written instead of the name, preceded if necessary by “\”. For example:

```
#\Control-Meta-Return
#\Hyper-Space
#\Control-A
#\Meta-\beta
#\C-M-Return
```

Note that not all COMMON LISP implementations provide for non-zero bits attributes; see `char-font-limit` (page 139).

Any character whose bits and font attributes are zero may be contained in strings. All such characters together constitute a subtype of the characters; this subtype is called `string-char`.

## 2.3. Symbols

Symbols are LISP data objects which serve several purposes and several interesting properties. Every symbol has a name, called its *print name*, or *pname*. Given a symbol, one can obtain its name in the form of a string. More interesting, given the name of a symbol as a string one can obtain the symbol itself. (More precisely, symbols are organized into *packages*, and all the symbols in a package are uniquely identified by name.)

Symbols have a component called the *property list*, or *plist*. By convention this is always a list whose even-numbered components (calling the initial one component zero) are symbols, here functioning as property names, and whose odd-numbered components are associated property values. Functions are provided for manipulating this property list; in effect, these allow a symbol to be treated as an extensible record structure.

Symbols are also used to represent certain kinds of variables in LISP programs, and there are functions for dealing with the values associated with symbols in this role.

A symbol can be notated simply by writing its name. If its name is not empty, and if the name consists only of upper-case alphabetic, numeric, or certain “pseudo-alphabetic” special characters (but not delimiter characters such as parentheses or space), and if the name of the symbol cannot be mistaken for a number, then the symbol can be notated by the sequence of characters in its name.

For example:

FROBBOZ	;The symbol whose name is "FROBBOZ".
frobboz	;Another way to notate the same symbol.
fRObBoz	;Yet another way to notate it.
unwind-protect	;A symbol with a "-" in its name.
+\$	;The symbol named "+\$".
1+	;The symbol named "1+".
+1	;This is the integer 1, not a symbol.
pascal_style	;This symbol has an underscore in its name.
b^2-4*a*c	;This is a single symbol!
	; It has several special characters in its name.
file.rel.43	;This symbol has periods in its name.
/usr/games/zork	;This symbol has slashes in its name.

Besides letters and numbers, the following characters are normally considered to be "alphabetic" for the purposes of notating symbols:

+ - \* / ! @ \$ % ^ & ← = < > ? ~ .

Some of these characters have conventional purposes for naming things; for example, symbols which name functions having extremely implementation-dependent semantics generally have names beginning with "%". The last character, ".", is considered alphabetic *provided* that it does not stand alone. By itself, it has a role in the notation of conses. (It also serves as the decimal point.)

A symbol may have upper-case letters, lower-case letters, or both in its print name. However, the LISP reader normally converts lower-case letters to the corresponding upper-case letters when reading symbols. The net effect is that most of the time case makes no difference when *notating* symbols. However, case *does* make a difference internally and when printing a symbol. Internally the symbols which name all standard COMMON LISP functions, variables, and keywords have upper-case names; their names appear in lower case in this document for readability. Typing such names in lower case works because the function read will convert them to upper case.

If a symbol cannot be notated simply by the characters of its name, because the (internal) name contains special characters or lower-case letters, then there are two "escape" conventions for notating them. Writing a "\" character before any character causes the character to be treated itself as an ordinary character for use in a symbol name. If any character in a notation is preceded by \, then that notation can never be interpreted as a number.

For example:

\(	;The symbol whose name is "(".
\+1	;The symbol whose name is "+1".
+\1	;Also the symbol whose name is "+1".
\frobboz	;The symbol whose name is "fROBBOZ".
3.14159265\s0	;The symbol whose name is "3.14159265s0".
3.14159265\S0	;The symbol whose name is "3.14159265S0".
3.14159265s0	;A short-format floating-point approximation to $\pi$ .
APL\360	;The symbol whose name is "APL\360".
\(b^2)\ -\ 4*a*c	;The name is "(b^2) - 4*a*c".
	; It has parentheses and two spaces in it.

It may be tedious to insert a "\" before *every* delimiter character in the name of a symbol if there are many

of them. An alternative convention is to surround the name of a symbol with vertical bars; these cause every character between them to be taken as part of the symbol's name, as if "\ " had been written before each one, excepting only | itself and \, which must nevertheless be preceded by \.

For example:

```

|" |           ;The same as writing \".
|(b^2) - 4*a*c| ;The name is "(b^2) - 4*a*c".
|frobboz|      ;The name is "frobboz", not "FROBBOZ".
|APL\360|      ;The name is "APL360", because
               ; the "\" quotes the "3".
|APL\\360|     ;The name is "APL\360".
|ap1\\360|     ;The name is "ap1\360".
|\|\\|        ;Same as \| \| the name is "||".

```

## 2.4. Lists and Conses

A *cons* is a little record structure containing two components, called the *car* and the *cdr*. Conses are used primarily to represent lists.

A *list* is recursively defined to be either the empty list, which is a special data object notated as "()", or a cons whose *cdr* component is a list. A list is therefore a chain of conses linked by their *cdr* components and terminated by (). The *car* components of the conses are called the *elements* of the list. For each element of the list there is a cons. The empty list has no elements at all.

A list is notated by writing the elements of the list in order, separated by blank space (space, tab, or return characters) and surrounded by parentheses.

For example:

```

(a b c)       ;A list of three symbols.
(2.0s0 (a 1) #\*) ;A list of three things: a short floating-point number,
                ; another list, and a character object.

```

This is why the empty list is written as (); it is a list with no elements.

A *dotted list* is one whose last cons does not have () for its *cdr*, but some other data object (which is also not a cons, or the first-mentioned cons would not be the last cons of the list). Such a list is called "dotted" because of the special notation used for it: the elements of the list are written between parentheses as before, but after the last element and before the right parenthesis are written a dot (surrounded by blank space) and then the *cdr* of the last cons. As a special case, a single cons is notated by writing the *car* and the *cdr* between parentheses and separated by a space-surrounded dot.

For example:

```

(a . 4)       ;A cons whose car is a symbol
               ; and whose cdr is an integer.
(a b c . d)   ;A list with three elements whose last cons
               ; has the symbol d in its cdr.

```

Compatibility note: In MACLISP, the dot in dotted-list notation needed not be surrounded by white space or other delimiters.

The dot is required to be delimited in Lisp Machine LISP.

It is legitimate to write something like (a b . (c d)); this means the same as (a b c d). The standard LISP output routines will never print a list in the first form, however; they will avoid dot notation wherever possible.

Often the term *list* is used to refer either to true lists or to dotted lists. The term "true list" will be used to refer to a list terminated by ( ), when the distinction is important. Most functions advertised to operate on lists ~~will work on dotted lists and ignore the non-( ) cdr at the end.~~ *nominally require dotted lists.*

Sometimes the term *tree* is used to refer to some cons and all the other conses transitively accessible to it through *car* and *cdr* links until non-conses are reached; these non-conses are called the *leaves* of the tree.

*Non-conses are also considered to be (trivial) trees.*

Lists, dotted lists, and trees are not mutually exclusive data types; they are simply useful points of view about structures of conses. There are yet other terms, such as *association list*. None of these are true LISP data types. Conses are a data type, and ( ) is the sole object of type null. The LISP data type *list* is taken to mean the union of the cons and null data types, and therefore encompasses both true lists and dotted lists.

## 2.5. Arrays

An *array* is an object with components arranged according to a rectilinear coordinate system. In general, these components may be any LISP data objects.

The number of dimensions of an array is called its *rank* (this terminology is borrowed from APL). This is a non-negative integer; for convenience, it is in fact required to be a fixnum (an integer of limited magnitude). Likewise, each dimension has a length which is a non-negative fixnum. The total number of elements in the array is the product of all the dimensions.

*Query: impls limit rank? —*

It is permissible for a dimension to be zero. In this case, the array has no elements, and any attempt to access an element in error. However, other properties of the array (such as the dimensions themselves) may be used. If the rank is zero, then there are no dimensions, and the product of the dimensions is then by definition 1. A zero-rank array therefore has a single element.

An array element is specified by a sequence of indices. The length of the sequence must equal the rank of the array. Each index must be a non-negative integer strictly less than the corresponding array dimension. Array indexing is therefore zero-origin, not one-origin as in (the default case of) FORTRAN.

As an example, suppose that the variable *foo* names a 3-by-5 array. Then the first index may be 0, 1, or 2, and then second index may be 0, 1, 2, 3, or 4. One may refer to array elements using the function *aref* (page 160):

```
(aref foo 2 1)
```

refers to element (2,1) of the array. Note that *aref* takes a variable number of arguments: an array, and as many indices as the array has dimensions. A zero-rank array has no dimensions, and therefore *aref* would

take such an array and no indices, and return the sole element of the array.

One-dimensional arrays and lists are collectively considered to be *sequences*. They differ in that any component of a one-dimensional array can be accessed in constant time, while the average component access time for a list is linear in the length of the list; on the other hand, adding a new element to the front of a list takes constant time, while the same operation on an array takes time linear in the length of the array.

In general, arrays can be multi-dimensional, can have *fill pointers*, can share their contents with other array objects, and can have their size altered dynamically after creation. The important special case of a one-dimensional array with no fill pointer, unshared with any other array, and of unincreasable size is called a *vector*. Some implementations can handle vectors in an especially efficient manner.

The general notation for arrays is rather complicated. It generally begins with "#nA", where *n* is the rank of the array, and is followed by a description of the contents of the array. The notation is described in full in ???.

A general vector (a one-dimensional array of S-expressions with no additional paraphernalia) can be notated by notating the components in order, separated by whitespace and surrounded by "#(" and ")".

For example:

```
#(a b c)           ; A vector of length 3.
#(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
; A vector containing the primes below 50.
#()               ; An empty vector.
```

Rationale: Numerous people have suggested that square brackets be used to notate vectors, "[a b c]" instead of "#(a b c)". This would be shorter, perhaps more readable, and certainly in accord with cultural conventions in other parts of computer science and mathematics. However, to preserve the usefulness of the user-definable macro-character feature of the function `read` (page 197), it is necessary to leave some characters to the user for this purpose. Experience in MacLisp has shown that users, especially implementors of AI languages, often want to define special kinds of brackets. Therefore COMMON LISP avoids using these characters in its syntax so that the user may freely redefine their syntax: "[ ] { } ! ?".

Implementations may provide certain specialized representations of arrays for efficiency in the case where all the components are of the same specialized (typically numeric) type. All implementations provide specialized arrays for the cases when the components are characters or when the components are always 0 or 1; the one-dimensional instances of these specializations are respectively called *strings* and *bit-strings*. Special notations are provided for the further restriction of these types to the vector case. A string vector can be written as the sequence of characters contained in the string, preceded and followed by a "" (double-quote) character. Any "" or "\ character in the sequence must additionally have a "\ character before it.

For example:

```
"Foo"           ; A string with three characters in it.
""              ; An empty string.
"\APL\360?\\" he cried." ; A string with twenty characters.
"|x| = |-x|"    ; A ten-character string.
```

Notice that any vertical bar "|" in a string need not be preceded by a "\". Similarly, any double-quote in the name of a symbol written using vertical-bar notation need not be preceded by a "\". The double-quote and vertical-bar notations are similar but distinct: double-quotes indicate a character string containing the

sequence of characters, while vertical bars indicate a symbol whose name is the contained sequence of characters.

A bit-string vector is written much like a string, using double-quotes; however, a “#” is written before it, and the elements of the bit vector must be 0 or 1.

For example:

```
#" 10110"           ; A bit vector with five bits. Bit 0 is 1.
#" "               ; A null bit vector.
#" 110101000101000101" ; Bit n of this bit vector is 1 iff n+2 is prime.
```

## 2.6. Structures

Different structures may print out in different ways; the definition of a structure type may specify a print procedure to use for objects of that type (see the `:printer` (page DEFSTRUCT-PRINTER-KWD) option to `defstruct` (page 167)). The default notation for structures is:

```
#S(structure-name
    slot-name-1 slot-value-1
    slot-name-2 slot-value-2
    ...)
```

where “#S” indicates structure syntax, *structure-name* is the name (a symbol) of the structure type, each *slot-name* is the name (also a symbol) of a component, and each corresponding *slot-value* is the representation of the LISP object in that slot.

## 2.7. Functions

## 2.8. Randoms

Objects of type `random` tend to have implementation-dependent semantics, and so may print in implementation-dependent ways. As a rule, such objects cannot reliably be reconstructed from a printed representation, and so they are printed usually in a format informative to the user but not acceptable to the `read` function:

```
#<useful information>
```

A hypothetical example might be:

```
#<stack-pointer si:rename-within-new-definition-maybe 311037552>
```

The LISP reader will signal an error on encountering “#<”.

It is not necessarily the case that all objects which are printed in the form “#<...>” are of type `random`; however, any object of type `random` will be printed in that form.



## Chapter 3

# Program Structure

In the previous chapter the syntax was sketched for notating data objects in COMMON LISP. The same syntax is used for notating programs, because all COMMON LISP programs have a representation as COMMON LISP data objects.

### 3.1. Forms

The standard unit of interaction with a COMMON LISP implementation is the *form*, which is simply an S-expression meant to be *evaluated* as a program to produce ~~one or more~~ *values* (which are also data objects). One may request evaluation of *any* data object, but only certain ones (such as symbols and lists) are meaningful forms, while others (such as most arrays) are not. Examples of meaningful forms are 3, whose value is 3, and (+ 3 4), whose value is 7. We write "3 => 3" and "(+ 3 4) => 7" to indicate these facts ("=>" means "evaluates to").

Meaningful forms may be divided into three categories: self-evaluating forms, such as numbers; symbols, which stand for variables; and lists. The lists in turn may be divided into three categories: special forms, macro calls, and function calls.

#### 3.1.1. Self-Evaluating Forms

All numbers, strings, and bit-strings are *self-evaluating* forms. When such an object is evaluated form, that object itself (or possibly a copy in the case of numbers) is returned as the value of the form. The empty list () is also a self-evaluating form: the value of () is ().

Should all vectors self-eval?

#### 3.1.2. Variables

Symbols are used as names of variables in COMMON LISP programs. When a symbol is evaluated as a form, the value of the variable it names is produced. For example, after doing (setq items 3), which assigns the value 3 to the variable named items, then items => 3. Variables can be *assigned* to (as by `setq` (page 39)) or *bound*. Any program construct that binds a variable effectively saves the old value of the variable and causes it to have a new value, and on exit from the construct the old value is reinstated.

There are actually two kinds of variables in COMMON LISP, called *lexical* (or *static*) variables and *special* (or

should one speak of one variable (i.e., symbol) as having two kinds of values?

dynamic) variables. At any given time either or both kinds of variable with the same name may have a current value. Which of the two kinds of variable is referred to when a symbol is evaluated depends on the context of the evaluation. The general rule is that if the symbol occurs textually within a program construct that creates a binding for a variable of the same name, then the reference is to the kind of variable specified by the binding; if no such program construct textually contains the reference, then it is taken to refer to the special variable of that name.

The distinction between the two kinds of variable is one of scope and access. A lexically bound variable can be referred to *only* by forms occurring at any *place* textually within the program construct that binds the variable. A dynamically bound (special) variable can be referred to at any *time* from the time the binding is made until the time evaluation of the construct that binds the variable terminates. Therefore lexical binding imposes spatial limitations on occurrences of references, whereas dynamic binding imposes temporal limitations.

The value a special variable has when there are currently no bindings of that variable is called the *global* value of the variable. A global value can be given to a variable only by assignment, because a value given by binding by definition is not global.

The symbols `t` and `nil` are reserved. One may not assign a value to `t` or `nil`, and one may not bind `t` or `nil`. The global value of `t` is always a true (non-()) value, and the global value of `nil` is always false (()). Constant symbols defined by `defconst` (page 23) also become reserved and may not be further assigned to or bound.

Rationale: It would seem appropriate for the compiler to be justified in issuing a warning if one does a `setq` on a constant defined by `defconst`. If one cannot assign, one should not be able to bind, either.

Examples

??? Query: Unfortunately this violates the principle of alpha-conversion for lexical variables. What would people think of forbidding only *special* bindings of `t` and `nil`, but permitting lexical bindings (and *caveat* the hacker who writes

```
(defun time-dependent-fn (t)
  (cond ((plusp t) 5) (t -5)))
```

or something similar. Actually, that particular one would probably work anyway...)? This allows someone using lexical variables to use any name without interference from the rest of the world, and also prevents anyone from clobbering global variables such as `t`, on the constancy of which other functions depend.

Other vars described as constants, and

### 3.1.3. Special Forms

If a list is to be evaluated as a form, the first step is to examine the first element of the list. If the first element is one of the symbols appearing in Table SPECIAL-FORM-TABLE, then the list is called a *special form*. (This use of the word "special" is unrelated to its use in the phrase "special variable".)

(The page numbers indicate where the definitions of these special forms appear.)

critical

use

Table 3-1: Names of All COMMON LISP Special Forms

Special forms are generally environment and control constructs. Every special form has its own idiosyncratic syntax. An example is the `if` special form: “(if *p* (+ *x* 4) 5)” in COMMON LISP means what “if *p* then *x*+4 else 5” would mean in ALGOL.

The evaluation of a special form normally produces a value (but it may instead call for a non-local exit (see `throw` (page 65)) or produce no values or more than one value (see `values` (page 59))).

The set of special forms is fixed in COMMON LISP; no way is provided for the user to define more. The user can create new syntactic constructs, however, by defining macros.

### 3.1.4. Macros

If a form is a list and the first element is not the name of a special form, it may be the name of a *macro*; if so, the form is said to be a *macro call*. A macro is essentially a function from forms to forms that will, given a call to that macro, compute a new form to be evaluated in place of the macro call. (This computation is sometimes referred to as *macro expansion*.) For example, the macro named `push` (page 92) will take a form such as `(push x stack)` and from that form compute a new form `(setf stack (cons x stack))`. We say that the old form *expands* into the new form. The new form is then evaluated in place of the original form; the value of the new form is returned as the value of the original form.

There are a number of standard macros in COMMON LISP, and the user can define more by using `defmacro` (page DEFMACRO-FUN).

### 3.1.5. Function Calls

If a list is to be evaluated as a form and the first element is not a symbol that names a special form or macro, then the list is assumed to be a *function call*. The first element of the list is taken to name a function. Any and all remaining elements of the list are forms to be evaluated; one value is obtained from each form, and these values become the *arguments* to the function. The function is then *applied* to the arguments. The functional computation normally produces a value (but it may instead call for a non-local exit (see `throw` (page 65)) or produce no values or more than one value (see `values` (page 59))). If and when the function returns, whatever value(s) it returns ~~becomes~~ the value(s) of the function-call form.

For example, consider the evaluation of the form `(+ 3 (* 4 5))`. The symbol `+` names the addition function, not a special form or macro. Therefore the two forms `3` and `(* 4 5)` are evaluated to produce arguments. The form `3` evaluates to 3, and the form `(* 4 5)` is a function call (to the multiplication function). Therefore the forms `4` and `5` are evaluated, producing arguments 4 and 5 for the multiplication. The multiplication function calculates the number 20 and returns it. The values 3 and 20 are then given as arguments to the addition function, which calculates and returns the number 23. Therefore we say `(+ 3 (* 4 5)) => 23`.

Conventions in this doc about fns, macros, & special forms?

### 3.2. Functions

There are two ways to indicate a function to be used in a function call form. One is to use a symbol that names the function. This use of symbols to name functions is completely independent of their use in naming special and lexical variables. The other way is to use a *lambda-expression*, which is a list whose first element is the symbol `lambda`. A *lambda-expression* is not a form; it cannot be meaningfully evaluated. Lambda-expressions and symbols as names of functions can appear only as the first element of a function-call form, or as the second element of the `function` (page 38) special form.

