MOON

CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

This copy to be marked up and returned to Steele.

Spice Lisp Reference Manual

Guy L. Steele Jr.
Scott E. Fahlman

13 August 1981

COMMENTS IN THIS FONT ARE FROM DLW.
— DLW

I REALLY WISH WE COULD STANDARDIZE ON VERB-NOUN EVERYWHERE!

Swiss Cheese Edition
Full of Holes — Very Drafty

Spice Document S061
Keywords and index categories: PE Lisp & DS External
Location of machine-readable file: slm.mss @ CMU

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.

Table of Contents

1. INTRO 3

1.1. Purpose 3
1.2. Notational Conventions 4

2. Data Types 7

2.1. Numbers 8
2.1.1. Integers 8
2.1.2. Floating-point Numbers 10
2.1.3. Ratios 12
2.1.4. Complex Numbers 13
2.2. Characters 13
2.3. Symbols 14
2.4. Lists and Conses 16
2.5. Vectors 17
2.6. Arrays 19
2.7. Structures 19
2.8. Functions 20
2.9. Randoms 20

3. Program Structure 21

3.0.1. Stuff I'm Not Sure Where to Put It Yet 21

4. Predicates 23

4.1. Data Type Predicates 23
4.1.1. Specific Data Type Predicates 26
4.2. Equality Predicates 30
4.3. Logical Operators 32

5. Program Structure 35

5.1. Constants and Variables 36
5.1.1. Reference 36
5.1.2. Assignment 37
5.2. Function Invocation 39
5.3. Simple Sequencing 40
5.4. Environment Manipulation 41
5.5. Conditionals 43
5.6. Iteration 47
5.6.1. General iteration 47
5.6.2. Simple Iteration Constructs 50
5.6.3. Mapping 52
5.6.4. The Program Feature 54
5.7. Multiple Values 58
5.7.1. Constructs for Handling Multiple Values 58
5.7.2. Rules for Tail-Recursive Situations 60
5.8. Non-local Exits 61

5.8.1. Catch Forms 62
5.8.2. Throw Forms 64

6. FUNC 67

7. MACRO 69

8. Declarations 71

8.1. Declaration Syntax 71
8.2. Declaration Keywords 72

9. Symbols 75

9.1. The Property List 75
9.2. The Print Name 78
9.3. Creating Symbols 79

10. Numbers 81

10.1. Predicates on Numbers 82
10.2. Comparisons on Numbers 82
10.3. Arithmetic Operations 84
10.4. Irrational and Transcendental Functions 85
10.5. Type Conversions on Numbers 88
10.6. Logical Operations on Numbers 91
10.7. Byte Manipulation Functions 94
10.8. Random Numbers 95

11. Characters 97

11.1. Predicates on Characters 98
11.2. Character Construction and Selection 101
11.3. Character Conversions 102
11.4. Character Control-Bit Functions 104

12. Sequences 107

13. Manipulating List Structure 123

13.1. Conses 123
13.2. Lists 124
13.3. Alteration of List Structure 130
13.4. Substitution of Expressions 131
13.5. Using Lists as Sets 132
13.6. List-Specific Sequence Operations 138
13.7. Association Lists 141
13.8. Hash Tables 146
13.8.1. Hashing on EQ 147
13.8.2. Hashing on EQUAL 149
13.8.3. Primitive Hash Function 149

14. Strings 151

14.1. String Access and Modification 151
14.2. String Comparison 152

14.3. String Construction and Manipulation 153
14.4. Type Conversions on Strings 154
14.5. Sequence Functions on Strings 155

15. Vectors 161

15.1. Creating Vectors 162
15.2. Functions on General Vectors (Vectors of LISP Objects) 162
15.3. Functions on Bit-Vectors 166
15.4. Functions on Vectors of Explicitly Specified Type 170

16. Arrays 175

16.1. Array Creation 175
16.2. Array Access 177
16.3. Array Information 177
16.4. Array Leaders 178
16.5. Fill Pointers 179
16.6. Changing the Size of an Array 180

17. Structures 183

17.1. Introduction to Structures 183
17.2. How to Use Defstruct 185
17.3. Using the Automatically Defined Macros 186
  17.3.1. Constructor Macros 186
  17.3.2. Alterant Macros 187
17.4. `defstruct` Slot-Options 187
17.5. Options to `defstruct` 188
17.6. By-position Constructor Macros 193
17.7. The `si:defstruct-description` Structure 194

18. EVAL 197

19. Input/Output 199

19.1. Printed Representation of LISP Objects 199
  19.1.1. What the `read` Function Accepts 200
  19.1.2. Sharp-Sign Abbreviations 204
  19.1.3. The Readtable 211
  19.1.4. What the `print` Function Produces 214
19.2. Input Functions 214
  19.2.1. Input from ASCII Streams 214
  19.2.2. Input from Binary Streams 219
  19.2.3. Input Editing 219
19.3. Output Functions 219
  19.3.1. Output to ASCII Streams 219
  19.3.2. Output to Binary Streams 221
19.4. Formatted Output 221
19.5. Querying the User 230
19.6. Streams 234
  19.6.1. Standard Streams 234

19.6.2. Creating New Streams 235
19.6.3. Operations on Streams 236
19.7. File System Interface 237
19.7.1. File Names 237
19.7.2. Opening and Closing Files 240
19.7.3. Renaming, Deleting, and Other Operations 242
19.7.4. Loading Files 243
19.7.5. Accessing Directories 244
20. Errors 245
20.1. Signalling Conditions 245
20.2. Establishing Handlers 246
20.3. Error Handlers 247
20.4. Signalling Errors 249
20.5. Break-points 251
20.6. Standard Condition Names 251
21. The Compiler 253
22. STORAG 255
23. LOWLEV 257
Index 259