*form*

*stat*

#### 3.2.1. Named Functions

A name can be given to a function in one of two ways. A *global name* can be given to a function by using the `defun` (page DEFUN-FUN) special form. A *local name* can be given to a function by using the `labels` (page LABELS-FUN) special form. If a symbol appears as the first element of a function-call form, then it refers to the definition established by the innermost `labels` construct that textually contains the reference, or if to the global definition (if any) if there is no such containing `labels` construct.

When a function is named, a lambda-expression is associated with that name (in effect). See `defun` (page DEFUN-FUN) and `labels` (page LABELS-FUN) for an explanation of these lambda-expressions.

#### 3.2.2. Lambda-Expressions

A *lambda-expression* is a list with the following syntax:

```
(lambda lambda-list . body)
```

The first element must be the symbol `lambda`. The second element must be a list. It is called the *lambda-list*, and specifies names for the *parameters* of the function. When the function denoted by the lambda-expression is applied to arguments, the arguments are matched with the parameters specified by the lambda-list. The *body* may then refer to the arguments by using the parameter names. The *body* consists of any number of forms (possibly zero). These forms are evaluated in sequence, and the value(s) of the *last* form only are returned as the value(s) of the application (the empty list `()` is returned if there are zero forms in the body).

*Problem with non-bindability of keyword symbols.*

The complete syntax of a lambda-expression is:

```
(lambda ({var}*
  {[&optional {var | (var [initform [svar]])}*] [&rest var]
  | [&key {var | ((keyword var))}*]
  [optional {var | (var | (keyword var)) [initform [svar]]}*]
  [&aux {var | (var [initform])}*]
  {form}*))
```

*{(declare {declaration})\*}*

*are symbols whose names*

Each element of a lambda-list is either a *parameter specifier* or a *separator token*; separator tokens begin with "&". In all cases *var* must be a symbol, the name of a variable, and similarly for *svar*; *keyword* must be a keyword symbol. An *initform* may be any form.

A lambda-list has three parts, any or all of which may be empty:

- Specifiers for the *required* parameters. These are all the parameter specifiers up to the first separator token; if there is no such token, then all the specifiers are for required parameters.
- Either *optional* and *rest* parameters or *keyword* parameters (but not both).
  - If the token `&key` is not present, then the *optional* parameter specifiers are those following the token `&optional` up to the next separator token or the end of the list. Following the optional parameter specifiers may be a single *rest* parameter specifier preceded by the token `&rest`.
  - If the token `&key` is present, all specifiers up to the `&aux` token or the end of the list are *keyword* parameter specifiers. Any appearing after the token `&optional` are *optional keyword* parameters; all others are *required keyword* parameters.
- If the token `&aux` is present, all specifiers after it are *auxiliary variable* specifiers.

Awkward?

Should all  
kws  
be  
optional?

Sometimes *required* and *optional* parameters are called *required positional* parameters and *optional positional* parameters to contrast them explicitly with required and optional keyword parameters.

Compatibility note: What is provided here is a subset of the functionality currently provided in Lisp Machine LISP. The principal restrictions here are:

- Keyword parameters may not be mixed with positional optional and rest parameters. The rationale for not mixing keyword parameters and positional optionals is that it would be very awkward to define a function in such a way that one could not specify any keyword parameters unless all positional optionals were specified. If the positional ones are to be non-trivially optional, then all the keyword parameters should also be optional, and as a matter of style it would be better for all the optional parameters to have keywords. (We know how to make interleaved required and optional positional parameters work, too, but as a matter of style we only allow optionals to follow required.) The rationale for not mixing keyword and rest parameters is less strong, and motivated primarily by a feeling of awkwardness in letting more than one parameter receive the same argument. If we allow that, then why not (`&rest x a b &optional c d`)?
- No keyword argument may be provided for which there is no matching keyword parameter. This is a logical consequence of not mixing keyword and rest parameters, and also greatly improves program readability: the lambda-list enumerates all relevant keywords. Is non-trivial use made of `&allow-extra-keywords` in Lisp Machine LISP?
- The user may specify that some keyword arguments are required while other are optional, instead of all of them being optional. The implementor is encouraged to error-check, of course.

How do people feel about this? Lisp Machine LISP will run correct programs constructed according to the above specifications (modulo accepting `&optional` after `&key`), but may want to error-check required keyword arguments.

When the function represented by the lambda-expression is applied to arguments, the arguments and parameters are processed in order from left to right. In the simplest case, only required parameters are present in the lambda-list; each is specified simply by a name *var* for the parameter variable. When the function is applied, there must be exactly as many arguments as there are parameters, and each parameter is bound to one argument. Here, and in general, the parameter is bound as a lexical variable unless a declaration has been made that it should be a special binding (see `declare` (page 72)).

In the more general case, if there are  $n$  required parameters ( $n$  may be zero), there must be at least  $n$  arguments, and the required parameters are bound to the first  $n$  arguments. The other parameters are then processed using any remaining arguments.

If *optional* parameters are specified, then each one is processed as follows. If any unprocessed arguments remain, then the parameter variable *var* is bound to the next remaining argument, just as for a required parameter. If no arguments remain, however, the *initform* part of the parameter specifier is evaluated, and the parameter variable is bound to the resulting value (or to `()` if no *initform* appears in the parameter specifier). If another variable name *svar* appears in the specifier, it is bound to *true* if an argument was available, and to *false* if no argument remained (and therefore *initform* had to be evaluated). The variable *svar* is called a *supplied-p* parameter; it is not bound to an argument, but to a value indicating whether or not an argument had been supplied for another parameter.

After all *optional* parameter specifiers have been processed, then there may or may not be a *rest* parameter. If there is none, then there should be no unprocessed arguments (it is an error if there are). If there is a *rest* parameter, it is bound to a list of all as-yet-unprocessed arguments. (If no unprocessed arguments remain, the *rest* parameter is bound to `()`.)

Instead of *optional* and *rest* parameters, *keyword* parameters may be specified instead. In that case, after all required parameters (and an equal number of arguments) have been processed, there must remain an even number of arguments; these are processed in pairs, the first argument in each pair being interpreted as a keyword name and the second as the corresponding value. No two argument pairs should have the same keyword name.

In each keyword parameter specifier must be a name *var* for the parameter variable. If an explicit *keyword* is specified, that is the keyword name for the parameter; otherwise the name *var* serves also as the keyword name. For each keyword parameter specifier, if there is an argument pair whose keyword name matches that specifier's keyword name, then the parameter variable for that specifier is bound to the second item (the value) of that argument pair. If no such argument pair exists, then it is an error unless the keyword parameter specifier occurs after the `&optional` token, in which case the *initform* for that specifier is evaluated and the parameter variable is bound to that value (or to `()` if no *initform* was specified). The variable *svar* is treated as for ordinary *optional* parameters: it is bound to *true* if there was a matching argument pair, and to *false* otherwise. It is an error if an argument pair has a keyword name not matched by any parameter specifier.

After all parameter specifiers have been processed, the auxiliary variable specifiers (those following the token `&aux`) are processed from left to right. For each one the *initform* is evaluated and the variable *var* bound to that value (or to `()` if no *initform* was specified). (Nothing can be done with `&aux` variables that cannot be done with the special form `let` (page 43). Which to use is purely a matter of style.)

As a rule, whenever any *initform* is evaluated for any parameter specifier, that form may refer to any parameter variable to the left of the specifier in which the *initform* appears, including any supplied-p variables, and may rely on no other parameter variable having yet been bound (including its own parameter variable).

*Compatibility note:* At present, one cannot depend on this in Lisp Machine LISP for keyword parameters. It is the "obvious" generalization of the current state of affairs for optional parameters and aux variables. Opinions?

Once the lambda-list has been processed; the forms in the body of the lambda-expression are executed.

These forms may refer to the arguments to the function by using the names of the parameters. On exit from the function, either by a normal return of the function's value(s) or by a non-local exit, the parameter bindings (whether lexical or special) are no longer in effect; any such binding can later be reinstated only if a *closure* over that binding was created and saved before the exit occurred.

Examples:

```
((lambda (a b) (+ a (* b 3))) 4 5) => 19
((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5) => 19
((lambda (a &optional (b 2)) (+ a (* b 3))) 4) => 10
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)))
=> (2 () 3 () ())
((lambda (&optional (a 2) (c 3) &rest x) (list a c x)) 6)
=> (6 t 3 () ())
((lambda (&optional (a 2) (c 3) &rest x) (list a c x)) 6 3)
=> (6 t 3 t ())
((lambda (&optional (a 2) (c 3) &rest x) (list a c x)) 6 3 8)
=> (6 t 3 t (8))
((lambda (&optional (a 2) (c 3) &rest x) (list a c x)) 6 3 8 9 10 11)
=> (6 t 3 t (8 9 10 11))
```

The `&optional`, `&rest`, and `&key` parameter specifiers are not terribly useful in lambda-expressions appearing explicitly as the first element of a function-call form. They are extremely useful, however, in functions given global names by `defun`.

### 3.3. Top-Level Forms

The standard way for the user to interact with a COMMON LISP implementation is via what is called a *read-eval-print loop*: the system repeatedly reads a form from some input source (such as a keyboard or a disk file), evaluates it, and then prints the value(s) to some output sink (such as a display screen or another disk file). As a rule any form (evaluable S-expression) is acceptable. However, certain special forms are specifically designed to be convenient for use as *top-level forms*, ~~as opposed to form embedded within other forms, as (`(+ 2 4)`) is embedded within `(list 2 (+ 4) 5)`.~~ These top-level special forms may be used to define globally named functions, to define macros, to make declarations, and to define global values for special variables.

*Which can be used as other than top level?*

#### 3.3.1. Defining Named Functions

`defun` *name* *lambda-list* `{(declare {declaration})}* [doc-string] {form}* [Special form]`

Evaluating this special form causes the symbol *name* to be a global name for the function specified by the lambda-expression

`(lambda lambda-list {(declare {declaration})}* {form}*)`

defined in the lexical environment in which the `defun` form was executed (because `defun` forms normally appear at top level, this is normally the null lexical environment).

If the optional documentation string *doc-string* is present (it may be present only if at least one *form* is also specified, as it is otherwise taken to be a *form*), then it is put on the property list of the symbol *name* under the indicator *documentation* (see *putpr*). By convention, if the string contains multiple lines then the first line should be a complete summarizing sentence on which the remainder expands.

Other implementation-dependent bookkeeping actions may be taken as well by *defun*. The *name* is returned as the value of the *defun* form.

For example:

```
(defun discriminant (a b c)
  (declare (number a b c))
  "Compute the discriminant for a quadratic equation.
  Given arguments a, b, and c, the value  $b^2-4*a*c$  is calculated.
  A quadratic equation has real, multiple, or complex roots depend
  on whether this calculated value is positive, zero, or negative
  (- (* b b) (* 4 a c)))
  discriminant
  and now (discriminant 1 2/3 -2) => 76/9
```

It is permissible to redefine a function (for example, to install a corrected version of an incorrect definition!). It is not permissible to define as a function any symbol in use as the name of a special form or macro. To redefine a macro name as the name of a function, *fmakunbound* (page 40) must first be applied to the symbol.

??? Query: What do people think of this safety feature? The error handler could offer to do the *fmakunbound* for you and *retry*.

Silly?

### 3.3.2. Defining Macros

Macros are usually defined by using the special form *defmacro* (page *DEFMACRO-FUN*). This facility is fairly complicated, and is described in a separate chapter.

### 3.3.3. Declaring Global Variables and Named Constants

**defvar** *name* [*initial-value* [*documentation*]] [*Special form*]

*defvar* is the recommended way to declare the use of a special variable in a program. It is normally used only as a top-level form.

```
(defvar variable)
```

declares *variable* to be *special* (see *declare* (page 72)), and may perform other system-dependent bookkeeping actions. If a second "argument" is supplied:

```
(defvar variable initial-value)
```

then *variable* is initialized to the result of evaluating the form *initial-value* unless it already has a value. *initial-value* is not evaluated unless it is used; this is useful if it does something expensive like creating a large data structure. The initialization is performed by assignment, and so assigns the variable a global value unless there are currently special bindings of that variable.

~~defvar should be used only at top level, never in function definitions, and only for free variables used by more than one function.~~

defvar also provides a good place to put a comment describing the meaning of the variable (whereas an ordinary special declaration offers the temptation to declare several variables at once and not have room to describe them all). This can be a simple LISP comment:

```
(defvar tv-height 768) ;Height of TV screen in pixels.
```

or, better yet, a third "argument" to defvar, in which case various programs can access the documentation:

```
(defvar tv-height 768 "Height of TV screen in pixels")
```

The documentation should be a string.

*(the value of the initial-value form)*

```
defconst name initial-value [documentation] [Special form]
```

defconst is similar to defvar, but declares a global variable whose value is "constant". An initial value is always given to the variable. It is an error if there are currently any special bindings of the variable (but implementations may or may not check for this).

If the variable is already has a value, an error occurs unless the existing value is equal (page 32) to the specified ~~initial value.~~ *form*

Implementation note: Actually, a specific interaction should occur in which the user is asked whether it is permissible to alter the constant. Perhaps there should be some mechanism to discover who uses the constant.

Rationale: defconst declares a constant, whose value will "never" be changed. Other code may depend on this fact. On the other hand, defvar declares a global variable, whose value is initialized to something but will then be changed by the functions that use it to maintain some state.

Once a symbol has been declared by defconst to be constant, ~~no~~ further assignments or bindings of that variable are ~~permitted~~. This is the case for such system-supplied constants as t and max-fixnum.

*Top-level only?*  
*errors*

*DEFCONSTs can be compiled in.  
(But address must be preserved.)*

*A big example.  
The def convention. defsetf  
defstruct*



## Chapter 4

# Predicates

A *predicate* is a function which tests for some condition involving its arguments and returns `()` if the condition is false, or some non-`()` value if the condition is true. One may think of a predicate as producing a Boolean value, where `()` stands for *false* and anything else stands for *true*. Conditional control structures such as `cond` (page 45), `if` (page 46), `when` (page 46), and `unless` (page 47) test such Boolean values. We say that a predicate *is true* when it returns a non-`()` value, and *is false* when it returns `()`; that is, it is true or false according to whether the condition being tested is true or false.

By convention, the names of predicates usually end in the letter “p” (which stands for “predicate”).

The control structures which test Boolean values only test for whether or not the value is `()`, which is considered to be false. Any other value is considered to be true. A function which returns `()` if it “fails” and some *useful* value when it “succeeds” is called a *pseudo-predicate*, because it can be used not only as a test but also for the useful value provided in case of success. An example of a pseudo-predicate is `member` (page 97).

### 4.1. Data Type Predicates

Perhaps the most important predicates in LISP are those which test for data types; that is, given a data object one can determine whether or not it belongs to a given type.

In COMMON LISP, types are named by LISP objects, specifically symbols and lists. The type symbols defined by the system include:

<code>null</code>	<code>cons</code>	<code>list</code>	<code>symbol</code>
<code>vector</code>	<code>string</code>	<code>bit-string</code>	<code>array</code>
<code>function</code>	<code>structure</code>	<code>random</code>	<code>character</code>
<code>number</code>	<code>stream</code>	<code>float</code>	<code>string-char</code>
<code>integer</code>	<code>fixnum</code>	<code>bignum</code>	<code>bit</code>
<code>short-float</code>	<code>single-float</code>	<code>double-float</code>	<code>long-float</code>
<code>complex</code>	<code>ratio</code>	<code>readtable</code>	<code>package</code>
<code>sequence</code>			

In addition, when a structure type is defined using `defstruct` (page 167), the name of the structure type becomes a valid type symbol.

If the name of a type is a list, the *car* of the list is a symbol, and the rest of the list is subsidiary type information. As a general rule, any subsidiary item may be replaced by `?`, or simply omitted if it is the last item of the list; in any of these cases the item is said to be unspecified.

List names of type `s` generally refer to *specializations* of data types named by symbols. These specializations may be reflected by more efficient representations in the underlying implementation. As an example, consider the type `(vector short-float)`. Implementation A may choose to provide a specialized representation for vectors of short floating-point numbers, and implementation B may choose not to. If you should want to create a vector for the express purpose of holding only short-float objects, you may optionally specify to `make-vector` (page 155) the element type `short-float`, meaning, "Produce the most specialized vector representation capable of holding short-floats which the implementation can provide." Implementation A will then produce a specialized short-float vector, and implementation B will produce an ordinary vector (one of type `(vector t)`).

If one were then to ask whether the vector were actually of type `(vector short-float)`, both implementations could properly say "yes"; implementation B might or might not verify that the vector actually contained short-floats. On the other hand, implementation A, if asked whether a vector of type `(vector t)` were of type `(vector short-float)`, it could properly say "no" without even checking the contents of the vector. All this is a bit tricky, but is designed to allow some implementations to provide efficient specialized representations without having to burden all implementations with irrelevant specialized data types.

No, this is bogus. Feb!

The valid list-format names for data types are:

- `(array type dimensions)`: a specialized array whose elements are all members of the type *type* and whose dimensions match *dimensions*. To be more precise, this type encompasses those arrays which can result by specifying *type* to the function `make-array` (page 159). *type* must be a valid type label. *dimensions* may be a non-negative integer, which is the number of dimensions, or it may be a list of non-negative integers representing the length of each dimension (any given dimensions may be unspecified).

-if unspecified then t is assumed

For example:

*(array integer)*

```
(array integer 3) ; Three-dimensional arrays of integers.
(array integer (? ? ?)) ; Three-dimensional arrays of integers.
(array ? (4 5 6)) ; 4-by-5-by-6 arrays.
(array character (3 ?)) ; Two-dimensional arrays of characters
; which have exactly three rows.
(array short-float ()) ; Zero-rank arrays of short floating-point numbers.
```

- `(vector type size)`: a specialized vector (a one-dimensional array with no fill pointer and no displacement) whose elements are all members of the type *type* and which is of length *size*. To be more precise, this type encompasses those vectors which can result by specifying *type* to the function `make-array` (page 159). *size* must be a non-negative integer, and *type* a valid type label. If *type* is unspecified then `t` is assumed (components may be of any type); if *size* is unspecified, then vectors of all sizes are included.

For example:

(vector double-float) ; Vectors of double-format floating-point numbers.  
 (vector ? 5) ; Vectors of length 5.  
 (vector (mod 32) ?) ; Vectors of integers between 0 and 31.

The types (vector string-char) and (vector bit) are so useful that they have the special names `string` and `bit-string`; every COMMON LISP implementation must provide these as distinct data types.

Rationale: NIL had been using the name `bits` for a bit vector. This tended to lead to awkward prose: one had to speak of "a bits". The singular noun `bit-vector` is easier to discuss.

- (integer *low high*): any integer between *low* and *high*. The limits *low* and *high* must each be an integer, a list of an integer, or (); an integer is an inclusive limit, a list of an integer is an exclusive limit, and () means that a limit does not exist and so effectively denotes minus or plus infinity, respectively. The type `fixnum` is simply a name for (integer *smallest largest*) for implementation-dependent values of *smallest* and *largest*. The type (integer 0 1) is so useful that it has the special name `bit`.
- (mod *n*): a non-negative integer less than *n*. This is equivalent to (integer 0 *n*) or (what is the same thing) (integer 0 (*n*)). } minus
- (signed-byte *s*): equivalent to (integer  $-2^{s-1} 2^{s-1}-1$ ).
- (unsigned-byte *s*): equivalent to (mod  $2^s$ ), that is, (integer 0  $2^s-1$ ).
- (float *low high*): any floating-point number between *low* and *high*. The limits *low* and *high* must each be a floating-point number, a list of a floating-point number, or (); a floating-point number is an inclusive limit, a list of a floating-point number is an exclusive limit, and () means that a limit does not exist and so effectively denotes minus or plus infinity, respectively. As examples, the result of the cosine function may be described as being of type (float -1.0 1.0), and the argument to the logarithm function must be of type (float (0.0) ()).

In a similar manner one may use (short-float *low high*), (single-float *low high*), (double-float *low high*), or (long-float *low high*); the limits must be floating-point numbers of the appropriate type.

- (complex *rtype itype*): a complex number whose real part is of type *rtype* and whose imaginary part is of type *itype*. To be more precise, this type encompasses all those complex numbers which can result by giving arguments of the specified types to the function `complex` (page 130). In a break with the usual convention on omitted items, if *itype* is omitted then it is taken to be the same as *rtype*. As examples, gaussian integers might be described as (complex integer), and the result of the complex logarithm function might be described as being of type (complex float (float #.(- pi) #.pi)). If unspecified, default is to number.
- (function (*arg1-type arg2-type ...*) *value1-type value2-type ...*): this specifies a function which accepts arguments at *least* of the types specified by the *argj-type* forms, and returns values which are members of the types specified by the *valuej-type* forms. The `&optional` and `&rest` keywords may appear in either list of types. As an example, the function `cons` is of type (function (t t) cons), because it can accept any two arguments and always returns a cons. It is also of type (function (float string) list), because it can certainly accept a floating-point number and a string (among other things), and its result is always

of type `list` (in fact `cons` and never `null`, but that does not matter for this type determination).

- `(oneof object1 object2 ...)`: a name for a type containing precisely those objects named. An object is of this type if and only if it is `eq1` (page 31) to one of the specified objects.
- `(not type)`: all those objects which are *not* of the specified type.
- `(or type1 type2 ...)`: the union of the specified types. For example, the type `list` by definition is the same as `(or null cons)`. Also, the value returned by the function `position` (page 80) is always of type `(or null (integer 0 ()))` (either `()` or a non-negative integer).
- `(and type1 type2 ...)`: the intersection of the specified types.

Subtype fn.

`typep object &optional type`

[Function]

`(typep object type)` is a predicate which is true if `object` is of type `type`, and is false otherwise. Note that an object can be "of" more than one type, since one type can include another. The `type` may be any of the type names mentioned above.

Error signalled if cannot determine?

`(typep object)` returns an implementation-dependent result: some `type` of which the `object` is a member. Implementations are encouraged to return the most specific type which can be conveniently computed and is likely to be useful to the user. It is required that if the argument is a named structure created by `defstruct` then `typep` will return the name of that structure and not the symbol `structure`. Because the result is implementation-dependent, it is usually better to use `typep` of one argument primarily for debugging purposes, and to use `typep` of two arguments or the `typecase` (page 48) special form in programs.

#### 4.1.1. Specific Data Type Predicates

The following predicates are for testing for individual data types.

Should some of these return arg?

`null object`

[Function]

`null` is true if its argument is `()`, and otherwise is false. This is the same operation performed by the function `not` (page 34); however, `not` is normally used to invert a Boolean value, while `null` is normally used to test for an empty list. The programmer can therefore express *intent* by the choice of function name.

```
(null x) <=> (typep x 'null) <=> (eq x '())
```

`symbolp object`

[Function]

`symbolp` is true if its argument is a symbol, and otherwise is false.

```
(symbolp x) <=> (typep x 'symbol)
```

**Compatibility note:** In most LISP dialects, including MACLISP, INTERLISP, and even LISP 1.5, `()` is in fact represented by the symbol `nil`, and therefore `(symbolp '())` is true. This association of a symbol with the

empty list has caused problems. Programmers are advised to write code in such a way as not to depend on `()` and `nil` being either the same or not the same, if possible.

*atom object**[Function]*

The predicate `atom` is true if its argument is not a cons, and otherwise is false. It is the inverse of `consp`. Note that `(atom '())` is true.

```
(atom x) <=> (typep x 'atom) <=> (not (typep x 'cons))
```

*consp object**[Function]*

The predicate `consp` is true if its argument is a cons, and otherwise is false. It is the inverse of `atom`. Note that `(consp '()) => ()`.

```
(consp x) <=> (typep x 'cons) <=> (not (typep x 'atom))
```

Compatibility note: Some LISP implementations call this function `pairp` or `listp`. The name `pairp` was rejected for COMMON LISP because it emphasizes too strongly the dotted-pair notion rather than the usual usage of conses in lists. On the other hand, `listp` too strongly implies that the cons is in fact part of a list, which after all it might not be; moreover, `()` is a list, though not a cons. The name `consp` seems to be the appropriate compromise.

*listp object**[Function]*

`listp` is true if its argument is a cons or the empty list `()`, and otherwise is false. It does not check for whether the list is a "true list" (one terminated by `()`) or a "dotted list" (one terminated by a non-null atom).

```
(listp x) <=> (typep x 'list) <=> (typep x '(cons null))
```

Compatibility note: Lisp Machine LISP defines `listp` to mean the same as `pairp`, but this is under review. The definition given here is that adopted by NIL.

*numberp object**[Function]*

`numberp` is true if its argument is any kind of number, and otherwise is false.

```
(numberp x) <=> (typep x 'number)
```

*integerp object**[Function]*

`integerp` is true if its argument is an integer, and otherwise is false.

```
(integerp x) <=> (typep x 'integer)
```

Compatibility note: In MACLISP this is called `fixp`. Users have been confused as to whether this meant "integer" or "fixnum", and so these names have been adopted here.

*rationalp object**[Function]*

`rationalp` is true if its argument is a rational number (a ratio or an integer), and otherwise is false.

```
(rationalp x) <=> (typep x 'rational)
```

`floatp` *object* [Function]  
`floatp` is true if its argument is a floating-point number, and otherwise is false.  
(`floatp` `x`) <=> (`typep` `x` 'float)

`characterp` *object* [Function]  
`characterp` is true if its argument is a character, and otherwise is false.  
(`characterp` `x`) <=> (`typep` `x` 'character)

`stringp` *object* [Function]  
`stringp` is true if its argument is a string, and otherwise is false.  
(`stringp` `x`) <=> (`typep` `x` 'string)

`vectorp` *object* [Function]  
`vectorp` is true if its argument is a vector, and otherwise is false.  
(`vectorp` `x`) <=> (`typep` `x` 'vector)

`arrayp` *object* [Function]  
`arrayp` is true if its argument is an array, and otherwise is false.  
(`arrayp` `x`) <=> (`typep` `x` 'array)

`structurep` *object* [Function]  
`structurep` is true if its argument is a structure, and otherwise is false.  
(`structurep` `x`) <=> (`typep` `x` 'structure)

`functionp` *object* [Function]  
`functionp` is true if its argument is suitable for applying to arguments, using for example the `funcall` or `apply` function. Otherwise `functionp` is false.

`subrp` *object* [Function]  
`subrp` is true if its argument is any compiled code object, and otherwise is false.  
(`subrp` `x`) <=> (`typep` `x` 'subr)

`closurep` *object* [Function]  
`closurep` is true if its argument is a closure, and otherwise is false.

refs —

*even more general than equal*

## 4.2. Equality Predicates.

COMMON LISP provides a spectrum of predicates for testing for equality of two objects: `eq` (the most specific), `eq1`, `equal`, and `equalp` (the most general). `eq` and `equal` have the meanings traditional in LISP. `eq1` was added because it is frequently needed, and `equalp` was added primarily to complement the arithmetic comparison predicates `lessp` (page LESSP-FUN) and `greaterp` (page GREATERP-FUN). If two objects satisfy any one of these equality predicates, then they also satisfy all those which are more general.

`eq x y`*[Function]*

`(eq x y)` is true if and only if `x` and `y` are the same identical object. (Implementationally, `x` and `y` are usually `eq` if and only if they address the same identical memory location.)

It should be noted that things that print the same are not necessarily `eq` to each other. Symbols with the same print name usually are `eq` to each other, because of the use of the `intern` (page INTERN-FUN) function. However, numbers with the same value need not be `eq`, and two similar lists are usually not `eq`.

For example:

```
(eq 'a 'b) is false
(eq 'a 'a) is true
(eq 3 3) might be true or false, depending on the implementation
(eq 3 3.0) is false
(eq (cons 'a 'b) (cons 'a 'c)) is false
(eq (cons 'a 'b) (cons 'a 'b)) is false
(setq x '(a . b)) (eq x x) is true
(eq #\A #\A) might be true or false, depending on the implementation
(eq "Foo" "Foo") is false
(eq "FOO" "foo") is false
```

*probably*

**Implementation note:** `eq` simply compares the two pointers given it, so any kind of object which is represented in an "immediate" fashion will indeed have like-valued instances satisfy `eq`. On the PERQ, for example, fixnums and characters happen to "work". However, no program should depend on this, as other implementations of COMMON LISP might not use an immediate representation for these data types.

`eq1 x y`*[Function]*

The `eq1` predicate is true if its arguments are `eq`, or if they are numbers of the same type with the same value (that is, they are `=` (page 118)), or if they are character objects which represent the same character (that is, they are `char=` (page 142)).

For example:

```
(eql 'a 'b) is false
(eql 'a 'a) is true
(eql 3 3) is true
(eql 3 3.0) is false
(eql (cons 'a 'b) (cons 'a 'c)) is false
(eql (cons 'a 'b) (cons 'a 'b)) is false
(setq x '(a . b)) (eql x x) is true
(eql #\A #\A) is true
(eql "Foo" "Foo") is false
(eql "FOO" "foo") is false
```

`equal x y``[Function]`

The `equal` predicate is true if its arguments are similar (isomorphic) objects. A rough rule of thumb is that two objects are `equal` if and only if their printed representations are the same.

Numbers and characters are compared as for `eql`. Symbols are compared as for `eq`. This can violate the rule of thumb about printed representations, but only in the case of two distinct symbols with the same print name, and this does not ordinarily occur.

Objects which have components are `equal` if they are of the same type and corresponding components are `equal`. This test is implemented in a recursive manner, and will fail to terminate for circular structures. For conses, `equal` is defined recursively as the two *car*'s being `equal` and the two *cdr*'s being `equal`.

Two arrays are `equal` if and only if they have the same number of dimensions, the dimensions match, the element types match, and the corresponding components are `equal`.

*Compatibility note:* In Lisp Machine Lisp, `equal` ignores the difference between upper and lower case in strings. This violates the rule of thumb about printed representations, however, which is very useful, especially to novices. It is also inconsistent with the treatment of single characters, which are represented as fixnums.

For example:

```
(equal 'a 'b) is false
(equal 'a 'a) is true
(equal 3 3) is true
(equal 3 3.0) is false
(equal (cons 'a 'b) (cons 'a 'c)) is false
(equal (cons 'a 'b) (cons 'a 'b)) is true
(setq x '(a . b)) (equal x x) is true
(equal #\A #\A) is true
(equal "Foo" "Foo") is true
(equal "FOO" "foo") is false
```

To recursively compare only conses, and compare all atoms using `eq`, use `tree-equal` (page 88).

`equalp x y &optional fuzz``[Function]`

Two objects are `equalp` if they are `eql`, if they are characters and differ only in alphabetic case (that is, they are `char-equal` (page 142)), if they are numbers and have the same numerical value, even if they are of different types, or if they have components which are all `equalp`. ~~By this latter characteristic `equalp` complements `lessp` (page LISP-FUN) and `greaterp` (page~~

~~GREATERP-FUN), which perform inequality comparisons among numbers of possibly differing types. When comparing floating-point numbers, or comparing a floating-point number to any other kind of number, the optional argument *fuzz* is used. Two numbers are considered to be equal if the absolute value of their difference is no greater than *fuzz* times the absolute value of the one with the larger absolute value; that is,  $x$  and  $y$  are considered equal if  $abs(x-y) \leq fuzz * max(abs(x), abs(y))$ . If no third argument is supplied, then *fuzz* defaults to 0.0, and in this case  $x$  and  $y$  must be exactly equal for `equalp` to be true. (See the function `=` (page 118).)~~

Objects which have components are `equalp` if they are of the same type and corresponding components are `equalp`. This test is implemented in a recursive manner, and will fail to terminate for circular structures. ~~For conses, `equalp` is defined recursively as the two cdr's being `equalp` and the two car's being `equalp`.~~

Two arrays are `equalp` if and only if they have the same number of dimensions, the dimensions match, the element types match, and the corresponding components are `equalp`.

??? Query: How about eliminating the clause "the element types match" from the above specification? This would allow a string and a general array that happens to contain character to be `equalp`, for example.

For example:

```
(equalp 'a 'b) is false
(equalp 'a 'a) is true
(equalp 3 3) is true
(equalp 3 3.0) is true
(equalp (cons 'a 'b) (cons 'a 'c)) is false
(equalp (cons 'a 'b) (cons 'a 'b)) is true
(setq x '(a . b)) (equalp x x) is true
(equalp #\A #\A) is true
(equalp "Foo" "Foo") is true
(equalp "FOO" "foo") is true
```

### 4.3. Logical Operators

COMMON LISP provides three operators on Boolean values: `and`, `or`, and `not`. Of these, `and` and `or` are also control structures, because their arguments are evaluated conditionally. `not` necessarily examines its single argument, and so is a simple function.

`not x`

[Function]

`not` is true if  $x$  is `()`, and otherwise is false. It therefore inverts its argument, interpreted as a Boolean value.

`null` (page 36) is the same as `not`; both functions are included for the sake of clarity. As a matter of style, it is customary to use `null` to check whether something is the empty list, and to use `not` to invert the sense of a logical value.

and {form}\*

[Special form]

(and form1 form2 ... ) evaluates each form, one at a time, from left to right. If any form evaluates to ( ), and immediately is false without evaluating the remaining forms. If every form but the last evaluates to a non-( ) value, and returns whatever the last form returns. Therefore in general and can be used both for logical operations, where ( ) stands for false and non-( ) values stand for true, and as a conditional expression.

For example:

```
(if (and (>= n 0)
        (lessp n (length a-vector))
        (eq (vref a-vector n) 'foo))
    (princ "Foo!"))
```

The above expression prints "Foo!" if element n of a-vector is the symbol foo, provided also that n is indeed a valid index for a-vector. Because and guarantees left-to-right testing of its parts, vref is not performed if n is out of range. (In this example writing

```
(and (>= n 0)
      (lessp n (length a-vector))
      (eq (vref a-vector n) 'foo))
      (princ "Foo!"))
```

would accomplish the same thing; the difference is purely stylistic.) Because of the guaranteed left-to-right ordering, and is like the and then operator in ADA, rather than the and operator. ref

See also if (page 46) and when (page 46), which are sometimes stylistically more appropriate than and for conditional purposes.

From the general definition, one can deduce that (and x) <=> x. Also, (and) is true, which is an identity for this operation.

and can be defined in terms of cond (page 45) as follows:

```
(and x y z ... w) <=>
  (cond ((not x) ())
        ((not y) ())
        ((not z) ())
        ...
        (t w))
```

or {form}\*

[Special form]

(or form1 form2 ... ) evaluates each form, one at a time, from left to right. If any form evaluates to something other than ( ), or immediately returns it without evaluating the remaining forms. If every form but the last evaluates to ( ), or returns whatever evaluation of the last of the forms returns. Therefore in general or can be used both for logical operations, where ( ) stands for false and non-( ) values stand for true, and as a conditional expression. Because of the guaranteed left-to-right ordering, or is like the or else operator in ADA, rather than the or operator. ref

See also if (page 46) and unless (page 47), which are sometimes stylistically more appropriate than or for conditional purposes.

From the general definition, one can deduce that  $(\text{or } x) \Leftrightarrow x$ . Also,  $(\text{or})$  is false, which is the identity for this operation.

`or` can be defined in terms of `cond` (page 45) as follows:

```
(or x y z ... w)  $\Leftrightarrow$   
  (cond (x) (y) (z) ... (t w))
```



## Chapter 5

# Control Structure

LISP provides a variety of special structures for organizing programs. Some have to do with flow of control (control structures), while others control access to variables (environment structures). Most of these features are implemented either as special forms or as macros (which typically expand into complex program fragments involving special forms).

Function application is the primary method for construction of LISP programs. Operations are written as the application of a function to its arguments. Usually, LISP programs are written as a large collection of small functions, each of which implements a simple operation. These functions operate by calling one another, and so larger operations are defined in terms of smaller ones. LISP functions may call upon themselves recursively, either directly or indirectly.

LISP, while more applicative in style than statement-oriented, nevertheless provides many operations which produce side-effects, and consequently requires constructs for controlling the sequencing of side-effects. The construct `progn` (page 42), which is roughly equivalent to an ALGOL `begin-end` block with all its semicolons, executes a number of forms sequentially, discarding the values of all but the last. Many LISP control constructs include sequencing implicitly, in which case they are said to provide an "implicit `progn`". Other sequencing constructs include `prog1` (page 42) and `prog2` (page 43).

For looping, COMMON LISP provides the general iteration facility `do` (page 49), as well as a variety of special-purpose iteration facilities for iterating or mapping over various data structures.

COMMON LISP provides the simple one-way conditionals `when` and `unless`, the simple two-way conditional `if`, and the more general multi-way conditionals such as `cond` and `selectq`. The choice of which form to use in any particular situation is a matter of taste and style.

### Non-local exits, binding of temps, multiple values. Eventually make all this in order corresponding to main text.

## 5.1. Constants and Variables

### 5.1.1. Reference

`quote` *object*

[Special form]

`(quote x)` simply returns `x`. The argument is not evaluated, and may be any LISP object. This construct allows any LISP object to be written as a constant value in a program.

For example:

```
(setq a 43)
(list a (cons a 3)) => (43 (43 . 3))
(list (quote a) (quote (cons a 3))) => (a (cons a 3))
```

Since `quote` forms are so frequently useful but somewhat cumbersome to type, a standard abbreviation is defined for them: any form preceded by a single quote ( `'` ) character is assumed to have `"(quote )"` wrapped around it.

For example:

```
(setq x '(the magic quote hack))
      is normally interpreted when read to mean
(setq x (quote (the magic quote hack)))
```

by the Lisp reader

ref?

`function` *fn*

[Special form]

The value of `function` is always the functional interpretation of the form `fn`; `fn` is interpreted as if it had appeared in the functional position of a function invocation. In particular, if `fn` is a symbol, the functional value of the variable whose name is that symbol is returned.

Compatibility note: ???

If `fn` is a lambda-expression, then a functional object (a lexical closure) is returned.

Since `function` forms are so frequently useful (for passing functions as arguments to other function) but somewhat cumbersome to type, a standard abbreviation is defined for them: any form preceded by a sharp sign and then a single quote ( `#'` ) is assumed to have `"(function )"` wrapped around it.

For example:

```
(rem-if #'numberp '(1 a b 3))
      is normally interpreted when read to mean
(rem-if (function numberp) '(1 a b 3))
```

mapcar

by the Lisp reader

mapcar

`symbol` *symbol*

[Function]

`symbol` returns the current value of the dynamic (special) variable named by `symbol`. An error occurs if the symbol has no value; see `boundp` (page 39) and `makunbound` (page 40).

`symbol` cannot access the value of a local (lexically bound) variable.

lexical

This function is particularly useful for implementing interpreters for languages embedded in LISP. The corresponding assignment primitive is `set` (page 40).