List of Tables

Table 2-1: Hierarchy of Numeric Types 9
Table 19-1: Standard Character Syntax Attributes 201
Table 19-2: Standard Sharp-Sign Macro Character Syntax 205
Table 19-3: Standard Readtable Character Attributes 212

Acknowledgements

The contributions of Jon L White, Richard Gabriel, and <others> are hereby gratefully acknowledged. 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 GACHA family of Dover fonts for code. At CMU, at least, GACHA suffers from the swung-dash, uparrow, and backquote deficiencies. 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:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]↑_
'abcdefghijklmnopqrstuvwxyz{|}~
```

I hope to correct these problems somehow in future editions.

—Guy L. Steele Jr.

Use LPT font family,
available from Moon.
This will be in the next
Xerox font release, I'm told.

Notes to the Swiss Cheese Edition

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

- The necessary type-specific functions for floating-point numbers are not yet included.
- The necessary generic and type-specific functions for complex numbers are not yet included.
- 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`. Wait until we finish redesigning our package system (should happen during September)
- No single entire pass has been made yet to catch inconsistencies.

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

Chapter 1

INTRO

YOU MIGHT MENTION THE BASIC IDEA, NAMELY THAT THIS IS SUPPOSED TO BE A COMMON SUBSET FOR MANY UPWARD-COMPATIBLE DIALECTS.

1.1. Purpose

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

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 porgamming 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).

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.)

Not done to the extent that I would prefer

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.)

~~[illegible]~~

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.

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.

The COMMON LISP documentation is divided into two parts. This document specifies the core language; any system code is permitted to use constructs documented here. The second part is a collection of independent modules; the code in each may use anything in the core language, but may not use another module unless it is carefully and specifically documented to do so.

This seems weird—motivate
Maybe these modules are optional
at Implementation's choice?

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 "`()`". The standard non-false value is "`t`".

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) ==> (setq v (cons x v))
```

means "the result of expanding the macro-call form `(push x v)` is `(setq 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.

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, as shown by these examples:

`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)`.

In functions, these really stand for pieces of data, not pieces of code.

In general, the text of actual code 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. The `&optional` syntax is actually used in COMMON LISP function definitions for this purpose. Parameter lists may also contain "`&rest`", indicating that an indefinite number of arguments may appear.

`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`.

I think using & keywords in special forms is grossly confusing. See new syntax in chinual version 4

I STRONGLY AGREE! THIS AGGRAVATES THE CONFUSION OF THINKING THAT SPECIAL FORMS ARE FUNCTION CALLS! PLEASE FIX!

`sample-special-form` `&rest` *body* [*Special form*]

This evaluates all the forms in *body* as many times as specified by the global variable `sample-variable`. The *body* is an implicit `progn`. `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.

`sample-macro` *var* `&rest` *body* [*Macro*]

This evaluates the forms in *body* (an implicit `progn`) with the variable *var* bound to `43`. `sample-macro` returns what the last form in *body* returns.

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

In the last example, 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.

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 211)) which reads characters from an input stream and interprets them by parsing as representations of LISP objects.

them

?

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

Identical?

'	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.
\	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, `#o105` 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.
'	Accent grave ("backquote") signals that the next expression is a template which may contains 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`.

"big change!"

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 top 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.

period

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~~ THAT are typed, not variables. Any variable can have any LISP object as its value.

class

class

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 26) 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`.

Objects may be roughly divided into the following categories (which are in fact types): `number`, `character`, `symbol`, `list`, `vector`, `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.

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.

- *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 memory limitations. Floating-point numbers of various ranges and precisions are also provided. Some implementations may choose to provide rational numbers (ratios of integers) and Cartesian complex numbers.

  I THINK IT SHOULD BE REQUIRED! IT'S VERY EASY TO DO AND QUITE VALUABLE; IF IT'S OPTIONAL, IT IS FAR LESS USEFUL.

- *Characters* represent printed glyphs such as letters or text formatting operations. Strings are vectors of characters. COMMON LISP provides a rich character set, including ways 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.)

- *Vectors* are sequences represented as a directly indexable row of components. Some vectors 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 vectors are *strings*, which are vectors of characters, and *bit-vectors*, which are vectors which can contain only the integers `0` and `1`.

- *Arrays* are multi-dimensional collections of objects. While vectors have only one axis, and are indexed by a single integer, arrays may have any non-negative number of dimensions, and are indexed by a sequence of integers.

- *Structures* are user-defined record structures, objects which have named components. The `defstruct` (page 181) 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*.

- *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 generally return `t` or `()`.)

Not zero values?

- *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.

2.1. Numbers

There are several kinds of numbers defined in Common Lisp. Not all implementations support all of them; complex and rational numbers may be absent.

DITTO

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.

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

? SHOULDN'T THIS BE "REAL"?

Table 2-1: Hierarchy of Numeric Types

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 representation is important; in particular, it is guaranteed that the length of a vector or any dimension of 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 $[-2^{29}, 2^{29}-1]$.

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`	; $e^{\pi i}$
`15511210043330985984000000.`	; 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

#nnrddddd

means the integer in radix-*nn* notation denoted by the digits *ddddd*. 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:

```
#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.
```

2.1.2. Floating-point Numbers

Generally speaking, a floating-point number is a rational number of the form $(-1)^s*m*2^{e-p}$, where s is a sign bit (0 or 1), p is the precision (in bits) of the floating-point number, m is a positive integer between 2^{p-1} and 2^p-1 (inclusive), and e is an 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*.
- A *big* floating-point number uses a variable-precision representation, which can represent floating-point numbers of arbitrarily large precision and range.

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 three to five decimal places; single floating-point numbers, at least to about seven decimal places; and double floating-point numbers, at least to about twelve decimal places.

In any given implementation the categories may overlap or coincide. For example, short might mean the

Should the words which are
the same as those used in
the IEEE standard mean
the same things?

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, preferably to be roughly equivalent in precision and range to the IEEE floating-point standard single-precision (32-bit) and double-precision (64-bit) types.
>
> 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).

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 decription of floating-point notation. The notation "<x>*" means zero or more occurrences of "x", the notation "<x>$^{+}$" means one or more occurrences of "x", and the notation "<x>$^{?}$" means zero or one occurrences of "x".

<floating-point number> ::= <sign>$^{?}$ <digit>* . <digit>$^{+}$ <exponent>$^{?}$ | <sign>$^{?}$ <digit>$^{+}$. <digit>* <exponent>
<sign> ::= + | -
<digit> ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
<exponent> ::= <exponent marker> <sign>$^{?}$ <digit>$^{+}$
<exponent marker> ::= `e` | `s` | `f` | `d` | `l` | `b` | `E` | `F` | `D` | `S` | `L` | `B`

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`.

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 *integer* zero, not a floating-point number!
`0.0s0`	; A floating-point zero in *short* format.
`3.1415926535897932384d0`	; A *double*-format approximation to π.
`3.1415926535897932384B0`	; A *big*-format approximation to π.
`6.02E+23`	; Avogadro's number, in default format.
`3.1010299957f-1`	; log— 2, in *single* format.
`-0.000000001s9`	; $e^{\pi i}$ in *short* format, the hard way.

The notation described above should suffice for nearly all programs. However, to make it easier to notate exact floating-point constants for machine-dependent algorithms, a floating-point number may be preceded by the same kind of radix specifier used for integers (one of #*nn*R, #O, #B, or #X). In this case *both* the fraction and the exponent are taken to be notated in the specified radix. There is an ambiguity for radices larger than ten, because the exponent marker might be taken to be a digit. This can be avoided by enclosing the exponent marker between a "<" and ">".

For example:

`#o1.61337611067`	; The square root of π, in octal notation.
`#x0.D17217F8`	; ln 2, in hexadecimal notation.
`#x0.D17217<F>8`	; 2^{32}ln 2, *single* format, in hexadecimal.

2.1.3. Ratios

DITTO

The rationals include the integers, and also *ratios* of two integers; ratios may or may not be supported by a COMMON LISP implementation. The canonical 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). It is possible to notate non-canonical ratios, but most arithmetic functions produce rational results in canonical form. Non-canonical notations may or may not be reduced to canonical form when read by the LISP reader.

? ? INDEED.

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 noncanonical 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 #*nn*R, #O, #B, or #X) as for integers.

For example:

`#o-101/75`	; Octal notation for `-65/61`.
`#3r120/21`	; Ternary notation for `15/7`.

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 scalar (integer, floating-point number, or ratio).

Complex numbers may be generally 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 COMPLEX-FUN).)

For example:

```
    #C(3.0s1 2.0s-1)
    #C(5 -3)                                ; A Gaussian integer.
    #C(5/3 7.0)
```

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, "`#\rubout`" (or "`#\RUBOUT`" or "`#\Rubout`", for example) means the rubout 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 97).

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-\β
#\C-M-Return
```

Note that not all COMMON LISP implementations provide for non-zero bits attributes; see `char-font-limit` (page 97).

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

> ??? Query: There is a strong assumption implicit in the definition of the `string-char` type about the way character objects are implemented. Is everyone concerned willing to live with that?

?

2.3. Symbols

Symbols are LISP data objects which serve several purposes and have 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.)

DIFFERS WITH DEFINITION IN LISP MACHINE MANUAL;
SEE REV 3, PAGE 66-67, WHICH IS DEALING WITH ISSUES OF DISEMBODIED PLISTS.

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 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 symbols 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; the print name determines what case is used when printing the symbol. However, the LISP reader normally ignores case when recognizing 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.

If a symbol cannot be notated simply by the characters of its name, because the name contains special characters or because case differences are important for some reason, 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. The use of \ also inhibits case conversion on the following character. If any character in a sequence is preceded by \, then that sequence 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 first letter is definitely lower-case.
                        ;    This might be recognized as "frobboz" or "frOBbOZ",
                        ;    but never as "Frobboz" or "FROBBOZ".
3.14159265\s0           ;The symbol whose name is "3.14159265s0".
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".
|apl\\360|                  ;The name is "apl\360".
|\|\||                      ;Same as \|\|: the name is "||".
```

I don't see how this can work. If you have typed in both Foo and |foo|, then how do these print? with or without slashes? And if you have a list (Foo |foo|) and read back in its printed representation do you always get a list of two non-EQ symbols? Regardless of the order they are first seen? There are a lot of subtleties to this.

??? Query: How do people feel about the following plan?

Some programmers, particularly INTERLISP people, like to use case in interesting ways, and insist on case being preserved. For example, they like to use names such as `GrossMeOut`. (This is hearsay; the INTERLISP manual certainly shows no examples of this.) (All Interlisp system functions are uppercase but you must shift for yourself, or let DWIM correct it, as I understand.)

Anyway, it has been proposed that the internal form of a symbol's print name be not upper-case, but whatever case the symbol was first interned in (and therefore in whatever form it was first typed). So if one says

```
(Defun GrossMeOut (Hackp) (Cond ...))
```

and later types `(grossmeout t)`, this will correctly access the defined function, and `(print 'grossmeout)` will print `GrossMeOut`, not `GROSSMEOUT`.

There is a set of implications here: `intern` must do `string-equal` hashing rather than `string=`. Can use of vertical bars force the existence of distinct symbols differing only in case, and if so which one gets chosen when a symbol is typed whose capitalization differs from any existing one? I think all this can be worked out; what do people think of it?

I AM AGAINST IT. I AGREE THAT THE SUBTLETIES HERE ARE TOO MUCH TO DEAL WITH —HEL

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.

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.

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.