**fsymeval** *symbol*

[Function]

**fsymeval** returns the current global function definition named by *symbol*. An error occurs if the symbol has no function definition; see **boundp** (page 39) and **makunbound** (page 40).

**fsymeval** cannot access the value of a local function name (lexically bound as by **flet** (page FLET-FUN) or **labels** (page LABELS-FUN)).

This function is particularly useful for implementing interpreters for languages embedded in LISP. The corresponding assignment primitive is **fset** (page 40).

**boundp** *symbol*

[Function]

**fboundp** *symbol*

[Function]

**boundp** is true if the dynamic (special) variable named by *symbol* has a value; otherwise, it returns (). **fboundp** is the analogous predicate for the global function definition named by *symbol*.

See also **set** (page 40), **fset** (page 40), **makunbound** (page 40), and **fmakunbound** (page 40).

*Compatibility note:* I believe that in Lisp Machine LISP **boundp** can manage to refer to a local variable if its argument appears as a quoted constant. If so, it is an incredible hack, and violates the rule that a function cannot tell how its arguments were computed. In COMMON LISP, **boundp** can never refer to a local variable, and **fboundp** can never refer to a local function definition.

split —

### 5.1.2. Assignment

**setq** {*var form*}\*

[Special form]

The special form (**setq** *var1 form1 var2 form2* ...) is the "simple variable assignment statement" of LISP. First *form1* is evaluated and the result is assigned to *var1*, then *form2* is evaluated and the result is assigned to *var2*, and so forth. The variables are represented as symbols, of course, and are interpreted as referring to static or dynamic instances according to the usual rules. **setq** returns the last value assigned, that is, the result of the evaluation of its last argument. As a boundary case, the form (**setq**) is legal and returns (). As a rule there must be an even number of argument forms.

For example:

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

*x* is set to 6, *y* is set to (6), and the **setq** returns (6). Note that the first assignment was performed before the second form was evaluated, allowing that form to use the new value of *x*.

See also the description of **setf** (page SETF-FUN), which is the "general assignment statement", capable of assigning to variables, array elements, and other locations.

**psetq** {*var form*}\*[*Special form*]

A **psetq** form is just like a **setq** form, except that the assignments happen in parallel; first all of the forms are evaluated, and then the variables are set to the resulting values. The value of the **psetq** form is ().

For example:

```
(setq a 1)
(setq b 2)
(psetq a b b a)
a => 2
b => 1
```

In this example, the values of *a* and *b* are exchanged by using parallel assignment! (Note that the **do** (page 49) iteration construct performs a very similar thing when stepping iteration variables.)

see exchf

**set** *symbol value*[*Function*]

**set** allows alteration of the value of a dynamic (special) variable. **set** causes the dynamic variable named by *symbol* to take on *value* as its value. Only the value of the current dynamic binding is altered; if there are no bindings in effect, the most global value is altered.

For example:

```
(set (if (eq a b) 'c 'd) 'foo)
```

will either set *c* to *foo* or set *d* to *foo*, depending on the outcome of the test (*eq a b*).

Both functions return *value* as the result value.

**set** cannot alter the value of a local (lexically bound) variable. The special form **setq** (page 47) is usually used for altering the values of variables (lexical or dynamic) in programs. **set** is particularly useful for implementing interpreters for languages embedded in LISP. See also **prog** (page 45), a construct which performs binding rather than assignment of dynamic variables.

**fset** *symbol value*[*Function*]

**fset** allows alteration of the global function definition named by *symbol* to be *value*. **fset** returns *value*.

**fset** cannot alter the value of a local (lexically bound) function definition, as made by **flet** (page FLET-FUN) or **labels** (page LABELS-FUN). ~~The special form **setq** (page 47) is usually used for altering the values of variables (lexical or dynamic) in programs.~~ **fset** is particularly useful for implementing interpreters for languages embedded in LISP.

**makunbound** *symbol*[*Function*]**fmakunbound** *symbol*[*Function*]

**makunbound** causes the dynamic (special) variable named by *symbol* to become unbound (have no value). **fmakunbound** does the analogous thing for the global function definition named by *symbol*.

For example:

split-

```
(setq a 1)
a => 1
(makunbound 'a)
a => causes an error
(defun foo (x) (+ x 1))
(foo 4) => 5
(fmakunbound 'foo)
(foo 4) => causes an error
```

Both functions return *symbol* as the result value.

**Compatibility note:** I believe that in Lisp Machine LISP `makunbound` can manage to refer to a local variable if its argument appears as a quoted constant. If so, it is an incredible hack, and violates the rule that a function cannot tell how its arguments were computed. In COMMON LISP, `makunbound` can never refer to a local variable, and `fmakunbound` can never refer to a local function definition.

## 5.2. Generalized Variables

In LISP, a variable can remember one piece of data, a LISP object. The main operations on a variable are to recover that piece of data, and to alter the variable to remember a new object; these operations are often called *access* and *update* operations. The concept of variables named by symbols can be generalized to any storage location that can remember one piece of data, no matter how that location is named. Examples of such storage locations are the *car* and *cdr* of a cons, elements of an array, and components of a structure.

For each kind of generalized variable, there are typically two functions which implement the conceptual *access* and *update* operations. For a variable, merely mentioning the name of the variable accesses it, while the `setq` (page 47) special form can be used to update it. The function `car` (page 87) accesses the *car* of a cons, and the function `rplaca` (page 94) updates it. The function `aref` (page 160) accesses an array element, and the function `aset` (page 160) updates it.

Rather than thinking about two distinct functions that respectively access and update a storage location somehow deduced from their arguments, we can instead simply think of a call to the access function with given arguments as a *name* for the storage location. Thus, just as `x` may be considered a name for a storage location (a variable), so `(car x)` is a name for the *car* of some cons (which is in turn named by `x`), and `(aref a 105)` is a name for element 105 of the array named `a`. Now, rather than having to remember two functions for each kind of generalized variable (having to remember, for example, that `aset` corresponds to `aref`), we adopt a uniform syntax for updating storage locations named in this way, using the `setf` special form. This is analogous to the way we use the `setq` special form to convert the name of a variable (which is also a form which accesses it) into a form which updates it. The uniformity of this approach may be seen from the following table:

Access function	Update function	Update using <code>setf</code>
<code>x</code>	<code>(setq x newvalue)</code>	<code>(setf x newvalue)</code>
<code>(car x)</code>	<code>(rplaca x newvalue)</code>	<code>(setf (car x) newvalue)</code>
<code>(aref a 105)</code>	<code>(aset newvalue a 105)</code>	<code>(setf (aref a 105) newval</code>
<code>(nth n x)</code>	<code>(setnth n x newvalue)</code>	<code>(setf (nth n x) newvalue)</code>

`setf` is actually a macro that examines an access form and expands into the appropriate update function.

`setf` *place newvalue*

[Macro]

`setf` takes a form *place* that when evaluated *accesses* a data object in some location, and “inverts” it to produce a corresponding form to *update* the location. A call to the `setf` macro therefore expands into an update form that stores the result of evaluating the form *newvalue* into the place referred to by the *access-form*.

For example:

```
(setf a 3) ==> (setq a 3)
(setf (plist 'a) '(foo bar)) ==> (setplist 'a '(foo bar))
(setf (aref q 2) 56) ==> (aset 56 q 2)
(setf (cadr w) x) ==> (rplaca (cdr w) x)
```

The form *place* may be any one of the following:

- The name of a variable (either lexical or dynamic).
- A function call form whose first element is the name of any one of the following functions:

car	(page 87) caaar	
	(page CAAAR-FUN)	caddr
	(page CADDR-FUN)	
cdr	(page 87) cdaaar	
	(page CDAAAR-FUN)	cddddr
	(page CDDDDR-FUN)	
caar	(page CAAR-FUN)	cadaar
	(page CADAAR-FUN)	elt
	(page 76)	
cdar	(page CDAR-FUN)	cddaar
	(page CDDAAR-FUN)	nth
	(page 89)	
cadr	(page CADR-FUN)	caadar
	(page CAADAR-FUN)	vref
	(page 156)	
caddr	(page CDDR-FUN)	cdadar
	(page CDADAR-FUN)	aref
	(page 160)	
caaar	(page CAAAR-FUN)	caddar
	(page CADDAR-FUN)	symeval
	(page 46)	
cdaar	(page CDAAR-FUN)	cdddar
	(page CDDDAR-FUN)	fsymeval
	(page 47)	
cadar	(page CADAR-FUN)	caadr
	(page CAAADR-FUN)	getpr
	(page GETPR-FUN)	
cddar	(page CDDAR-FUN)	cdaadr
	(page CDAADR-FUN)	gethash
	(page 109)	
caadr	(page CAADR-FUN)	cadadr
	(page CADADR-FUN)	plist
	(page 114)	
cdadr	(page CDADR-FUN)	cddadr
	(page CDDADR-FUN)	
caddr	(page CADDR-FUN)	caaddr
	(page CAADDR-FUN)	
cddddr	(page CDDDR-FUN)	cdaddr
	(page CDADDR-FUN)	

- A function call form whose first element is the name of a selector function constructed

How does the fit into this?

by `defstruct` (page 167).

- A function call form whose first element is the name of any one of the following functions, provided that the new value is of the specified type so that it can be used to replace the specified "location" (which is in each of these cases not really a truly generalized variable):

Function name	Required type	Update function <i>u</i>
<code>char</code> (page 149) (page 150)	<code>string-char</code>	<code>rplachar</code>
<code>bit</code> (page 156) (page 156)	<code>(mod 2)rplacbit</code>	
<code>subseq</code> (page 76) (page 77)	<code>sequence</code>	<code>replace</code>

- A function call form whose first element is the name of any one of the following functions, provided that the specified argument to that function is in turn a *place* form; in this case the new *place* has stored back into it the result of applying the specified "update" function (which is in each of these cases not a true update function):

Function name	Argument that is a <i>place</i>	Update function <i>us</i>
<code>char-bit</code> (page CHAR-BIT-FUN) (page SET-CHAR-BIT-FUN)	First	<code>set-char-bit</code>
<code>ldb</code> (page 134) (page 135)	Second	<code>dpb</code>
<code>mask-field</code> (page 135) (page 135)	Second	<code>deposit-field</code>

- A macro call, in which case the macro call is expanded and `setf` then analyzes the resulting form.

`setf` carefully arranges to preserve the usual left-to-right order in which the various subforms are evaluated. For example,

```
(setf (aref (compute-an-array) 105) (compute-newvalue))
```

does not expand precisely into

```
(aset (compute-newvalue) (compute-an-array) 105)
```

lest side effects in the computations `(compute-an-array)` and `(compute-newvalue)` occur in the wrong order. Instead this example will expand into something more like

```
(let ((G1 (compute-an-array))
      (G2 105)
      (G3 (compute-newvalue)))
  (aset G3 G1 G2))
```

The exact expansion for any particular form is not guaranteed and may even be implementation-

dependent; all that is guaranteed is that the expansion of a `setf`-form will be an update form that works for that particular implementation, and that the left-to-right evaluation of subforms is preserved.

Compatibility note: Lisp Machine LISP, at least, officially does not preserve the order of evaluation, but also seems to regard this as a bug to be fixed. ~~What shall Common Lisp do?~~

The ultimate result of evaluating a `setf` form is the value of `newvalue`. (Therefore `(setf (car x) y)` does not expand into precisely `(rplaca x y)`, but into something more like

`(let ((G1 x) (G2 y)) (rplaca x y) y)`

the precise expansion being implementation-dependent.)

Compatibility note: Presently the value of a `setf` form is generally considered to be undefined. Are implementors willing always to return `newvalue`?

Yes!

The user can define new `setf` expansions by using `defsetf` (page DEFSETF-FUN).

`swapf place newvalue`

[Macro]

The datum in `place` is replaced by `newvalue`, and then the old value of `place` is returned. The form `place` may be any form acceptable as a generalized variable to `setf` (page 50).

For example:

```
(setq x '(a b c))
(swapf (cadr x) 'z) => b
and now x => (a z c)
```

The effect of `(swapf place newvalue)` is roughly equivalent to

```
(prog1 place (setf place newvalue))
```

except that the latter would evaluate any subforms of `place` twice, while `swapf` takes care to evaluate them only once.

For example:

```
(setq n 0)
(setq x '(a b c d))
(swapf (nth (setq n (+ n 1)) x) 'z) => b
and now x => (a z c d)
```

but

```
(setq n 0)
(setq x '(a b c d))
(prog1 (nth (setq n (+ n 1)) x)
      (setf (nth (setq n (+ n 1)) x) 'z)) => b
and now x => (a b z d)
```

Moreover, for certain `place` forms `swapf` may be significantly more efficient than the `prog1` version.

`exchf place1 place2`

[Macro]

The data in `place1` and `place2` is exchanged, and then the old value of `place2` (which has become the new value of `place1`) is returned. The forms `place1` and `place2` may be any forms acceptable as generalized variables to `setf` (page 50). If `place1` and `place2` refer to the same generalized

variable, then the effect is to leave it unchanged and return its value.

For example:

```
(setq x '(a b c))
(exchf (car x) (cadr x)) => b
and now x => (b a c)
```

The effect of `(exchf place1 place2)` is roughly equivalent to

```
(setf place1 (prog1 place2 (setf place2 place1)))
```

except that the latter would evaluate any subforms of *place1* and *place2* twice, while `exchf` takes care to evaluate them only once. Moreover, for certain *place* forms `exchf` may be significantly more efficient than the `prog1` version.

Other macros that manipulate generalized variables include `incf` (page INCF-FUN), `decf` (page DECF-FUN), `push` (page 92), and `pop` (page 92).

*guess, push, remf*

### 5.3. Function Invocation

The most primitive form for function invocation in LISP of course has no name; any list which which has no other interpretation as a macro call or special form is taken to be a function call. Other constructs are provided for less common but nevertheless frequently useful situations.

`apply function arglist`

[Function]

This applies *function* to the list of arguments *arglist*. *arglist* should be a list; *function* can be a compiled-code object, or it may be a "lambda expression", that is, a list whose *car* is the symbol `lambda`, or it may a symbol, in which case the ~~dynamic functional value~~ of that symbol is used (but it is illegal in this case for that symbol to be the name of a macro or special form).

For example:

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

*that*  
*function definition for lambda-exps?*

Of course, *arglist* may be `()` (in which case the function is given no arguments.)

Compatibility note: ???

`funcall fn &rest arguments`

[Function]

`(funcall fn a1 a2 ... an)` applies the function *fn* to the arguments *a1*, *a2*, ..., *an*. *fn* may not be a special form nor a macro; this would not be meaningful.

For example:

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

The difference between `funcall` and an ordinary function call is that the function is obtained by

ordinary LISP evaluation rather than by the special interpretation of the function position that normally occurs.

Compatibility note: This corresponds roughly to the INTERLISP primitive `apply*`.

`funcall* f &rest args` [Function]

`funcall*` is like a cross between `apply` and `funcall`. (`funcall* a1 a2 ... an list`) applies the function `f` to the arguments `a1` through `an` followed by the elements of `list`. Thus we have:

```
(funcall f a1 ... an) <=> (funcall* f a1 ... an '())
(apply f list) <=> (funcall* f list)
```

However, when `apply` or `funcall` fits the situation at hand, it may be stylistically clearer to use that than to use `funcall*`, whose use implies that something more complicated is going on.

```
(funcall* #' + 1 1 1 '(1 1 1)) => 6
```

```
(defun report-error (&rest args)
  (funcall* (function format) error-output args))
```

Compatibility note: ???

## 5.4. Simple Sequencing

`progn {form}*` [Special form]

The `progn` construct takes a number of forms and evaluates them sequentially, in order, from left to right. The values of all the forms but the last are discarded; whatever the last form returns is returned by the `progn` form. One says that all the forms but the last are evaluated for *effect*, because their execution is useful only for the side effects caused, but the last form is executed for *value*.

`progn` is the primitive control structure construct for "compound statements"; it is analogous to `begin-end` blocks in ALGOL-like languages. Many LISP constructs are "implicit `progn`" forms, in that as part of their syntax each allows many forms to be written which are evaluated sequentially, the results of only the last of which are used for anything. clarify

If the last form of the `progn` returns multiple values, then those multiple values are returned by the `progn` form. If there are no forms for the `progn`, then the result is `()`. These rules generally hold for implicit `progn` forms as well.

`prog1 first {form}*` [Special form]

`prog1` is similar to `progn`, but it returns the value of its *first* form. All the argument forms are executed sequentially; the value the first form produces is saved while all the others are executed, and is then returned.

`prog1` is most commonly used to evaluate an expression with side effects, and return a value which must be computed *before* the side effects happen.

For example:

```
(prog1 (car x) (rplaca x 'foo))
```

alters the *car* of *x* to be *foo* and returns the old *car* of *x*.

(See swapt .)

`prog1` always returns a single value, even if the first form tries to return multiple values. A consequence of this is that `(prog1 x)` and `(progn x)` may behave differently if *x* can produce multiple values. See `mvprog1` (page 60).

`prog2 first second {form}*`

[Special form]

`prog2` is similar to `prog1`, but it returns the value of its *second* form. All the argument forms are executed sequentially; the value of the second form is saved while all the other forms are executed, and is then returned.

`prog2` is provided ~~mostly~~ for historical compatibility.

primarily

```
(prog2 a b c ... z) <=> (progn a (prog1 b c ... z))
```

Occasionally it is desirable to perform one side effect, then a value-producing operation, then another side effect; in such a peculiar case `prog2` is fairly perspicuous.

For example:

```
(prog2 (open-a-file) (compute-on-file) (close-the-file))
;value is that of compute-on-file
```

`prog2`, like `prog1`, always returns a single value, even if the second form tries to return multiple values. A consequence of this is that `(prog2 x y)` and `(progn x y)` may behave differently if *y* can produce multiple values.

## 5.5. Environment Manipulation

declaration

`let ({var | (var value)}*) {form}*`

[Macro]

A `let` form can be used to execute a series of forms with specified variables bound to specified values.

For example:

```
(let ((var1 value1)
      (var2 value2)
      ...
      (varn valuen))
  body1
  body2
  ...
  bodyn)
```

first evaluates the expressions *value1*, *value2*, and so on, in that order, saving the resulting values. Then all of the variables *varj* are bound to the corresponding values in parallel; each binding will be a local binding unless there is a `special` (page 72) declaration to the contrary. The expressions *bodyj* are then evaluated in order; the values of all but the last are discarded (that is, the

body of a `let` form is an implicit `progn`). The `let` form returns what evaluating *body<sub>n</sub>* produces (if the body is empty, which is fairly useless, `let` returns `()` as its value). The bindings of the variables disappear when the `let` form is exited.

Instead of a list (*var<sub>j</sub> value<sub>j</sub>*) one may write simply *var<sub>j</sub>*. In this case *var<sub>j</sub>* is initialized to `()`. As a matter of style, it is recommended that *var<sub>j</sub>* be written only when that variable will be stored into (such as by `setq` (page 47)) before its first use. If it is important that the initial value is `()` rather than some undefined value, then it is clearer to write out (*var<sub>j</sub> ()*) or (*var<sub>j</sub> '()*).

Declarations may appear at the beginning of the body of a `let`; they apply to the code in the body *and* to the bindings made by `let`, but not to the code which produces values for the bindings.

The `let` form shown above is entirely equivalent to:

```
((lambda (var1 var2 ... varn)
  body1 body2 ... bodyn)
 value1 value2 ... valuen)
```

but `let` allows each variable to be textually close to the expression which produces the corresponding value, thereby improving program readability.