PLEASE PUT IN A NOTE SAYING THAT (SYMBOLP ()) MAY BE T IN SOME IMPLEMENTATIONS. MAYBE PUT THIS IN SMALL TYPE AND EXPLAIN THE HISTORY. OTHERWISE YOU'LL CONFUSE MANY PEOPLE.

2.5. Vectors

A vector is an object which contains a sequence of components. In general, these components may be any LISP data objects. Given a vector and an index, one can extract or replace the component specified by the index. The index is a non-negative integer (in fact, a fixnum); vector indices are always *zero-origin*, which is to say that the valid indices for a vector of length *n* are the integers from 0 to *n*-1.

Vectors and lists are both special kinds of *sequences*. They differ in that any component of a vector 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 a vector takes time linear in the length of the vector.

A *general vector* (sometimes called an *S-expression vector* or a *boxed vector*) is notated just like a list, except that a "#" is written before the left parenthesis. AND YOU CAN'T USE DOTS, OF COURSE.

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.
```

Implementations may provide certain specialized representations of vectors for efficiency in the case where all the components are of the same specialized (typically numeric) type. All implementations provide specialized vectors for the cases when the components are characters or or when the components are always `0` or `1`; these specializations are respectively called *strings* and *bit-vectors*.

The type of a specialized vector is indicated by a list of the symbol `vector` and the type of the components; for example, a specialized representation for vectors with short floating-point parts would be of type `(vector short-float)`. Similarly, the type `string`, while it has its own name for convenience, might also be referred to as `(vector string-char)`. The type `vector` encompasses all vector representations; the particular representation which allows components to be any LISP object is referred to as type `(vector t)`.

Two kinds of specialized vector which are provided by every COMMON LISP implementation are `(vector string-char)` (also called `string`) and `(vector (mod 2))` (also called `bit-vector`). Special notations are provided for these types.

A string 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|"                   ; Truth?
```

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 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. Arrays

An *array* is an object with components arranged according to a rectilinear coordinate system. Like a vector, an array can be accessed quickly (in constant time). Unlike a vector, which has exactly one axis or dimension, an array may be multidimensional. In addition, arrays have other associated information, such as an optional *fill pointer*. Also unlike vectors, arrays may be altered in size after creation.

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.

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 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 177):

```
    (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.

An array with a rank of one is indexed in much the same way as a vector is, using a single index. A one-dimensional array is not the same as a vector, however. They will be in the Lisp Machine.

The notation for arrays is rather complicated. It generally begins with "#*n*A", 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 ???.

2.7. 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 181)). 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.

I don't think structures and arrays ought to spill their guts by default (any more than symbols print their property list)

I AGREE. THIS WOULD BE QUITE UNWELDY IN THE DEFAULT CASE.

THIRD,

2.8. Functions

2.9. 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 usually printed 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 "#<".

Does this mean (typep x 'random) is required to be true of all objects that print with a "#<"?

Chapter 3

Program Structure

3.0.1. Stuff I'm Not Sure Where to Put It Yet

`defvar` *name* `&optional` *initial-value documentation* [*Special form*]

`defvar` is the recommended way to declare the use of a global variable in a program. Placed at top level in a file,

```
(defvar variable)
```

declares `variable` to be a dynamic variable for the sake of compilation, and records its location for the sake of the editor so that you can ask to see where the variable is defined. 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.

`defvar` should be used only at top level, never in function definitions, and only for global 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 `declare` 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.

MUST?

It would be nice to have a way to give documentation without an initial value, while retaining the important simple syntax (defvar var val)

If `defvar` is used in a patch file (see page PATCH-FACILITY) or is a form evaluated with an editor "compile" or "evaluate" command, and there is an *initial-value*, then the variable is always set to it regardless of whether it is already bound.

??? Query: Actually, the rules for this need to be worked out better? Maybe two kinds. one which always initializes and one which doesn't?

Seems confused

defconst *name initial-value* **&optional** *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. If the variable is already bound, an error occurs unless the existing value is ~~equal (page 31)~~ to the specified *initial-value*.

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

Terribly confused!

The rationale for this is that **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. On the other hand, **defconst** declares a constant, whose value will "never" be changed.

Defconst is not a declaration to the compiler that the value will never change and it may be wired into compiled code. If that is desired, there should be a separate declaration for it.

Yes, currently DEFCONST means exactly the opposite - unlike DEFVAR, val is evaluated every time! Maybe the Lisp Machine should change this name, maybe not.

Chapter 4

Predicates

USE "THAT" TO DISTINGUISH WHICH FUNCTIONS YOU MEAN,
USE "WHICH" TO ADD ADDITIONAL INFORMATION.

A *predicate* is a function ~~which~~ THAT tests for some condition involving its arguments and returns `t` if the condition is true, or `()` if it is not true. One may think of a predicate as producing a Boolean value, where `t` stands for *true* and `()` stands for *false*. Conditional control structures such as `cond` (page 43), `if` (page 44), `when` (page 45), and `unless` (page 45) test such Boolean values.

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

The control structures ~~which~~ THAT test Boolean values actually 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 as a test but also for the useful value provided in case of success. An example of a pseudo-predicate is `member` (page 133).

Didn't these used to be called "semipredicates"?

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:

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

In addition, when a structure type is defined using `defstruct` (page 181), 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 types 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 162) 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.

The valid list-format names for data types are:

- `(vector` *type size*`)`: a specialized vector 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-vector` (page 162). *size* must be a non-negative integer, and *type* a valid type label. If *type* is unspecified then `t` is assumed; if *size* is unspecified, then vectors of any size 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-vector`; 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.

Unclear to me why it's useful to have 1-dimensional bit vectors but not 2-D ones. I guess this is an implementation kludge.

- `(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 175). *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).

  For example:

```
(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.
```

- `(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`.

THIS MAKES IT MORE DIFFICULT FOR A PROGRAM TO DETERMINE THE MAX VALUE; YOU NEED A STUPID COND THAT DECIDES WHETHER TO ADD ONE. I'D GET RID OF THE OPTION, BUT I DO NOT FEEL STRONGLY.
OH, I SEE, FLOATING POINT COMPATIBLE. O WELL.

- `(mod` *n*`)`: a non-negative integer less than *n*. This is equivalent to `(integer 0` *n*–*1*`)` or (what is the same thing) `(integer 0 (`*n*`))`.

@Minus

- `(signed-byte` *s*`)`: equivalent to `(integer` -2^{s-1} $2^{s-1}-1$`)`.

rom.

- `(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 COMPLEX-FUN). 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))`.

Cap?

- `(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 a `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 `eql` (page 30) 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 114) is always of type `(or null (integer 0 ()))` (either `()` or a non-negative integer).
- `(and` *type1* *type2* `...)`: the intersection of the specified types.

`typep` *object* `&optional` *type* [*Function*]

`(typep` *object* *type*`)` is a predicate which returns `t` if *object* is of type *type*, or `()` 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.

`(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. 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 TYPECASE-FUN) special form in programs.

What does it return for a structure, or is that undefined?

4.1.1. Specific Data Type Predicates

The following predicates are for testing for individual data types. These predicates return `t` if the argument is of the type indicated by the name of the function, `()` if it is of some other type.

`null` *object* [*Function*]

`null` returns `t` if its argument is `()`, and otherwise returns `()`. This is the same operation performed by the function `not` (page 32); however, `not` is normally used to invert a Boolean value, while `null` is 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` returns `t` if its argument is a symbol, and otherwise returns `()`.

```
(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 '())` => `t`. 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.

PLEASE NOTE EXPLICITLY THAT IN SOME IMPLEMENTATIONS OF COMMON LISP, (SYMBOLP '()) MAY BE T.

`atom` *object* [*Function*]

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

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

`consp` *object* [*Function*]

The predicate `consp` returns `t` if its argument is a cons, and otherwise returns `()`. 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. Yes. Yes. Yes, UCI-LISP uses it.

`listp` *object* [*Function*]

`listp` returns `t` if its argument is a cons or the empty list `()`, and otherwise returns `()`. 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).

or

```
(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. LM will change I CAN'T WAIT!

`numberp` *object* [*Function*]

`numberp` returns `t` if its argument is any kind of number, and otherwise returns `()`.

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

`integerp` *object* [*Function*]

`integerp` returns `t` if its argument is an integer, and otherwise returns `()`.

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

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

the name integerp has

`bigp` *object* [*Function*]

`bigp` returns `t` if *object* is a bignum (a large integer), and otherwise returns `()`.

Note: the distinction between fixnums and bignums is an implementational rather than semantic matter. The set of integers which are fixnums is implementation-dependent. Programs should avoid depending on the distinction.

```
(bigp x) <=> (typep x 'bignum)
```

`fixnump` *object* [*Function*]

`fixnump` returns `t` if *object* is a fixnum (a small integer), and otherwise returns `()`.

Note: the distinction between fixnums and bignums is an implementational rather than semantic matter. The set of integers which are fixnums is implementation-dependent. Programs should avoid depending on the distinction.

```
(fixnump x) <=> (typep x 'fixnum)
```

`rationalp` *object* [*Function*]

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

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

`ratiop` *object* [*Function*]

`ratiop` returns `t` if its argument is a ratio, and otherwise returns `()`.

```
(ratiop x) <=> (typep x 'ratio)
```

`floatp` *object* [*Function*]

`floatp` returns `t` if its argument is a floating-point number, and otherwise returns `()`.

```
(floatp x) <=> (typep x 'float)
```

`short-floatp` *object* [*Function*]

`short-floatp` returns `t` if *object* is a short-format floating-point number, and otherwise returns `()`.

```
(short-floatp x) <=> (typep x 'short-float)
```

Maybe it would be better not to introduce these and leave it to typep.

`single-floatp` *object* [*Function*]

`single-floatp` returns `t` if *object* is a single-format floating-point number, and otherwise returns `()`.

```
(single-floatp x) <=> (typep x 'single-float)
```

`double-floatp` *object* [*Function*]

`double-floatp` returns `t` if *object* is a double-format floating-point number, and otherwise returns `()`.

```
(double-floatp x) <=> (typep x 'double-float)
```

`long-floatp` *object* [*Function*]

`long-floatp` returns `t` if *object* is a long-format floating-point number, and otherwise returns `()`.

```
(long-floatp x) <=> (typep x 'long-float)
```

`characterp` *object* [*Function*]

`characterp` returns `t` if its argument is a character, and otherwise returns `()`.

```
(characterp x) <=> (typep x 'character)
```

In LM this will be the same as fixnump

`stringp` *object* [*Function*]

`stringp` returns `t` if its argument is a string, and otherwise returns `()`.

```
(stringp x) <=> (typep x 'string)
```

`bit-vectorp` *object* [*Function*]

`bit-vectorp` returns `t` if its argument is a bit-vector, and otherwise returns `()`.

```
(bit-vectorp x) <=> (typep x 'bit-vector)
```

`bitp` *object* [*Function*]

`bitp` returns `t` if its argument is a bit (either of the integers 0 or 1), and otherwise returns `()`.

```
(bitp x) <=> (typep x 'bit)
(bitp x) <=> (or (equal x 0) (equal x 1))
```

eql?

`vectorp` `object` FONT [*Function*]

`vectorp` returns `t` if its argument is a vector, and otherwise returns `()`.

```
(vectorp x) <=> (typep x 'vector)
```

IN LM, ALL VECTORS ARE ARRAYS.

`arrayp` *object* [*Function*]

`arrayp` returns `t` if its argument is an array, and otherwise returns `()`.