`let*` (*{var | (var value)}*\*) *{form}*\* [Special form]

`let*` is similar to `let` (page 56), but the bindings of variables are performed sequentially rather than in parallel. This allows the expression for the value of a variable to refer to variables previously bound in the `let*` form.

More precisely, the form:

```
(let* ((var1 value1)
      (var2 value2)
      ...
      (varn valuen))
  body1
  body2
  ...
  bodyn)
```

first evaluates the expression *value<sub>1</sub>*, then binds the variable *var<sub>1</sub>* to that value; then it evaluates *value<sub>2</sub>* and binds *var<sub>2</sub>*; and so on. The expressions *body<sub>j</sub>* are then evaluated in order; the values of all but the last are discarded (that is, the body of a `let*` form is an implicit `progn`). The `let*` form returns the results of evaluating *body<sub>n</sub>* (if the body is empty, which is fairly useless, `let*` returns `()` as its value). The bindings of the variables disappear when the `let*` form is exited.

Instead of a list (*var<sub>j</sub> value<sub>j</sub>*) one may write simply *var<sub>j</sub>*. In this case *var<sub>j</sub>* is initialized to `()`. As a matter of style, it is recommended that *var<sub>j</sub>* be written only when that variable will be stored into (such as by `setq` (page 47)) before its first use. If it is important that the initial value is `()` rather than some undefined value, then it is clearer to write out (*var<sub>j</sub> ()*) or (*var<sub>j</sub> '()*).

Declarations may appear at the beginning of the body of a `let`; they apply to the code in the body *and* to the bindings made by `let`, but not to the code which produces values for the bindings.

`progv` *symbols values* *{form}*\*. [Special form]

`progv` is a special form which allows binding one or more dynamic variables whose names may be determined at run time. The sequence of forms (an implicit `progn`) is evaluated with the dynamic variables whose names are in the list *symbols* bound to corresponding values from the list *values*. (If too few values are supplied, the remaining symbols are bound to `()`. If too many values are supplied, the excess values are ignored.) The results of the `progv` form are those of the last *form*. The bindings of the dynamic variables are undone on exit from the `progv` form. The lists of symbols and values are computed quantities; this is what makes `progv` different from, for example, `let` (page 56), where the variable names are stated explicitly in the program text.

`progv` is particularly useful for writing interpreters for languages embedded in LISP; it provides a handle on the mechanism for binding dynamic variables.

## 5.6. Conditionals

`cond` *{(test {form})\*}*\* [Special form]

The `cond` special form takes a number (possibly zero) of *clauses*, which are lists of forms. Each clause consists of a *test* followed by zero or more *consequents*.

For example:

```
(cond (test-1 consequent-1-1 consequent-1-2 ...)
      (test-2)
      (test-3 consequent-3-1 ...)
      ... )
```

The first clause whose *test* evaluates to non-`()` is selected; all other clauses are ignored, and the consequents of the selected clause are evaluated in order (as an implicit `progn`).

More specifically, `cond` processes its clauses in order from left to right. For each clause, the *test* is evaluated. If the result is `()`, `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, as an implicit `progn`. After evaluating the consequents, `cond` returns without inspecting any remaining clauses. The `cond` special form returns the results of evaluating the last of the selected consequents; if there were no consequents in the selected clause, then the single (and necessarily non-null) value of the *test* is returned. If `cond` runs out of clauses (every test produced `()`, and therefore no clause was selected), the value of the `cond` form is `()`.

If it is desired to select the last clause unconditionally if all others fail, the standard convention is to use `t` for the *test*. As a matter of style, it is desirable to write a last clause “`(t )`” if the value of the `cond` form is to be used for something. Similarly, it is in questionable taste to let the last clause of a `cond` be a “singleton clause”; an explicit `t` should be provided. (Note moreover that `(cond ... (x))` may behave differently from `(cond ... (t x))` if *x* might produce multiple values; the former always returns a single value, while the latter returns whatever values *x* returns.)

For example:

(setq z (cond (a 'foo) (b 'bar)))	; Possibly confusing.
(setq z (cond (a 'foo) (b 'bar) (t ())))	; Better.
(cond (a b) (c d) (e))	; Possibly confusing.
(cond (a b) (c d) (t e))	; Better.
(cond (a b) (c d) (t (values e)))	; Better (if one value is needed)
(cond (a b) (c))	; Possibly confusing.
(cond (a b) (t c))	; Better.
(if a b c)	; Also better.

A LISP `cond` form may be compared to a continued if-then-elseif as found in many algebraic programming languages:

(cond (p ...)		if <i>p</i> then ...
(q ...)	roughly	else if <i>q</i> then ...
(r ...)	corresponds	else if <i>r</i> then ...
...	to	...
(t ...))		else ...

`if pred then [else]`

[Special form]

The `if` special form corresponds to the if-then-else construct found in most algebraic programming languages. First the form *pred* is evaluated. If the result is not `()`, then the form *then* is selected; otherwise the form *else* is selected. Whichever form is selected is then evaluated, and `if` returns whatever evaluation of the selected form returns.

`(if pred then else) <=> (cond (pred then) (t else))`

but `if` is considered more readable in some situations.

The *else* form may be omitted, in which case if the value of *pred* is `()` then nothing is done and the value of the `if` form is `()`. If the value of the `if` form is important in this situation, then the `and` (page 42) construct may be stylistically preferable, depending on the context. If the value is not important, but only the effect, then the `when` (page 46) construct may be stylistically preferable.

`when pred {form}*`

[Special form]

`(when pred form1 form2 ...)` first evaluates *pred*. If the result is `()`, then no *form* is evaluated, and `()` is returned. Otherwise the *forms* constitute an implicit `progn`, and so are evaluated sequentially from left to right, and the value of the last one is returned.

`(when p a b c) <=> (and p (progn a b c))`  
`(when p a b c) <=> (cond (p a b c))`  
`(when p a b c) <=> (if p (progn a b c) '())`  
`(when p a b c) <=> (unless (not p) a b c)`

As a matter of style, `when` is normally used to conditionally produce some side effects, and the value of the `when`-form is normally not used. If the value is relevant, then `and` (page 42) or `if` (page 46) may be stylistically more appropriate.

`unless pred {form}*` [Special form]

`(unless pred form1 form2 ...)` first evaluates `pred`. If the result is `not ()`, then the `forms` are not evaluated, and `()` is returned. Otherwise the `forms` constitute an implicit `progn`, and so are evaluated sequentially from left to right, and the value of the last one is returned.

```
(unless p a b c) <=> (cond ((not p) a b c))
(unless p a b c) <=> (if p '() (progn a b c))
(unless p a b c) <=> (when (not p) a b c)
```

As a matter of style, `unless` is normally used to conditionally produce some side effects, and the value of the `unless`-form is normally not used. If the value is relevant, then `or` (page 42) or `if` (page 59) may be stylistically more appropriate.

*that*

`case keyform {{{key}*} {form}*})*` [Special form]

`case` is a conditional which chooses one of its clauses to execute by comparing a value to various constants, which are typically keyword symbols, integers, or characters (but may be any objects). Its form is as follows:

```
(case keyform
  (keylist-1 consequent-1-1 consequent-1-2 ...)
  (keylist-2 consequent-2-1 ...)
  (keylist-3 consequent-3-1 ...)
  ...)
```

*are duplicates allowed?*

Structurally `case` is much like `cond` (page 58), and it behaves like `cond` in selecting one clause and then executing all consequents of that clause. It differs in the mechanism of clause selection.

The first thing `case` does is to evaluate the form `keyform` to produce an object called the *key object*. Then `case` considers each of the clauses in turn. If `key` is in the *keylist* (that is, is `eq` to any item in the *keylist*) of a clause, the consequents of that clause are evaluated as an implicit `progn`, and `case` returns what was returned by the last consequent (or `()` if there are no consequents in that clause). If no clause is satisfied, `case` returns `()`.

Instead of a *keylist*, one may write *one of the symbols t and otherwise*. A clause with such a symbol always succeeds, and must be the last clause.

Compatibility note: Lisp Machine LISP uses `eq` for the comparison. In Lisp Machine LISP `case` therefore works for fixnums but not bignums. In the interest of hiding the fixnum-bignum distinction, `case` uses `eq1` in COMMON LISP.

There is another problem. It is useful to let `()` as a test mean an empty list, that is, a list of no keys. Such a clause cannot ever be selected. This is mostly useful for macros which want to compute lists of keys, where some lists might turn out to be empty. This is incompatible with LISP systems in which `()` is the same as `nil`, because they typically treat `nil` as a symbol and not as an empty list in this context.

For example:

```
(print (case errorcount
  ((0) '(no errors))
  ((1) '(1 error))
  (()) '(uncountable errors)) ; Cannot be selected.
  ((fatal die) '(fatal error - aborting))
  (t (list errorcount 'errors))))
```

If there is only one key for a clause, then that key may be written in place of a list of that key,

provided that no ambiguity results (the key should not be a cons or one of ( ), t, or otherwise).

`typecase` *keyform* *{(type {form}\*)}*\* [Special form]

`typecase` is a conditional which chooses one of its clauses by examining the type of an object. Its form is as follows:

```
(typecase keyform
  (type-1 consequent-1-1 consequent-1-2 ...)
  (type-2 consequent-2-1 ...)
  (type-3 consequent-3-1 ...)
  ...)
```

Structurally `typecase` is much like `cond` (page 58) or `case` (page 60), and it behaves like them in selecting one clause and then executing all consequents of that clause. It differs in the mechanism of clause selection.

The first thing `typecase` does is to evaluate the form *keyform* to produce an object called the key object. Then `typecase` considers each of the clauses in turn. The first clause for which the key is of that clause's specified *type* is selected, the consequents of this clause are evaluated as an implicit `prog`, and `typecase` returns what was returned by the last consequent (or ( ) if there are no consequents in that clause). If no clause is satisfied, `typecase` returns ( ).

As for `case`, the symbol `t` or `otherwise` may be written for *type* to indicate that the clause should always be selected.

It is permissible for more than one clause to specify a given type, particularly if one is a subtype of another; the earliest applicable clause is chosen.

For example:

```
(typecase an-object
  (string ...)
  ((array t) ...)
  ((array bit) ...)
  (array ...)
  (t ...))
```

This clause handles strings.  
 This clause handles general arrays.  
 This clause handles bit arrays.  
 ; This handles all other arrays.  
 ; This handles all other objects.

A COMMON LISP compiler may choose to issue a warning if a clause cannot be selected because it is completely shadowed by earlier clauses.

## 5.7. Iteration

COMMON LISP provides a number of iteration constructs. The `do` (page 49) and `do*` (page 51) constructs provides a general iteration facility. For simple iterations over lists or *n* consecutive integers, `dolist` (page 52) and related constructs are provided. The `prog` (page 55) construct is the most general, permitting arbitrary `go` (page 57) statements within it. All of the iteration constructs permit statically defined non-local exits in the form of the `return` (page 58) statement and its variants.

## 5.7.1. General iteration

*declarations*

```
do ({(var [init [step]])}*) (end-test {form}*) {tag | statement}* [Special form]
```

The `do` special form provides a generalized iteration facility, with an arbitrary number of "index variables". These variables are bound within the iteration and stepped in parallel in specified ways. They may be used both to generate successive values of interest (such as successive integers) or to accumulate results. When an end condition is met, the iteration terminates with a specified value.

In general, a `do` loop looks like this:

```
(do ((var1 init1 step1)
    (var2 init2 step2)
    ...
    (varn initn stepn))
    (end-test . result)
    . progbody)
```

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 `()` if it is omitted) and a stepping form *step*. If *step* is omitted, the *var* is not changed by the `do` construct between repetitions (though code within the `do` is free to alter the value of the variable by using `setq` (page 47)).

An index-variable specifier can also be just the name of a variable. In this case, the variable has an initial value of `()`, and is not changed between repetitions.

Before the first iteration, all the *init* forms are evaluated, and then each *var* is bound to the value of its respective *init*. This is a binding, not an assignment; when the loop terminates the old values of those variables will be restored. Note that *all* of the *init* forms are evaluated *before* any *var* is bound; hence *init* forms may refer to old values of the variables.

The second element of the `do`-form is a list of an end-testing predicate form *end-test*, and zero or more forms, called the *result* forms. This resembles a `cond` clause. At the beginning of each iteration, after processing the variables, the *end-test* is evaluated. If the result is `()`, execution proceeds with the body of the `do`. If the result is not `()`, the *result* forms are evaluated in order as an implicit `progn` (page 55), and then `do` returns. `do` returns the results of evaluating the last *result* form. If there are no *result* forms, the value of `do` is `()`; note that this is *not* quite analogous to the treatment of clauses in a `cond` (page 58) special form.

At the beginning of each iteration other than the first, the index variables are updated as follows. First every *step* form is evaluated, from left to right. Then the resulting values are assigned (as with `setq` (page 48)) to the respective index variables. Any variable which has no associated *step* form is not affected. Because *all* of the *step* forms are evaluated before *any* of the variables are altered, when a *step* form is evaluated it always has access to the *old* values of the index variables, even if other *step* forms precede it. After this process, the end-test is evaluated as described above.

If the end-test of a `do` form is `()`, the test will never succeed. Therefore this provides an idiom for

"do forever". The *body* of the `do` is executed repeatedly, stepping variables as usual, of course. The infinite loop can be terminated by the use of `return` (page 58), `return-from` (page 58), `go` (page 57) to an outer level, or `throw` (page 65).

For example:

```
(do ((j 0 (+ j 1)))
    (())
    (format t "~%Input ~D:" j)
    (let ((item (read)))
      (if (null item) (return)
          (format t "~&Output ~D: ~S" j (process item))))))
```

; Do forever. OBSOLETE

; Process items until () seen.

The remainder of the `do` form constitutes a `prog` body. The function `return` (page 58) and its variants may be used within a `do` form to terminate it immediately, returning a specified result. Tags may appear within the body of a `do` loop for use by `go` (page 57) statements. When the end of a `do` body is reached, the next iteration cycle (beginning with the evaluation of *step* forms) occurs.

`declare` (page 72) forms may appear at the beginning of a `do` body. They apply to code in the `do` body, to the bindings of the `do` variables, to the *step* forms (but *not* the *init* forms), to the *end-test*, and to the *result* forms.

Compatibility note: "Old-style" MACLISP `do` loops, of the form `(do var init step end-test . body)`, are not supported. They are obsolete, and are easily converted to a new-style `do` with the insertion of three pairs of parentheses. In practice the compiler can catch nearly all instances of old-style `do` loops because they will not have a legal format anyway.

For example:

```
(do ((i 0 (+ i 1))
    (n (array-length an-array))
    ((= i n))
    (aset 'empty an-array i))
    ; Sets every element of an-array to empty
```

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.

Some-times  
~~Very often~~ an iterative algorithm can be most clearly expressed entirely in the *step* forms of a `do`, and the *body* is empty.

For example:

```
(do ((x foo (cdr x))
    (y bar (cdr y))
    (z '() (cons (f (car x) (car y)) z)))
    ((or (null x) (null y))
     (nreverse z)))
```

does the same thing as `(mapcar #'f foo bar)`. Note that the *step* computation for `z` exploits

the fact that variables are stepped in parallel. Also, the body of the loop is empty. Finally, the use of `nreverse` (page 77) to put an accumulated `do` loop result into the correct order is a standard idiom.

Other examples:

```
(defun length (list)
  (do ((x list (cdr x))
      (j 0 (+ j 1)))
      ((atom x) j)))

(defun reverse (list)
  (do ((x list (cdr x))
      (y '() (cons (car x) y)))
      ((atom x) y)))
```

Not sed  
defns how! —

Note the use of `atom` rather than `null` to test for the end of a list in the above two examples. This results in more robust code; it will not attempt to `cdr` the end of a dotted list.

As an example of nested loops, suppose that `env` holds a list of conses. The *car* of each cons is a list of symbols, and the *cdr* of each cons is a list of equal length containing corresponding values. Such a data structure is similar to an association list, but is divided into "frames"; the overall structure resembles a rib-cage. A lookup function on such a data structure might be:

```
(defun ribcage-lookup (sym ribcage)
  (do backbone-loop
      ((r ribcage (cdr r))
       ((null r) ()))
    (do rib-loop
        ((s (caar r) (cdr s))
         (v (cdar r) (cdr v))
         ((null s)
          (when (eq (car s) sym)
              (return-from backbone-loop (car v))))))
```

(Notice the use of indentation in the above example to set off the bodies of the `do` loops.)

`do* bindspecs endtest {form}*`

copy from do [Function]

`do*` is exactly like `do` except that the bindings and steppings of the variables are performed sequentially rather than in parallel. At the beginning each variable is bound to the value of its *init* form before the *init* form for the next variable is evaluated. Similarly, between iterations each variable is given the new value computed by its *step* form before the *step* form of the next variable is evaluated.

## 5.7.2. Simple Iteration Constructs

The constructs `dolist` and `dotimes` perform a body of statements repeatedly. On each iteration a specified variable is bound to an element of interest which the body may examine. `dolist` examines successive elements of a list, and `dotimes` examines integers from 0 to *n* for some specified positive integer

*n*.

minors

The value of any of these constructs may be specified by an optional result form, which if omitted defaults to the value ().

The `return` (page 58) or `return-from` (page 58) statement may be used to return immediately from a `dolist` or `dotimes` form, discarding any following iterations which might have been performed. ~~The loop may be given a name for this purpose by writing it directly before the binding specification.~~ The body of the loop is in fact a `prog` (page 55) body; it may contain tags to serve as the targets of `go` (page 57) statements, and may have `declare` (page 72) forms at the beginning.

*declarations*  
`dolist` (*var* *listform* [*resultform*]) {*tag* | *statement*}\* [Special form]

`dolist` provides straightforward iteration over the elements of a list. The expression (`dolist` (*var* *listform* *resultform*) . *progbody*) evaluates the form *listform*, which should produce a list. It then performs *progbody* once for each element in the list, in order, with the variable *var* bound to the element. Then *resultform* (a single form, *not* an implicit `prog`) is evaluated, and the result is the value of the `dolist` form. If *resultform* is omitted, the result is ().

For example:

```
(dolist (x '(a b c d)) (prin1 x) (princ " ")) => ()
after printing "a b c d"
```

The loop may be named by placing the name before the binding specification. An explicit `return` statement may be used to terminate the loop and return a specified value.

Compatibility note: The *resultform* part of a `dolist` is not currently supported in Lisp Machine Lisp. It seems to improve the utility of the construct markedly.

*dots*  
`dotimes` (*var* *countform* [*resultform*]) {*tag* | *statement*}\* [Special form]

`dotimes` provides straightforward iteration over a sequence of integers. The expression (`dotimes` (*var* *countform* *resultform*) . *progbody*) evaluates the form *countform*, which should produce an integer. It then performs *progbody* once for each integer from zero (inclusive) to *count* (exclusive), in order, with the variable *var* bound to the integer; if the integer is zero or negative, then the *progbody* is performed zero times. Finally, *resultform* (a single form, *not* an implicit `prog`) is evaluated, and the result is the value of the `dotimes` form. If *resultform* is omitted, the result is ().

Altering the value of *var* in the body of the loop (by using `setq` (page 47), for example) will have unpredictable, possibly implementation-dependent results. A COMMON LISP compiler may choose to issue a warning if such a variable appears in a `setq`.

For example:

```
(defun string-posq (char string &optional
                  (start 0)
                  (end (string-length string)))
  (dotimes (k (- end start) '())
    (when (char= char (char string (+ start k)))
      (return k))))
```

The loop may be named by placing the name before the binding specification. An explicit `return`

statement may be used to terminate the loop and return a specified value.

### 5.7.3. Mapping

Mapping is a type of iteration in which a function is successively applied to pieces of one or more sequences. The result of the iteration is a sequence containing the respective results of the function applications. 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.

In COMMON LISP mapping is done by two kinds of constructs: mapping functions and for-loops. Mapping functions take functional arguments and apply them as described above. for-loops are special forms which are often syntactically more convenient; they have bodies, which can refer to a bound variable, and the value of the body provides a result.

<code>mapcar function list &amp;rest more-lists</code>	<code>[Function]</code>
<code>maplist function list &amp;rest more-lists</code>	<code>[Function]</code>
<code>mapc function list &amp;rest more-lists</code>	<code>[Function]</code>
<code>mapl function list &amp;rest more-lists</code>	<code>[Function]</code>
<code>mapcan function list &amp;rest more-lists</code>	<code>[Function]</code>
<code>mapcon function list &amp;rest more-lists</code>	<code>[Function]</code>

For each these mapping functions, the first argument is a function and the rest must be lists. The function must take as many arguments as there are lists.

`mapcar` operates on successive elements of the lists. First the function is applied to the *car* of each list, then to the *cadr* of each list, and so on. (Ideally all the lists are the same length; if not, the iteration terminates when the shortest list runs out, and excess elements in other lists are ignored.) The value returned by `mapcar` is a list of the results of the successive calls to the function.

For example:

```
(mapcar #'abs '(3 -4 2 -5 -6)) => (3 4 2 5 6)
(mapcar #'cons '(a b c) '(1 2 3)) => ((a . 1) (b . 2) (c . 3))
```

~~Often for lists (page FORLISTS-FUN) is more convenient to use than `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.

For example:

```
(maplist #'(lambda (x) (cons 'foo x))
 '(a b c d))
=> ((foo a b c d) (foo b c d) (foo c d) (foo d))
(maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1)))
 '(a b a c d b c))
=> (0 0 1 0 1 1 1)
; An entry is 1 iff the corresponding element of the input
; list was the last instance of that element in the input list.
```

`mapl` and `mapc` are like `maplist` and `mapcar` respectively, except that they do not accumulate

the results of calling the function.

*with*

**Compatibility note:** In all LISP systems since LISP 1.5, `map1` has been called `map`. In the chapter on sequences it is explained why this was a bad choice. Here the name `map` is used for the far more useful generic sequence mapper, in closer accordance to the computer science literature, especially the growing body of papers on functional programming.

These functions are used when the function is being called merely for its side-effects, rather than its returned values. The value returned by `map1` or `mapc` is the second argument, that is, the first sequence argument.

`mapcan` and `mapcon` are like `mapcar` and `maplist` respectively, except that they combine the results of the function using `nconc` (page 92) instead of `list`. That is,

```
(mapcon f x1 ... xn)
=> (apply #'nconc (maplist f x1 ... xn))
```

and similarly for the relationship between `mapcan` and `mapcar`. Conceptually, these functions allow the mapped function to return a variable number of items to be put into the output list. This is particularly useful for effectively returning zero or one item:

```
(mapcan #'(lambda (x) (and (numberp x) (list x)))
      '(a 1 b c 3 4 d 5))
=> (1 3 4 5)
```

*?* *atet*

In this case the function serves as a filter; this is a standard LISP idiom using `mapcan`. (The function `rem-if-not` (page 79) might have been useful in this particular context, however.) Remember that `nconc` is a destructive operation, and therefore so are `mapcan` and `mapcon`; the lists returned by the *function* are altered in order to concatenate them.

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

The functional argument to a mapping function must be acceptable to `apply`; it cannot be a macro or the name of a special form. Of course, there is nothing wrong with using functions which have `&optional` and `&rest` parameters.

There are also functions (`mapatoms` (page MAPATOMS-FUN) and `mapatoms-all` (page MAPATOMS-ALL-FUN)) for mapping over all symbols in certain packages.

#### 5.7.4. The Program Feature

LISP implementations since LISP 1.5 have had what was originally called "the program feature", as if it were impossible to write programs without it! The `prog` construct allows one to write in an ALGOL-like or FORTRAN-like statement-oriented style, using `go` statements which can refer to tags in the body of the `prog`. Contemporary LISP programming style tends to use `prog` rather infrequently. The various iteration constructs, such as `do` (page 62), have bodies with the characteristics of a `prog`.

*ref*

*decls*

`prog` (*{var | (var init)}*\*) (*tag | statement*)\* [Special form]

`prog` is a special form which provides bound temporary variables, sequential evaluation of forms, and a "goto/return" facility. It is this latter characteristic which distinguishes `prog` from other LISP constructs; `lambda` (page LAMBDA-FUN) and `let` (page 56) also provide local variable bindings, and `progn` (page 55) also evaluates forms sequentially.

A typical `prog` looks like:

```
(prog (var1 var2 (var3 init3) var4 (var5 init5))
      statement1
      tag1
      statement2
      statement3
      statement4
      tag2
      statement5
      ...
    )
```

The list after the keyword `prog` is a set of specifications for binding `var1`, `var2`, etc., which are temporary variables, bound locally to the `prog`. This list is processed exactly as the list in a `let` (page 56) statement: first all the *init* forms are evaluated from left to right (where `()` is used for any omitted *init* form), and then the variables are all bound in parallel to the respective results. (`prog*` (page 57) is the same as `prog` except that this initialization is sequential rather than parallel.)

The part of a `prog` after the variable list is called the *body*. An item in the body may be a symbol or a number, in which case it is called a *tag*, or any other COMMON LISP form, in which case it is called a *statement*.

After `prog` binds the temporary variables, it processes each form in its body sequentially. *tags* are ignored; *statements* are evaluated, and their returned values discarded. If the end of the body is reached, the `prog` returns `()`. However, two special forms may be used in `prog` bodies to alter the flow of control. If `(return x)` is evaluated, `prog` stops processing its body, evaluates `x`, and returns the result. If `(go tag)` is evaluated, `prog` jumps to the part of the body labelled with the *tag* (that is, with an atom `eq1` (page 39) to *tag*). *tag* is not evaluated.

Compatibility note: The "computed go" feature of MACLISP is not supported. The syntax of a computed go is idiosyncratic, and the feature is not supported by Lisp Machine LISP, NIL, or INTERLISP.

`go` and `return` forms must be *lexically* within the scope of the `prog`; it is not possible for one function to `return` to a `prog` which is in progress in its caller. Thus, a program which contains a `go` which is not contained within the body of a `prog` (or other constructs such as `do`, which have `prog` bodies) are in error. A dynamically scoped non-local exit mechanism is provided by `catch` (page 63) and `throw` (page 65) and other related operations.

Sometimes code which is lexically within more than one `prog` form needs to `return` from one of the outer `progs`. However, the `return` function normally returns from the innermost `prog`.

Here is a fine example of what can be done with `prog`:

*clarify*

```

(defun king-of-confusion (w)
  (prog (x y z) ; initialize x, y, z to ()
    (setq y (car w) z (cdr w))
    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
      (error "Mismatch - gleep!")
      (setq z y)
      (go rejoin))

```

which is accomplished somewhat more perspicuously by:

```

(defun prince-of-clarity (w)
  (do ((y (car w) (cdr y))
       (z (cdr w) (cdr z))
       (x '() (cons (cons (car y) (car z)) x)))
      ((null y) x)
      (when (null z)
        (error "Mismatch - gleep!")
        (setq z y))))

```

Declarations may appear at the beginning of a prog body; see `declare` (page 72).

### prog\*

*copy from prog* [Special form]

The `prog*` special form is almost the same as `prog`. The only difference is that the binding and initialization of the temporary variables is done *sequentially*, so that the *init* form for each one can use the values of previous ones. Therefore `prog*` is to `prog` as `let*` (page 57) is to `let` (page 56).

For example:

```

(prog* ((y z) (x (car y)))
  (return x))

```

returns the car of the value of z.

### go tag

[Special form]

The `(go tag)` special form is used to do a "go to" within a `prog` body. The *tag* must be a symbol or a number; *tag* is not evaluated. `go` transfers control to the point in the body labelled by a tag equal to the one given. If there is no such tag in the body, the bodies of lexically containing `prog` bodies (if any) are examined as well. It is an error if there is no matching tag.

The `go` form does not ever return a value. A `go` form may not appear as an argument to an ordinary function, but only at the top level of a `prog` body or within certain special forms such as conditionals which are within a `prog` body.

For example:

```
(prog ((n (string-length a-string)) (j 0))
      loop (when (= j n) (return a-string))
          (when (char= #\Space (char j a-string))
              (return (substring a-string 0 j)))
          (increment j)
          (go loop))
```

returns the first "word" in *a-string*, where words are separated by spaces. This could of course have been expressed more succinctly as:

```
(dotimes (j (string-length a-string) a-string)
  (when (char= #\Space (char j a-string))
    (return (substring a-string 0 j))))
```

As a matter of style, it is recommended that the user think twice before using a *go*. Most purposes of *go* can be accomplished with one of the iteration primitives, nested conditional forms, or *return-from* (page 58). If the use of *go* seems to be unavoidable, perhaps the control structure implemented by *go* should be packaged up as a macro definition. (If the use of *go* is avoidable, and *return* also is not needed, then *prog* probably is not needed either; *let* can be used to bind variables and then execute some statements.)

### *return result*

[*Special form*]

*return* is used to return from a *prog*, *do*, or similar iteration construct. Whatever the evaluation of *result* produces is returned by the construct being exited by *return*.

```
(defun member (item list)
  (do ((x list (cdr x))
      ((null x) '())
      (when (equal item (car x))
          (return x))))
```

*return* is, like *go*, a special form which does not return a value. Instead, it causes a containing iteration construct to return a value. If the evaluation of *result* produces multiple values, those multiple values are returned by the construct exited.

If the symbol *t* is used as the name of a *prog*, then it will be made "invisible" to *return* forms; any *return* inside that *prog* will return to the next outermost level whose name is not *t*. (*return-from t ...*) will return from a *prog* named *t*. This feature is not intended to be used by user-written code; it is for macros to expand into.

### *return-from progname result*

[*Special form*]

This is just like *return*, except that before the *result* form is written a symbol (not evaluated), which is the name of the construct from which to return. See the descriptions of the special forms *do* (page 62) and *prog* (page 68) for examples.

## 5.8. Multiple Values

Ordinarily the result of calling a LISP function is a single LISP object. Sometimes, however, it is convenient for a function to compute several quantities and return them. COMMON LISP provides a mechanism for handling multiple values directly. This mechanism is cleaner and more efficient than the usual tricks involving returning a list of results or stashing results in global variables.

### 5.8.1. Constructs for Handling Multiple Values

Normally multiple values are not used. Special forms are required both to *produce* multiple values and to *receive* them. If the caller of a function does not request multiple values, but the called function produces multiple values, then the first value is given to the caller and all others are discarded (and if the called function produces zero values then the caller gets `()` as a value).

The primary primitive for producing multiple values is `values` (page 59), which takes any number of arguments and returns that many values. If the last form in the body of a function is a `values` with three arguments, then a call to that function will return three values. Other special forms also produce multiple values, but they can be described in terms of `values`. Some built-in COMMON LISP functions (such as `floor` (page 128)) return multiple values; those which do are so documented.

The special forms for receiving multiple values are `multiple-value-setq` (page MULTIPLE-VALUE-SETQ-FUN), `multiple-value-let` (page MULTIPLE-VALUE-LET-FUN), `multiple-value-list` (page 60), and `multiple-value-vector` (page 60). These specify a form to evaluate and an indication of where to put the values returned by that form.

`values` &rest *args*

[Function]

Returns all of its arguments, in order, as values.

For example:

```
(defun polar (x y)
  (values (sqrt (+ (* x x) (* y y))) (atan y x)))
(multiple-value-let (r theta) (polar 3.0 4.0)
  (list r theta))
=> (5.0 0.9272952)
```

The expression `(values)` returns zero values.

`values-list` *list*

[Function]

Returns as multiple values all the elements of *list*.

For example:

```
(values-list (list a b c)) <=> (values a b c)
```

`multiple-value-list form` [Special form]

`multiple-value-vector form` [Special form]

`multiple-value-list` evaluates *form*, and returns a list of the multiple values it returned. `multiple-value-vector` is similar, but returns a general vector containing the multiple values.

For example:

```
(multiple-value-list (floor -3 4)) => (-1 1)
(multiple-value-vector (floor -3 4)) => #(-1 1)
```

`mvcall function {form}*` [Special form]

`mvcall` first evaluates *function* to obtain a function, and then evaluates all of the *forms*. All the values of the *forms* are gathered together (not just one value from each), and given as arguments to the function. The result of `mvcall` is whatever is returned by the function.

For example:

```
(mvcall #'(lambda (x y) (+ x y)) (floor 5 3) (floor 7 3)) <=> (+ 1 2 2 1) => 6
(multiple-value-list form) <=> (mvcall #'list form)
```

`mvprog1 form {form}*` [Special form]

`mvprog1` evaluates the first *form* and saves all the values produced by that form. It then evaluates the other *forms* from left to right, discarding their values. The values produced by the first *form* are returned by `mvprog1`. See `prog1` (page 55), which always returns a single value.

`mvlet lambda-list values-form {form}*` [Special form]

`mvlet` evaluates *values-form*, possibly obtaining multiple values, and binds the variables specified in *lambda-list* to these values while the forms in *body* (an implicit `progn`) are evaluated. Whatever is returned by the last form of *body* is returned by `mvlet`.

```
(mvlet (bindings form) . body)
```

does exactly the same thing as

```
(apply #'(lambda (bindings . body) (multiple-value-list form))
```

but using `mvlet` is potentially much more efficient.

`mvsetq lambda-list form` [Special form]

This special form causes the variables in *lambda-list* to get as values the multiple values returned from the evaluation of *form*; the assignment to the variables is as with `setq` (page 47).

The *lambda-list* is allowed to have the full syntax of the binding specifications for a `lambda` expression, including `&optional` and `&rest` keywords. However, this construct performs *assignment* rather than *binding*.

The result of a `mvsetq` form is a single value, the first one returned by *form*, or `()` if *form* returns zero values.

more

flush

`multiple-value-bind` *({var}\*) values-form {form}\* [Special form]*

The *values-form* is evaluated, and each of the variables *var* is bound to the respective value returned by that form. If there are more variables than values returned, extra values of `()` are given to the remaining variables. If there are more values than variables, the excess values are simply discarded. The variables are bound to the values over the execution of the forms, which make up an implicit *progn*.

Compatibility note: This is compatible with Lisp Machine LISP.

For example:

```
(multiple-value-bind (x) (floor 5 3) (list x)) => (1)
(multiple-value-bind (x y) (floor 5 3) (list x y)) => (1 2)
(multiple-value-bind (x y z) (floor 5 3) (list x y z))
=> (1 2 ())
```

In general,

```
(multiple-value-bind (x y z ...) form . body)
<=>
(mvlet (&optional (x ()) (y ()) (z ()) &rest ())
 form . body)
```

`multiple-value variables form [Special form]`

The *variables* must be a list of variables. The *form* is evaluated, and the variables are *set* (not bound) to the values returned by that form. If there are more variables than values returned, extra values of `()` are assigned to the remaining variables. If there are more values than variables, the excess values are simply discarded.

Compatibility note: This is compatible with Lisp Machine LISP.

`multiple-value` always returns a single value, which is the first value returned by *form*, or `()` if *form* produces zero values.

## 5.8.2. Rules for Tail-Recursive Situations

It is often the case that the value of a special form is defined to be the value of one of its sub-forms. For example, the value of a `cond` is the value of the last form in the selected clause. In most such cases, if the sub-form produces multiple values, the original form will also produce all of those values. This *passing-back* of multiple values of course has no effect unless eventually one of the special forms for receiving multiple values is reached.

To be explicit, multiple values can result from a special form under precisely these circumstances:

- `eval` (page EVAL-FUN) returns multiple values if the form given it to evaluate produces multiple values.
- `apply` (page 54), `funcall` (page 54), `funcall*` (page 55), `mvcall` (page 72), `subrcall` (page SUBRCALL-FUN), and `subrcall*` (page SUBRCALL\*-FUN) pass back multiple values from the function applied or called.

- When a `lambda` (page LAMBDA-FUN)-expression is invoked, the function passes back multiple values from the last form of the `lambda` body (which is an implicit `progn`).
- Indeed, `progn` (page 55) itself passes back multiple values from its last form, as does any construct some part of which is defined to be an "implicit `progn`"; these include `prog` (page 58), `let` (page 56), `let*` (page 57), `when` (page 59), `unless` (page 60), `case` (page 60), `typecase` (page 61), `mvlet` (page 72), `mvsetq` (page 72), `multiple-value-bind` (page 73), `multiple-value` (page 73), `catch` (page 63), and `catch-all` (page 63).
- `mvprog1` (page 72) passes back multiple values from its first form. However, `prog1` (page 55) always returns a single value.
- `unwind-protect` (page 64) returns multiple values if the form it protects does.
- `catch` (page 63) returns multiple values if the result form in a `throw` (page 65) exiting from such a `catch` produces multiple values.
- `cond` (page 58) passes back multiple values from the last form of the implicit `progn` of the selected clause. If, however, the clause selected is a singleton clause, then only a single value (the non-`()` predicate value) is returned. This is true even if the singleton clause is the last clause of the `cond`. It is *not* permitted to treat a final clause "`(x)`" as being the same as "`(t x)`" for this reason; the latter passes back multiple values from the form `x`.
- `if` (page 59) passes back multiple values from whichever form is selected (the *then* form or the *else* form).
- `and` (page 42) and `or` (page 42) pass back multiple values from the last form, but not from forms other than the last.
- `do` (page 62), `prog` (page 68), `prog*` (page 69), and other constructs from which `return` (page 70) can return, each pass back the multiple values of the form appearing in `In` addition, `do` passes back multiple values from the last form of the exit clause, exactly as if the exit clause were a `cond` clause.

Among special forms which *never* pass back multiple values are `setq` (page 47), `multiple-value` (page 73), and `prog1` (page 55). A good way to force only one value to be returned from a form `x` is to write `(values x)`.

The most important rule about multiple values, however, is:

No matter how many values a form produces,  
if the form is an argument form in a function call,  
then exactly ONE value (the first one) is used.

For example, if you write `(cons (foo x))`, then `cons` will receive *exactly* one argument, even if `foo` returns two values. Each argument form produces exactly one argument. If such a form returns zero values, `()` is used for the argument. Similarly, conditional constructs which test the value of a form will use exactly one value (the first) from that form and discard the rest, or use `()` if zero values are returned.

*rationale*

## 5.9. Dynamic Non-local Exits

COMMON LISP provides a facility for exiting from a complex process in a non-local, dynamically scoped manner. There are two classes of special forms for this purpose, called *catch* forms and *throw* forms, or simply *catches* and *throws*. A catch form evaluates some subforms in such a way that, if a throw form is executed during such evaluation, the evaluation is aborted at that point and the catch form immediately returns a value specified by the throw. Unlike `block` (page BLOCK-FUN) and `return` (page 70), which allow for so exiting a `block` form from any point lexically within the body of the `block`, the catch/throw mechanism works even if the throw form is not textually within the body of the catch form. The throw need only occur within the extent (time span) of the evaluation of the body of the catch. This is analogous to the distinction between dynamically bound (special) variables and lexically bound (local) variables.