```
(arrayp x) <=> (typep x 'array)
```

Compatibility note: Lisp Machine LISP defines strings to be arrays. In COMMON LISP, arrays are a particular data type, distinct from both strings and vectors.

`structurep` `object` FONT [*Function*]

`structurep` returns `t` if its argument is a structure, and otherwise returns `()`.

```
(structurep x) <=> (typep x 'structure)
```

`functionp` *object* [*Function*]

`functionp` returns `t` if its argument is suitable for applying to arguments, using for example the `funcall` or `apply` function. Otherwise `functionp` returns `()`.

See Chinual version 4 (optional arg here)

`subrp` *object* [*Function*]

`subrp` returns `t` if its argument is any compiled code object, and otherwise returns `()`.

```
(subrp x) <=> (typep x 'subr)
```

Does this have to be called "subrp"?? If so, put a hysterical (historical) note here.

`closurep` *object* [*Function*]

`closurep` returns `t` if its argument is a closure, and otherwise returns `()`.

4.2. Equality Predicates

COMMON LISP provides a spectrum of predicates for testing for equality of two objects: `eq` (the most specific), `eql`, `equal`, and `equalp` (the most general). `eq` and `equal` have the meanings traditional in LISP. `eql` 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*`) => t` if and only if *x* and *y* are the same object.

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) => ()
(eq 'a 'a) => t
(eq 3 3) might be t or (), depending on the implementation
(eq 3 3.0) => ()
(eq (cons 'a 'b) (cons 'a 'c)) => ()
(eq (cons 'a 'b) (cons 'a 'b)) => ()
(setq x '(a . b)) (eq x x) => t
(eq #\A #\A) might be t or (), depending on the implementation
(eq "Foo" "Foo") => ()
(eq "FOO" "foo") => ()
```

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.

`eql` *x y* [*Function*]

The `eql` predicate returns `t` if its arguments are `eq`, or if they are numbers of the same type with the same value (that is, they are `=` (page 82)), or if they are character objects which represent the same character (that is, they are `char=` (page 100)).

For example:

```
(eql 'a 'b) => ()
(eql 'a 'a) => t
(eql 3 3) => t
(eql 3 3.0) => ()
(eql (cons 'a 'b) (cons 'a 'c)) => ()
(eql (cons 'a 'b) (cons 'a 'b)) => ()
(setq x '(a . b)) (eql x x) => t
(eql #\A #\A) => t
(eql "Foo" "Foo") => ()
(eql "FOO" "foo") => ()
```

`equal` *x y* [*Function*]

The `equal` predicate returns `t` 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 vectors are `equal` if and only if they are of the same length and corresponding components are `equal`.

Two strings are `equal` if they have the same length, and the characters composing them 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.

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`.

For example:

```
(equal 'a 'b) => ()
(equal 'a 'a) => t
(equal 3 3) => t
(equal 3 3.0) => ()
(equal (cons 'a 'b) (cons 'a 'c)) => ()
(equal (cons 'a 'b) (cons 'a 'b)) => t
(setq x '(a . b)) (equal x x) => t
(equal #\A #\A) => t
(equal "Foo" "Foo") => t
(equal "FOO" "foo") => ()
```

To recursively compare only conses, and compare all atoms using `eq`, use `tree-equal` (page 124).

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

Two objects are `equalp` if they are `equal`, or if they are characters and differ only in alphabetic case (that is, they are `char-equal` (page 100)), or if they are numbers and have the same numerical value, even if they are of different types. By this latter characteristic `equalp` complements `lessp` (page LESSP-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 $abs(x-y) \le fuzz*max(abs(x), abs(y))$. 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 return `t`. (See the function = (page 82).)

For example:

```
(equalp 'a 'b) => ()
(equalp 'a 'a) => t
(equalp 3 3) => t
(equalp 3 3.0) => t
(equalp (cons 'a 'b) (cons 'a 'c)) => ()
(equalp (cons 'a 'b) (cons 'a 'b)) => t
(setq x '(a . b)) (equalp x x) => t
(equalp #\A #\A) => t
(equalp "Foo" "Foo") => t
(equalp "FOO" "foo") => t
```

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` returns `t` if *x* is `()`, and otherwise returns `()`. It therefore inverts its argument, interpreted as a Boolean value.

`null` (page 26) 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` `&rest` *forms* [*Special form*]

`(and` *form1* *form2* `... )` evaluates the *forms* one at a time, from left to right. If any *form* evaluates to `()`, `and` immediately returns `()` without evaluating the remaining *forms*. If all the *forms* but the last evaluate non-`()`, `and` returns whetever the last *form* returns. Therefore in general `and` can be used both for logical operations, where `()` stands for *false* and `t` stands 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. (It would also work in this example to write simply

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

This is a matter of Your Taste. I would leave this value judgment out.

but this is stylistically much less tasteful.) Because of the guaranteed left-to-right ordering, `and` is like the **and then** operator in ADA, rather than the **and** operator.

See also `if` (page 44) and `when` (page 45), which are often more appropriate than `and` for conditional purposes.

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

`or &rest` *forms* [*Special form*]

`(or` *form1 form2* `...` `)` evaluates the *forms* 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 all the *forms* but the last evaluate 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 `t` stands 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.

See also `if` (page 44) and `unless` (page 45), which are often more appropriate than `or` for conditional purposes.

From the general definition, one can deduce that `(or` *x*`)` <=> *x*. Also, `(or)` => `()`, which is the identity for this operation.

TPILB

Chapter 5

Program 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 40), 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 41) and `prog2` (page 41).

For looping, COMMON LISP provides the general iteration facility `do` (page 47), 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 personal 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)))
```

THERE NEEDS TO BE A PARAGRAPH OF ENGLISH EXPLAINING THIS CONCEPT.

`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.

,S 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))
```

`symeval` *symbol* [*Function*]

`symeval` returns the current value of the dynamic (special) variable named by *symbol*. An error occurs if the symbol has no value; see `boundp` (page 37) and `makunbound` (page 38).

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

This function is particularly useful for implementing interpreters for languages embedded in LISP. The corresponding assignment primitive is `set` (page 38).

f f

`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 37) and `makunbound` (page 38).

f `symeval` 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 38).

`boundp` *symbol* [*Function*]
`fboundp` *symbol* [*Function*]

`boundp` returns `t` 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 38), `fset` (page 38), `makunbound` (page 38), and `fmakunbound` (page 38).

> **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.

I doubt it, since there is no such thing as an unbound local variable.

5.1.2. Assignment

`setq &rest` *specs* [*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.

page reference

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 &rest` *stuff* [*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 symbols are set to the resulting values. The value of the `psetq` form is (). ||

why? Why not be same as setq? A little random I suppose, but surely more useful than ().

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 47) iteration construct performs a very similar thing when stepping iteration variables.)

`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 37) 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 `progv` (page 43), 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*.

font *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 37) 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:

```
(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.

I doubt it.

5.2. Function Invocation

THAT

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 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).

be

↑ Applying of special forms is used internally in things like trace + advise. I'm not sure what ought to be said here, certainly the naive reader can and be confused by it. But it's wrong to provide no way at all to do it.

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)
```

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 is 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: ???

Requires at least two arguments
(args may not be null)

5.3. Simple Sequencing

`progn &rest` *forms* [*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.

Declarations may appear at the beginning of a `progn` body; see `declare` (page 72). The scope of the declarations is the body of the `progn` form. Any construct which is an implicit `progn` may contain declarations in a similar manner; the scope of such declarations includes any variables bound by the construct. This is described elsewhere for individual constructs.

This is somewhat bogus, since everything else with declarations in its body applies those declarations to the variable bindings it is doing around the body.

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.

This is the first place multiple values are mentioned. Many places earlier are as reasonable as progn to mention them. Flush!

`prog1` *first* `&rest` *others* [*Special form*]

`prog1` is similar to `progn`, but it returns the results of its *first* form. All the argument forms are executed sequentially; whatever 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.

INDEED. IN FACT, THE COMPILER HAS TO GO THROUGH PAIN TO MAKE THIS WORK.

I SEE HOW, BY TRYING TO PUT THIS INTO PROGN, YOU ARE TRYING TO SIMPLIFY THINGS, BUT IT DOESN'T [illegible]

For example:

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

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

`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.

This is inconsistent with Lisp machine, somewhat. Anyway, discuss this in multiple value section.

`prog2` *first second* `&rest` *others* [*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.

```
(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:

UNFORTUNATE EXAMPLE, SINCE THIS SHOULD USE UNWIND-PROTECT.

```
(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.

5.4. Environment Manipulation

`let` *bindings* `&rest` *body* [*Macro*]

? Special form

WHETHER IT IS A MACRO CAN BE LEFT UP TO THE IMPLEMENTATION. AS FAR AS THIS DEFINITION IS CONCERNED, LET IS A SPECIAL FORM.

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)
      ...
      (varm valuem))
  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 bindingunless 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 *bodyn* 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.

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 ... varm)
         body1 body2 ... bodyn)
 value1 value2 ... valuem)
```

Variables alone without inits? People seem to use that a lot in LM

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

YOU MAY NOT HAVE THE SAME VAR TWICE.

`let*` *bindings* `&rest` *body* [*Macro*]

`let*` is similar to `let` (page 41), 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)
       ...
       (varm valuem))
      body1
      body2
      ...
      bodyn)
```

first evaluates the expression *value1*, then binds the variable *var1* to that value; then its evaluates *value2* and binds *var2*; and so on. 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 the results of evaluating *bodyn* (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.

The `let*` form shown above is entirely equivalent to: NO!

```
((lambda (var1)
         ((lambda (var2)
                  ...
                  ((lambda (varm)
                           body1 body2 ... bodyn)
                   valuem) ...)
          value2))
 value1)
```

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

IT'S OK TO HAVE THE SAME VAR TWICE.

??? Query: There is a problem with the interaction of this definition of `let*` with declarations; if one does things in the obvious manner, declarations cannot apply to any variables except *varm*. This seems unfortunate.

INDEED! THIS DEFINITION IS NO GOOD. I WENT THROUGH THIS WHOLE THING WHILE WRITING THE L-MACHINE COMPILER. MY SUGGESTION IS THAT YOU NOT DEFINE IT THIS WAY.

Any suggestions? Don't call it a macro.

`progv` *symbols values* `&rest` *body* · [*Special form*]

`progv` is a special form which allows binding one or more dynamic variables whose names may be determined at run time. The body (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 in the body. 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 41), 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.

5.5. Conditionals

Fo?

`cond` `&rest` *clauses* [*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 (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 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)))                ;poor
(setq z (cond (a 'foo) (b 'bar) (t ())))         ;good
(cond (a b) (c d) (e))                           ;poor
(cond (a b) (c d) (t e))                         ;good
(cond (a b) (c))                                 ;poor
(cond (a b) (t c))                               ;good
(if a b c)                                       ;good
```

This is debatable; it depends on what z really means.

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

```
(cond (p ...)                               if p then ...
      (q ...)            roughly            else if q then ...
      (r ...)          corresponds          else if r then ...
      ...                  to               ...
      (t ...))                              else ...
```

Fo?

`if` *pred then* `&optional` *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 32) construct can do the same thing and may be stylistically preferable, depending on the context. If the value is not important, but only the effect, then the `when` (page 45) construct may be preferable.

fo?

`when` *pred* `&rest` *forms* [*Special form*]

`(when` *pred form1 form2* `... )` first evaluates *pred*. If the result is `()`, 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.

```
(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 not used. If the value is relevant, then `and` (page 32) or `if` (page 44) may be stylistically more appropriate.

ecch (everywhere it appears)!