### 5.9.1. Catch Forms

`catch tag {form}*`

[Special form]

The `catch` special form is the simplest catcher. The *tag* is evaluated first to produce an object that names the catch; it may be any LISP object. The *forms* are evaluated as an implicit `prog`, and the results of the last form are returned, except that if during the evaluation of the *forms* a throw should be executed, such that the *tag* of the throw matches (is `eq` to) the *tag* of the `catch`, then the evaluation of the *forms* is aborted and the results specified by the throw are immediately returned from the `catch` expression.

The tag is used to match up throws with catches (using `eq`, not `eq1`; therefore numbers should not be used as catch tags). (`catch 'foo form`) will catch a (`throw 'foo form`) but not a (`throw 'bar form`). It is an error if throw is done when there is no suitable catch (or one of its variants) ready to catch it.

**Compatibility note:** This syntax for `catch` is not compatible with MACLISP. Lisp Machine LISP defines `catch` to be compatible with that of MACLISP, but discourages its use. The definition here is compatible with `( )`.

Lisp Machine LISP defines `*catch` to return four values. This is complicated, implementation-dependent, and not terribly useful. Here we simply define `catch` to be consistent with the standard convention on the interaction of multiple values with implicit `prog` forms, and with a throw "tail-recursing" out of the matching catch, by analogy with `return` and `prog`.

`catch-all catch-function {form}*`

[Special form]

`unwind-all catch-function {form}*`

[Special form]

`catchall` behaves roughly like `catch`, except that instead of a *tag*, a *catch-function* is provided. If no throw occurs during the evaluation of the *forms*, then this behaves just as for `catch`: the `catchall` form returns what is returned from evaluation of the last of the *forms*. `catch-all` will catch *any* throw not caught by some inner catcher, however; if such a throw occurs, then the function is called, and whatever it returns is returned by `catch-all`. The *catch-function* will get one or more arguments; the first argument is always the throw tag, and the other arguments are the thrown results (there may be more than one if the *result* form for the throw produces multiple values).

The `catch-all` is not in force during execution of the *catch-function*. If a throw occurs within the *catch-function*, it will throw to some catch exterior to the `catch-all`. This is useful because the *catch-function* can examine the tag, and if it is not of interest can relay the throw.

```
(catch-all #'(lambda (tag &rest results)
              (caseq tag                               ;Check tag.
                (win (values-list results))           ;If win, return results.
                (lose (cleanup)                       ;If lose, clean up
                     (ferror "Lose lose!"))         ; and signal an error.
                (otherwise                             ;Otherwise relay throw.
                 (throw tag (values-list results))))
            (determine-win-or-lose))
```

`unwind-all` is just like `catch-all` except that the *catch-function* is always called, even if no throw occurs; in that case the first argument (the "tag") to the *catch-function* is `()`, and the other arguments are the results from the last of the *forms*. Often `unwind-protect` is more suitable for a given task than `unwind-all`, however; the choice should be weighed for any particular application.

`unwind-protect` *protected-form* {*cleanup-form*}\* [*Special form*]

Sometimes it is necessary to evaluate a form and make sure that certain side-effects take place after the form is evaluated; a typical example is:

```
(progn (start-motor)
       (drill-hole)
       (stop-motor))
```

The non-local exit facility of Lisp creates a situation in which the above code won't work, however: if `drill-hole` should do a throw to a catch which is outside of the `progn` form (perhaps because the drill bit broke), then `(stop-motor)` will never be evaluated (and the motor will presumably be left running). This is particularly likely if `drill-hole` causes a LISP error and the user tells the error-handler to give up and abort the computation. (A possibly more practical example might be:

```
(prog2 (open-a-file)
       (process-file)
       (close-the-file))
```

where it is desired always to close the file when the computation is terminated for whatever reason.)

In order to allow the above program to work, it can be rewritten using `unwind-protect` as follows:

```
(unwind-protect
  (progn (turn-on-water-start-motor)
         (drill-hole))
  (stop-motor))
```

If `drill-hole` does a throw which attempts to quit out of the `unwind-protect`, then `(stop-motor)` will be executed.

As a general rule, `unwind-protect` guarantees to execute all the *cleanup-forms* before exiting, whether it terminates normally or is aborted by a throw of some kind. `unwind-protect` returns

whatever results from evaluation of the *protected-form*, and discards all the results from the *cleanup-forms*.

## 5.9.2. Throw Forms

*throw tag result*

[*Special form*]

The *throw* special form is the simplest thrower. The *tag* is evaluated first to produce an object called the throw tag. The most recent outstanding catch whose tag matches the throw tag is exited. Some catches, such as a *catch-all*, will match any throw tag; a *catch* matches only if the catch tag is *eq* to the throw tag.

In the process dynamic variable bindings are undone back to the point of the catch, and any intervening *unwind-protect* cleanup code is executed. The *result* form is evaluated before the unwinding process commences, and whatever results it produces are returned from the catch (or given to the *catch-function*, if appropriate).

If there is no outstanding catch whose tag matches the throw tag, no unwinding of the stack is performed, and an error is signalled. When the error is signalled, the outstanding catches and the dynamic variable bindings are those in force at the point of the throw.

**Implementation note:** These requirements imply that throwing must be done by two passes over the control stack. In the first pass one simply searches for a matching catch. In this search every *catch*, *catch-all*, and *unwind-all* must be considered, but every *unwind-protect* should be ignored. On the second pass the stack is actually unwound, one frame at a time, undoing dynamic bindings and outstanding *unwind-protect* in reverse order of creation until the matching catch is reached.

~~Actually, one pass suffi.~~



## Chapter 6

### FUNC



# Chapter 7

## MACRO



## Chapter 8 Declarations

Examples.

Point to  
DEFVAR &  
DEFCONST.

Declarations allow you to specify extra information about your program to the LISP system. All declarations are completely optional and do not affect the meaning of a correct program, with one exception: special declarations do affect the interpretation of variable bindings and references, and so must be specified where appropriate. All other declarations are of an advisory nature, and may be used by the LISP system to aid you by performing extra error checking or producing more efficient compiled code. Declarations are also a good way to add documentation to a program.

### 8.1. Declaration Syntax

`declare {declaration}*`

This form may occur only at top level, or at the beginning of the bodies of certain special forms; that is, a `declare` form not at top level may occur only as a statement of such a form, and all statements preceding it (if any) must also be `declare` forms. If a declaration is found anywhere else an error will be signalled.

Each *declaration* form is a list whose *car* is a keyword specifying the kind of declaration it is. Declarations may be divided into two classes: those that concern the bindings of variables, and those that do not. Those which concern variable bindings apply only to the bindings made by the special form at the head of whose body they appear; if such a declaration appears at top level, it applies to the dynamic value of the variable. For example, in

```
(defun foo (x) (declare (type float x)) ... (let ((x 'a)) ...))
```

the type declaration applies only to the outer binding of `x`, and not to the binding made in the `let`.

Compatibility note: This is different from MACLISP, in which type declarations are pervasive.

The top-level declaration

```
(declare (type float tolerance))
```

specifies that the dynamic value of `tolerance` should always be a floating-point number.

Declarations that do not concern themselves with variable bindings are pervasive, affecting all code in the body of the special form (but not code in any initialization forms used to compute initial values for bound variables).

For example:

Compatibility note: DECLARE forms are not evaluated! see EVAL-WHEN.

In both in interp & compiler

```
(defun foo (x y) (declare (notinline floor)) ...)
```

advises that everywhere within the body of `foo` the function `floor` should not be open-coded, but called as an out-of-line subroutine. Any pervasive declaration made at top level constitutes a universal declaration, always in force unless locally shadowed.

For example:

```
(declare (inline floor))
```

advises that `floor` should normally be open-coded in-line by the compiler (but within `foo` it will be compiled out-of-line anyway, because of the shadowing local declaration to that effect).

For example:

```
(defun (k x)
  (declare (type integer k))
  (let ((j (foo k x))
        (x (* k k)))
    (declare (inline foo) (special x))
    (foo x j)))
```

In this rather nonsensical example, `k` is declared to be of type `integer`. The `inline` declaration applies to the inner call to `foo`, but not to the one to whose value `j` is bound, because that is *code* in the binding part of the `let`. The `special` declaration of `x` causes the `let` form to make a special binding for `x`, and causes the reference to `x` in the body of the `let` to be a special reference. However, the reference to `x` in the first call to `foo` is a local reference, not a special one.

locally {*declare-form*}\* {*form*}\*

[*Special form*]

This special form may be used to make local pervasive declarations where desired. It does not bind any variables, and so cannot be used meaningfully for declarations of variable bindings.

For example:

```
(locally (declare (inline floor))
  (declare (notinline car cdr))
  (floor (car x) (cdr y)))
```

Q: are top-level special decl's pervasive bindings? screws  $\alpha$ -conversion.

## 8.2. Declaration Forms

Here is a list of valid declaration forms for use in `declare`. A construct is said to be "affected" by a declaration if it occurs within the scope of a declaration.

**special** (`special var1 var2 ...`) declares that all of the variables named are to be considered *special*. All variable bindings affected are made to be dynamic bindings, and affected variable references refer to the current dynamic binding rather than the current local binding. (Notice that inner bindings of a variable implicitly shadow a *special* declaration; inner bindings must be explicitly re-declared to be *special*.)

**:type** (`type type var1 var2 ...`) affects only variable bindings, and declares that the specified variables will take on values only of the specified type.

lexical

`@RandomKeyword[type]` (*type var1 var2 ...*) is an abbreviation for (`type type var1 var2 ...`) provided that *type* is one of the following symbols:

null	cons	list	symbol
vector	string	bit-string	array
function	structure	random	character
number	stream	float	string-char
integer	fixnum	bignum	bit
short-float	single-float	double-float	long-float
complex	ratio	readtable	package
sequence			

`@Keyword[ftype]` (*ftype type function1 function2 ...*) declares that the specified functions will be of the functional type *ftype*.

For example:

```
(declare (ftype (function (integer list) t) nth)
         (ftype (function (number) float) sin cos))
```

`@Keyword[function]` (*function name arglist result-type1 result-type2 ...*) is entirely equivalent to (`ftype (function name arglist result-type1 result-type2 ...) name`)

but may be more convenient for some purposes.

For example:

```
(declare (function nth (integer list) t)
         (function sin (number) float)
         (function cos (number) float))
```

`@Keyword[inline]` (*inline function1 function2 ...*) declares that it is desirable for the compiler to open-code calls to the specified functions; that is, the code for a specified function should be integrated into the calling routine, appearing "in line", rather than a procedure call appearing there. This may achieve extra speed at the expense of debuggability (calls to functions compiled in-line cannot be traced, for example). Remember that a compiler is free to ignore this declaration.

`@Keyword[notinline]` (*not inline function1 function2 ...*) declares that it is *undesirable* to compile the specified functions in-line. Remember that a compiler is free to ignore this declaration.

`@Keyword[ignore]` (*ignore var1 var2... varn*) declares that the bindings of the specified variables are never used. It is desirable for a compiler to issue a warning if a variable so declared is ever referred to or is also declared special, or if a variable is lexical, never referred to, and not declared to be ignored.

??? Query: This is a new idea: what do people think? This is more mnemonic than writing `ignore` or `()` for an ignored parameter because you can give a meaningful (and possibly conventional) name. It is more explicit and robust than simply mentioning the variable at the front of the lambda-body; the latter convention prevents the compiler from issuing a warning about a possibly malformed program.

`@Keyword[optimize]` (*optimize quality1 quality2 ...*) advises the compiler that *quality1* should be given greatest attention in producing compiled code, then *quality2*, and so on. The qualities may include speed (of the compiled code), space (both code size and run-time space), and safety (run-time error checking); any qualities not mentioned are assumed to be of lower priority than those mentioned. The

default situation is implementation-dependent, but implementors are encouraged to consider (optimize safety speed space) for the default.

For example:

```
(defun often-used-subroutine (x y)
  (error-check x y)
  (hairy-setup x)
  (locally
   ;; This inner loop really needs to burn.
   (declare (optimize speed))
   (do ((i 0 (+ i 1))
        (z x (cdr z)))
       ((null z)
        (declare (fixnum i))))))
```

??? Query: This is a new idea: what do people think?

An implementation is free to support other (implementation-dependent) declarations as well. On the other hand, a COMMON LISP compiler is free to ignore entire classes of declarations (for example, implementation-dependent declarations not supported by that compiler's implementation!). Compiler implementors are encouraged, however, to program the compiler by default to issue a warning if the compiler finds a declaration of a kind it never uses (as a hedge against spelling errors).

## Chapter 9

### Sequences (Cross-Product Version)

This chapter is one of several parallel chapters on sequences. The assumption is that one will be chosen to define the official set of COMMON LISP sequence functions. Reviewers of this draft are encouraged to compare these parallel chapters carefully.

The type `sequence` encompasses objects of type `list`, and one-dimensional arrays (all one-dimensional arrays, not just vectors). While these all are different data structures with different structural properties leading to different algorithmic uses, they do have a common property: each contains contain an ordered set of elements.

There are many operations which are useful on both lists and one-dimensional arrays because they deal with ordered sets of elements. One may ask the number of elements, reverse the ordering, concatenate two ordered sets to form a larger one, and so on. A set of operations are provided on sequences; these are generic operations, which may be applied to lists, vectors, or arrays:

<code>elt</code>	<code>reverse</code>	<code>map</code>	<code>remove</code>
<code>setelt</code>	<code>nreverse</code>	<code>some</code>	<code>position</code>
<code>subseq</code>	<code>concat</code>	<code>every</code>	<code>scan-over</code>
<code>copyseq</code>	<code>reduce</code>	<code>notany</code>	<code>count</code>
<code>length</code>	<code>left-reduce</code>	<code>notevery</code>	<code>mismatch</code>
<code>fill</code>	<code>right-reduce</code>	<code>merge</code>	<code>maxprefix</code>
<code>replace</code>	<code>sort</code>	<code>nmerge</code>	<code>maxsuffix</code>
			<code>search</code>

The operations in the last column involve search or comparison. Each of these comes in several varieties and two directions. The variety indicates how elements are to be compared; the direction can be either forward or reverse. For example, the `remove` operation has these ten variations:

<u>Forward direction</u>		<u>Reverse direction</u>
<code>remove</code>	<code>remove-from-end</code>	Compare elements using <code>equal</code>
<code>remq</code>	<code>remq-from-end</code>	Compare elements using <code>eq</code>
<code>rem</code>	<code>rem-from-end</code>	Compare elements with user predicate
<code>rem-if</code>	<code>rem-from-end-if</code>	Test elements with user predicate
<code>rem-if-not</code>	<code>rem-from-end-if-not</code>	Test elements with inverse user predicate

**Rationale:** All of these options multiplied out makes for a very large number of functions: This was deemed more perspicuous than passing flags to a smaller number of functions, and more consistent than providing an incomplete set. `rem-from-end-if-not` seems to be a conceptually atomic operation, for example, despite the fact that its name is made from three separate components.

**Compatibility note:** In a few of its string functions, Lisp Machine LISP uses the term `-reverse-` in function names to

indicate that the string is traversed in the backwards direction. Unfortunately, there is a possible confusion with the reversing of the string, which is not quite the same thing. NIL has proposed that the letter b be used, presumably standing for "backwards". Here the suffix `-from-end` is proposed; I believe the meaning of this to be more immediately evident.

`elt` *sequence index* [Function]

This returns the element of *sequence* specified by *index*, which must be a non-negative integer less than the length of the *sequence*. The first element of any sequence has index 0.

`setelt` *sequence index newvalue* [Function]

The object *newvalue* is stored into the component of the *sequence* specified by *index*, which must be a non-negative integer less than the length of the *sequence*. The first element of any sequence has index 0. If *sequence* is a specialized array, then the *newvalue* must be an object that the array can contain.

`subseq` *sequence start &optional end*. [Function]

This returns a subsequence of *sequence*, starting at the element specified by the integer index *start* and going up to, but not including, the element specified by the integer index *end*. The length of the subsequence is therefore *end* minus *start*. If *end* is not specified, it defaults to the length of the *sequence*, meaning that all elements after *start* are included. It is an error if *end* is less than *start*, or if either is less than zero or greater than the length of the string.

`subseq` always allocates a new sequence for a result; it never shares storage with an old sequence. The result subsequence is always of the same type as the argument *sequence*.

`copyseq` *sequence* [Function]

A copy is made of the argument *sequence*; the result is equal to the argument but not eq to it.

`(copyseq x) <=> (subseq x 0)`

but the name `copyseq` is more perspicuous when applicable.

`length` *sequence* [Function]

The number of elements in *sequence* is returned as a non-negative integer. Note that if the *sequence* is an array with a fill pointer, then `length` returns the value of the fill pointer (see `array-active-length` (page 161)), not the total storage space for the array. the total

`fill` *sequence item &optional start end* [Function]

The *sequence* is destructively modified by replacing some or all of its elements with the *item*. The *item* may be any LISP object, but must be a suitable element for the *sequence*. The *item* is stored into all the components of the *sequence*, beginning at the one specified by the index *start*, and up to but not including the one specified by the index *end*. The *start* index defaults to zero, and the end index to the length of the *sequence*. `fill` returns the modified *sequence*.

For example:

```
(setq x (vector 'a 'b 'c 'd 'e)) => #(a b c d e)
(fill x 'z 1 3) => #(a z z d e)
and now x => #(a z z d e)
(fill x 'p) => #(p p p p p)
and now x => #(p p p p p)
```

**replace** *target-sequence source-sequence* &optional *target-start target-end source-start source-end* [*F*]

The *target-sequence* is destructively modified by copying successive elements into it from *source-sequence*. The elements of *source-sequence* must be of a type that may be stored into the *target-sequence*. The leftmost element modified is specified by the index *target-start*, which defaults to zero; the leftmost element copied is specified by the index *source-start*, which also defaults to zero. The index *target-end* limits the region of *target-sequence* which is modified; it defaults to the length of the *target-sequence*. *source-end* limits the region of *source-sequence* which is copied; it defaults to the length of the *source-sequence*. The indices must all be integers and satisfy the relationships

```
(<= 0 target-start target-end (length target-sequence))
(<= 0 source-start source-end (length source-sequence)).
```

The number of elements copied may be expressed as:

```
(min (- target-end target-start) (- source-end source-start))
```

The value returned by **replace** is the modified *target-sequence*.

If *target-sequence* and *source-sequence* are the same object and the region being modified overlaps with the region being copied from, then it is as if the entire source region were copied to another place and only then copied back into the target region.

**reverse** *sequence* [Function]

The result is a new sequence of the same kind as *sequence*, containing the same elements but in reverse order. The argument is not modified.

**nreverse** *sequence* [Function]

The result is a sequence containing the same elements as *sequence* but in reverse order. The argument may be destroyed and re-used to produce the result. The result may or may not be `eq` to the argument, so it is usually wise to say something like `(setq x (nreverse x))`, because simply `(nreverse x)` is not guaranteed to leave a reversed value in *x*.

**concatenate** &rest *sequences* [Function]

The result is a new sequence which contains all the elements of all the sequences in order. All of the sequences are copied from; the result does not share any structure with any of the argument sequences (in this `concatenate` differs from `append`).

The type of the result may depend to some extent on the implementation. As a rule it should be the least general sequence type among those the implementation provides which can contain the elements of all the argument sequences. The implementation must be such that `concatenate` is

associative, in the sense that the elements of the result sequence are not affected by reassociation (but the type of the result sequence may be affected). If no arguments are provided, `concatenate` returns `()`.

`map` *function result-type sequence &rest more-sequences* [Function]

The *function* must take as many arguments as there are sequences provided; at least one sequence must be provided. The result of `map` is a sequence such that element *j* is the result of applying *function* to element *j* of each of the argument sequences. The result sequence is as long as the shortest of the input sequences.

If the *function* has side-effects, it can count on being called first on all the elements numbered 0, then on all those numbered 1, and so on.

The type of the result sequence is specified by the argument *result-type*.

Compatibility note: In MACLISP, Lisp Machine LISP, INTERLISP, and indeed even LISP 1.5, the function `map` has always meant a non-value-returning version. In my opinion they blew it. I suggest that for COMMON LISP this should be corrected, as the names `map` and `reduce` have become quite common in the literature, `map` always meaning what in the past LISP people have called `mapcar`. It would simplify things in the future to make the standard (according to the rest of the world) name `map` do the standard thing. Therefore the old `map` function is here renamed `map1` (page 66).

For example:

```
(map #'- 'list '(1 2 3 4)) => (-1 -2 -3 -4)
(map #'(lambda (x) (if (oddp x) 1 0)) 'bit-string '(1 2 3 4)) =
```

`some` *predicate sequence &rest more-sequences* [Function]

`every` *predicate sequence &rest more-sequences* [Function]

`notany` *predicate sequence &rest more-sequences* [Function]

`notevery` *predicate sequence &rest more-sequences* [Function]

These are all predicates. The *predicate* must take as many arguments as there are sequences provided. The *predicate* is first applied to the elements with index 0 in each of the sequences, and possibly then to the elements with index 1, and so on, until a termination criterion is met or the end of the shortest of the *sequences* is reached.

`some` returns as soon as any invocation of *predicate* returns a non-`()` value; `some` returns that value. If the end of a sequence is reached, `some` returns `()`. Thus as a predicate it is true if *some* invocation of *predicate* is true.

`every` returns `()` as soon as any invocation of *predicate* returns `()`. If the end of a sequence is reached, `every` returns a non-`()` value. Thus as a predicate it is true if *every* invocation of *predicate* is true.

`notany` returns `()` as soon as any invocation of *predicate* returns a non-`()` value. If the end of a sequence is reached, `notany` returns a non-`()` value. Thus as a predicate it is true if *no* invocation of *predicate* is true.

`notevery` returns a non-`()` value as soon as any invocation of *predicate* returns `()`. If the end of

a sequence is reached, `not every` returns `()`. Thus as a predicate it is true if *not every* invocation of *predicate* is true.

**Compatibility note:** The order of the arguments here is not compatible with INTERLISP and Lisp Machine LISP. This is to stress the similarity of these functions to `map`. The functions are therefore extended here to functions of more than one argument, and multiple sequences.

<code>remove item sequence &amp;optional count</code>	[Function]
<code>remq item sequence &amp;optional count</code>	[Function]
<code>rem predicate item sequence &amp;optional count</code>	[Function]
<code>rem-if predicate sequence &amp;optional count</code>	[Function]
<code>rem-if-not predicate sequence &amp;optional count</code>	[Function]
<code>remove-from-end item sequence &amp;optional count</code>	[Function]
<code>remq-from-end item sequence &amp;optional count</code>	[Function]
<code>rem-from-end predicate item sequence &amp;optional count</code>	[Function]
<code>rem-from-end-if predicate sequence &amp;optional count</code>	[Function]
<code>rem-from-end-if-not predicate sequence &amp;optional count</code>	[Function]

The result is a sequence of the same kind as the argument *sequence*, which has the same elements except that those satisfying a certain test have been removed. This is a nondestructive operation; the result is a copy of the input *sequence*, save that some elements are not copied.

For `remove`, an element is removed if *item* is `equal` to it.

For `remq`, an element is removed if *item* is `eq` to it.

For `rem`, an element is removed if *predicate* is true when applied to *item* and an element (in that order).

For `rem-if`, an element is removed if *predicate* is true of it.

For `rem-if-not`, an element is removed if *predicate* is not true of it.

The argument *count*, if supplied, limits the number of elements removed; if more elements than *count* satisfy the test, only the leftmost *count* such are removed.

The `-from-end` variants differ from the others only when *count* is provided; in that case only the rightmost *count* elements satisfying the test are removed.

For example:

```
(remove 4 '(1 2 4 1 3 4 5)) => (1 2 1 3 5)
(remove 4 '(1 2 4 1 3 4 5) 1) => (1 2 1 3 4 5)
(remove-from-end 4 '(1 2 4 1 3 4 5) 1) => (1 2 4 1 3 5)
(rem #'> 3 '(1 2 4 1 3 4 5)) => (4 3 4 5)
(rem-if #'oddp '(1 2 4 1 3 4 5)) => (2 4 4)
(rem-from-end-if #'evenp '(1 2 4 1 3 4 5) 1) => (1 2 4 1 3 5)
```

The result of `remove` and related functions may share with the argument *sequence*; a list result may share a tail with an input list, and the result may be `eq` to the input *sequence* if no elements need to be removed.

<code>position</code> <i>item sequence</i> &optional <i>start end</i>	[Function]
<code>posq</code> <i>item sequence</i> &optional <i>start end</i>	[Function]
<code>pos</code> <i>predicate item sequence</i> &optional <i>start end</i>	[Function]
<code>pos-if</code> <i>predicate sequence</i> &optional <i>start end</i>	[Function]
<code>pos-if-not</code> <i>predicate sequence</i> &optional <i>start end</i>	[Function]
<code>position-from-end</code> <i>item sequence</i> &optional <i>start end</i>	[Function]
<code>posq-from-end</code> <i>item sequence</i> &optional <i>start end</i>	[Function]
<code>pos-from-end</code> <i>predicate item sequence</i> &optional <i>start end</i>	[Function]
<code>pos-from-end-if</code> <i>predicate sequence</i> &optional <i>start end</i>	[Function]
<code>pos-from-end-if-not</code> <i>predicate sequence</i> &optional <i>start end</i>	[Function]

If the *sequence* contains an element satisfying a certain test, then the index within the sequence of the leftmost such element is returned as a non-negative integer; otherwise ( ) is returned.

For `position`, an element passes the test if *item* is `equal` to it.

For `posq`, an element passes the test if *item* is `eq` to it.

For `pos`, an element passes the test if *predicate* is true when applied to *item* and an element (in that order).

For `pos-if`, an element passes the test if *predicate* is true of it.

For `pos-if-not`, an element passes the test if *predicate* is not true of it.

The `-from-end` variants differ in that the index of the *rightmost* element passing the test, if any, is returned.

The implementation may choose to scan the sequence in any order; there is no guarantee on the number of times the test is made. For example, `position-from-end` might scan a list from left-to-right instead of from right-to-left. Therefore it is a good idea for a user-supplied *predicate* to be free of side-effects.

The arguments *start* and *end* limit the search to the specified subsequence; as usual, *start* defaults to zero and *end* to the length of the sequence.

<code>count</code> <i>item sequence</i> &optional <i>start end</i>	[Function]
<code>cntq</code> <i>item sequence</i> &optional <i>start end</i>	[Function]
<code>cnt</code> <i>predicate item sequence</i> &optional <i>start end</i>	[Function]
<code>cnt-if</code> <i>predicate sequence</i> &optional <i>start end</i>	[Function]
<code>cnt-if-not</code> <i>predicate sequence</i> &optional <i>start end</i>	[Function]

The result is always a non-negative integer, the number of elements in the *sequence* satisfying a certain test.

For `count`, an element passes the test if *item* is `equal` to it.

For `cntq`, an element passes the test if *item* is `eq` to it.

For `cnt`, an element passes the test if *predicate* is true when applied to *item* and an element (in that

order).

For `cnt-if`, an element passes the test if *predicate* is true of it.

For `cnt-if-not`, an element passes the test if *predicate* is not true of it.

There is no guarantee on the number of times a user-supplied *predicate* will be called. For example, a tricky implementation for bit-strings might call the predicate once each on the values 0 and 1, assume that those results are valid for all calls on 0 and 1, and then just count the actual bits and return an appropriate result. Therefore it is a good idea for a user-supplied *predicate* to be free of side-effects.

The arguments *start* and *end* limit the search to the specified subsequence; as usual, *start* defaults to zero and *end* to the length of the sequence.

<code>mismatch</code>	<i>sequence1</i>	<i>sequence2</i>	&optional	<i>start1</i>	<i>start2</i>	<i>end1</i>	<i>end2</i>	[Function]	
<code>mismatchq</code>	<i>sequence1</i>	<i>sequence2</i>	&optional	<i>start1</i>	<i>start2</i>	<i>end1</i>	<i>end2</i>	[Function]	
<code>mismatch</code>	<i>predicate</i>	<i>sequence1</i>	<i>sequence2</i>	&optional	<i>start1</i>	<i>start2</i>	<i>end1</i>	<i>end2</i>	[Function]
<code>mismatch-from-end</code>	<i>sequence1</i>	<i>sequence2</i>	&optional	<i>start1</i>	<i>start2</i>	<i>end1</i>	<i>end2</i>	[Function]	
<code>mismatchq-from-end</code>	<i>sequence1</i>	<i>sequence2</i>	&optional	<i>start1</i>	<i>start2</i>	<i>end1</i>	<i>end2</i>	[Function]	
<code>mismatch-from-end</code>	<i>predicate</i>	<i>sequence1</i>	<i>sequence2</i>	&optional	<i>start1</i>	<i>start2</i>	<i>end1</i>	<i>end2</i>	[Function]

The arguments *sequence1* and *sequence2* are compared element-wise. If they are of equal length and match in every element, the result is (). Otherwise, the result is a non-negative integer, the index of the leftmost position at which they fail to match; or, if one is shorter than and a matching prefix of the other, the length of the shorter sequence is returned.

For `mismatch`, elements are compared using `equal`.

For `mismatchq`, elements are compared using `eq`.

For `mismatch`, elements are compared by passing an element of *sequence1* and an element of *sequence2* (in that order) to a user-specified *predicate*.

The arguments *start1* and *end1* delimit a subsequence of *sequence1* to be matched, and *start2* and *end2* delimit a subsequence of *sequence2*. As usual, *start1* and *start2* default to zero, *end1* to the length of *sequence1*, and *end2* to the length of *sequence2*. The comparison proceeds by first aligning the left-hand ends of the two subsequences; the index returned is an index into *sequence1*. `mismatch` is therefore not commutative if *start1* and *start2* are not equal.

The `-from-end` variants differ in that the index of the *rightmost* position in which the sequences differ is returned. The (sub)sequences are aligned at their right-hand ends; the last elements are compared, the penultimate elements, and so on. The index returned is again an index into *sequence1*. If the first sequence is a proper suffix of the second, zero is returned; if the second is a proper suffix of the first, the length of the first less the that of the second is returned.

The implementation may choose to match the sequences in any order; there is no guarantee on the

number of times the test is made. For example, `mismatch-from-end` might match a list from left-to-right instead of from right-to-left. Therefore it is a good idea for a user-supplied *predicate* to be free of side-effects.

```
maxprefix sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
maxprefq sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
maxpref predicate sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
maxsuffix sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
maxsuffq sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
maxsuff predicate sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
```

The arguments *sequence1* and *sequence2* are compared element-wise. The result is a non-negative integer, the index of the leftmost position at which they fail to match; or, if one is shorter than and a matching prefix of the other, the length of the shorter sequence is returned. If they are of equal length and match in every element, the result is the length of each.

For `maxprefix`, elements are compared using `equal`.

For `maxprefq`, elements are compared using `eq`.

For `maxpref`, elements are compared by passing an element of *sequence1* and an element of *sequence2* (in that order) to a user-specified *predicate*.

The arguments *start1* and *end1* delimit a subsequence of *sequence1* to be matched, and *start2* and *end2* delimit a subsequence of *sequence2*. As usual, *start1* and *start2* default to zero, *end1* to the length of *sequence1*, and *end2* to the length of *sequence2*. The comparison proceeds by first aligning the left-hand ends of the two subsequences; the index returned is an index into *sequence1*. `maxprefix` is therefore not commutative if *start1* and *start2* are not equal.

The `suffix` and `suff` variants differ in that 1 plus the index of the *rightmost* position in which the sequences differ is returned. The (sub)sequences are aligned at their right-hand ends; the last elements are compared, the penultimate elements, and so on. The index returned is again an index into *sequence1*. If the first sequence is a proper suffix of the second, zero is returned; if the second is a proper suffix of the first, the length of the first less that of the second is returned.

The implementation may choose to match the sequences in any order; there is no guarantee on the number of times the test is made. For example, `maxsuffix` might match lists from left-to-right instead of from right-to-left. Therefore it is a good idea for a user-supplied *predicate* to be free of side-effects.

```
search sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
srchq sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
srch predicate sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
search-from-end sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
srchq-from-end sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
srch-from-end predicate sequence1 sequence2 &optional start1 start2 end1 end2 [Function]
```

A search is conducted for a subsequence of *sequence2* which element-wise matches *sequence1*. If there is no such subsequence, the result is `()`; if there is, the result is the index into *sequence2* of the leftmost element of the leftmost such matching subsequence.

For `search`, elements are compared using `equal`.

For `searchq`, elements are compared using `eq`.

For `search`, elements are compared by passing an element of *sequence1* and an element of *sequence2* (in that order) to a user-specified *predicate*.

The arguments *start1* and *end1* delimit a subsequence of *sequence1* to be matched, and *start2* and *end2* delimit a subsequence of *sequence2* to be searched. As usual, *start1* and *start2* default to zero, *end1* to the length of *sequence1*, and *end2* to the length of *sequence2*.

The `-from-end` variants differ in that the index of the leftmost element of the *rightmost* matching subsequence is returned.

The implementation may choose to cnt the sequence in any order; there is no guarantee on the number of times the test is made. For example, `search-from-end` might cnt a list from left-to-right instead of from right-to-left. Therefore it is a good idea for a user-supplied *predicate* to be free of side-effects.

`sort sequence predicate &optional selector` [Function]

`stable-sort sequence predicate &optional selector` [Function]

The *sequence* is destructively sorted according to an ordering determined by the *predicate*. The *predicate* should take two arguments, and return non-`()` if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then the *predicate* should return `()`.

The `sort` function determines the relationship between two elements by giving keys extracted from the elements to the *predicate*. The function *selector*, when applied to an element, should return the key for that element; the *selector* function defaults to the identity function, thereby making the element itself be the key.

The *selector* function should not have any side effects. A useful example of a *selector* function would be a component selector function for a `defstruct` (page 167) structure, for sorting a sequence of structures.

```
(sort a p s)
<=> (sort a #'(lambda (x y) (p (s x) (s y))))
```

While the above two expressions are equivalent, the first may be more efficient in some implementations for certain types of arguments. For example, an implementation may choose to apply *selector* to each item just once, putting the resulting keys into a separate table, and then sort the parallel tables, as opposed to applying *selector* to an item every time just before applying the *predicate*.

If the *selector* and *predicate* functions always return, then the sorting operation will always terminate, producing a sequence containing the same elements as the original sequence (that is, the result is a permutation of *sequence*). This is guaranteed even if the *predicate* does not really consistently represent a total order. If the *selector* consistently returns meaningful keys, and the *predicate* does reflect some total ordering criterion on those keys, then the elements of the result sequence will conform to that ordering.

The sorting operation performed by `sort` is not guaranteed *stable*, however; elements considered equal by the *predicate* may or may not stay in their original order. The function `stable-sort` guarantees stability, but may be somewhat slower.

The sorting operation may be destructive in all cases. In the case of an array argument, this is accomplished by permuting the elements. In the case of a list, the list is destructively reordered in the same manner as for `nreverse` (page 89). Thus if the argument should not be destroyed, the user must sort a copy of the argument.

Should the *selector* or *predicate* cause an 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.

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.

For example:

```
(defun mostcar (x)
  (if (symbolp x) x (mostcar (car x))))

(sort fooarray #'string-lessp #'mostcar)
```

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))
```

`merge` *sequence1* *sequence2* *predicate* &optional *selector* [Function]

The sequences *sequence1* and *sequence2* are destructively merged according to an ordering determined by the *predicate*. The *predicate* should take two arguments, and return `non-()` if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then the *predicate* should return `()`.

The `merge` function determines the relationship between two elements by giving keys extracted from the elements to the *predicate*. The function *selector*, when applied to an element, should return the key for that element; the *selector* function defaults to the identity function, thereby making the element itself be the key.

The *selector* function should not have any side effects. A useful example of a *selector* function would be a component selector function for a `defstruct` (page 167) structure, for merging a sequence of structures.

If the *selector* and *predicate* functions always return, then the merging operation will always terminate. The result of merging two sequences *x* and *y* is a new sequence *z* such that the length of *z* is the sum of the lengths of *x* and *y*, and *z* contains all the elements of *x* and *y*. If *x1* and *x2* are two elements of *x*, and *x1* precedes *x2* in *x*, then *x1* precedes *x2* in *z*; similarly for elements of *y*. In other words, *z* is an *interleaving* of *x* and *y*.

Moreover, if *x* and *y* were correctly sorted according to the *predicate*, then *z* will also be correctly sorted. If *x* or *y* is not so sorted, then *z* will not be sorted, but will nevertheless be an interleaving of *x* and *y*.

The merging operation is guaranteed *stable*; if two or more elements are considered equal by the *predicate*, then the elements from *sequence1* will precede those from *sequence2* in the result.

**Implementation note:** To guarantee stability, one should give elements of *sequence2* as the *first* argument to the *predicate*, and elements of *sequence1* as the *second* argument.

For example:

```
(merge '(1 3 4 6 7) '(2 5 8) #'<) => (1 2 3 4 5 6 7 8)
```



## Chapter 10

### Sequences (Functional Version)

This chapter is one of several parallel chapters on sequences. The assumption is that one will be chosen to define the official set of COMMON LISP sequence functions. Reviewers of this draft are encouraged to compare these parallel chapters carefully.

The type `sequence` encompasses both lists and one-dimensional arrays. While these are different data structures with different structural properties leading to different algorithmic uses, they do have a common property: each contains contain an ordered set of elements.

There are some operations which are useful on both lists and arrays because they deal with ordered sets of elements. One may ask the number of elements, reverse the ordering, extract a subsequence, and so on. For such purposes COMMON LISP provides a set of generic functions on sequences:

<code>elt</code>	<code>reverse</code>	<code>map</code>	<code>remove</code>	<code>remove-duplica</code>
<code>setelt</code>	<code>nreverse</code>	<code>some</code>	<code>delete</code>	<code>delete-duplica</code>
<code>subseq</code>	<code>concat</code>	<code>every</code>	<code>position</code>	<code>find</code>
<code>copyseq</code>	<code>length</code>	<code>notany</code>	<code>mismatch</code>	<code>substitute</code>
<code>fill</code>	<code>sort</code>	<code>notevery</code>	<code>maxprefix</code>	<code>search</code>
<code>replace</code>	<code>merge</code>		<code>maxsuffix</code>	<code>count</code>

Some of these operations come in more than one version. Such versions are indicated by adding prefixes or suffixes to the basic name of the operation.

If the operation requires testing sequence elements according to some criterion, then the criterion may be specified in one of two ways. The basic operation accepts an item, and elements are tested for being `eq1` to that item. The other version is a functional (indicated by prefixing the name with "f"). The functional takes a predicate and optionally some other arguments. The functional returns a function to perform the desired operation. This function will test elements of its sequence argument by calling the predicate, giving it the sequence element and also any extra arguments originally given to the functional, in that order. As an example,

```
(remove item sequence)
```

returns a copy of *sequence* from which all elements `eq1` to *item* have been removed;

```
((fremove #'equal item) sequence)
```

returns a copy of *sequence* from which all elements `equal` to *item* have been removed;