`unless` *pred* `&rest` *forms* [*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 not used. If the value is relevant, then `or` (page 33) or `if` (page 44) may be stylistically more appropriate.

or does not return the same value. Don't encourage bugs!

`selectq` *key* `&rest` *clauses* [*Special form*]

Where did SELECT go?

I would suggest that SELECTOR be flushed, and both SELECT + SELECTQ allow an optional predicate between the key + the clauses. It defaults to EQL.

`selectq` 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. Its form is as follows:

numbers (no?)

```
(selectq key
  (test-1 consequent-1-1 consequent-1-2 ...)
  (test-2 consequent-2-1 ...)
  (test-3 consequent-3-1 ...)
  ...)
```

Structurally `selectq` is much like `cond` (page 43), 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 `selectq` does is to evaluate the form *key* to produce an object called the key object. Then `selectq` considers each of the clauses in turn. If *key* satisfies the clause's *test*, the consequents of this clause are evaluated as an implicit `progn`, and `selectq` returns what was returned by the last consequent (or `()` if there are no consequents in that clause). If no clause is satisfied, `selectq` signals an error to indicate an invalid key.

Compatibility note: In Lisp Machine Lisp, a `selectq` which runs out of clauses returns `()`. This is easy and stylistically desirable to specify explicitly, using a `t` or `otherwise` clause. It is a very useful debugging feature for a failed `selectq` to signal a correctable error which, when corrected, retries the `selectq` with a new key. This is fairly easy to compile and fairly difficult to program explicitly.

I would prefer that selectq be left compatible and both selectq and cond (as well as select, typecase, etc., etc.) have a uniform syntax to say to do this, which might be a magic word similar to "otherwise" or might be a letter added to the function's name.

A *test* may be a symbol, a character, or an integer, in which case the test succeeds if the key object is `eql` (page 30) to the test; or it may be a list containing symbols, characters, integers, and/or `()`, in which case the test succeeds if the key object is `eql` to any element of the list. The symbols `t` and `otherwise` are special test keywords which always succeed, and should be used only in the last clause. To actually test for the key object being `t` or `otherwise`, one may put the symbol in a list. To test for the empty list one must always put `()` in a list.

Perhaps explain list case first then say singleton list may omit the parens if unambiguous

Compatibility note: Lisp Machine Lisp uses `eq` for the comparison. In Lisp Machine Lisp `selectq` therefore works for fixnums but not bignums. In the interest of hiding the fixnum-bignum distinction, `selectq` uses `eql` 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.

(This seems like a bug that can easily be corrected if desired, not anything fundamental. I don't see that it has much to do with whether or not ( ) is a symbol in either case eql is a sensible operation on ( ).

For example:

HEY, THIS IS THE 1980'S; WE HAVE STRINGS, DON'T WE?

```
(print (selectq errorcount
                (0 '(no errors))
                (1 '(1 error))
                (() '(uncountable errors))
                (fatal '(fatal error - aborting))
                (t (list errorcount 'errors))))
```

`caseq` *key* `&rest` *clauses* [*Special form*]

`caseq` is identical in syntax to `selectq`, and similar in meaning. It is slightly more akin to the case construct found in most algebraic languages, in that the objects tested for must all be of the same type. That is, excepting the special keywords `t` and `otherwise`, all objects appearing in a clause test in a `caseq` must be either all integers, all characters, or all either symbols or `()`. The compiler, if not the interpreter, will enforce this restriction. Moreover, the *key* for `caseq` must be of the same type as the objects of the clause tests; `selectq` has no such requirement. It is not permitted to perform a `caseq` with a cons for the *key*; a `selectq`, on the other hand, will readily accept a list and select the `otherwise` clause, if any.

integers or numbers?

This why I usually prefer to put in the parentheses even for singleton lists.

Implementation note: This construct is included because in certain kinds of LISP implementations `caseq` can be compiled more efficiently than `selectq`. MACLISP is one such.

`typecaseq` `&rest` *clauses* [*Special form*]

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

poor word choice

```
(selectq key
   (type-1 consequent-1-1 consequent-1-2 ...)
   (type-2 consequent-2-1 ...)
   (type-3 consequent-3-1 ...)
   ...)
```

?

Somewhere earlier in the manual this is called "typecase"

Structurally `typecaseq` is much like `cond` (page 43) or `selectq` (page 45), 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 `typecaseq` does is to evaluate the form *key* to produce an object called the key object. Then `typecaseq` 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 `progn`, and `typecaseq` returns what was returned by the last consequent (or `()` if there are no consequents in that clause). If no clause is satisfied, `typecaseq` signals an error to indicate an invalid key. As for `selectq`, 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:

semicolon

```
(typecaseq an-object
   (:string ...)                    ;This clause handles strings.
   ((:vector t) ...)                 This clause handles general vectors.
   (:vector ...)                    ;This handles all other vectors.
   (t ...))                         ;This handles all other objects.
```

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

5.6. Iteration

COMMON LISP provides a number of iteration constructs. The `do` (page 47) and `do*` (page 50) constructs provides a general iteration facility. For simple iterations over lists, vectors, or *n* consecutive integers, `dolist` (page 51) 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 57) statement and its variants.

5.6.1. General iteration

`do` *bindspecs endtest* `&rest` *progbody* [*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 37)).

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 40), 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 the value of the *end-test*; this is analogous to the treatment of clauses in a `cond` (page 43) special form.

> **Compatibility note:** Other LISP systems which have this `do` facility return `()` if there are no result forms. I know of no code which depends on this, and many users have asked that the value of the end-test be returned.

Many users will be pissed at this, since their code will stop working with no error indication from the system. This is just the sort of change which causes the most weeping, wailing, and gnashing of teeth.

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 `psetq` (page 38)) 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.

The reason for this was probably only that the current behavior is much easier to implement with a macro

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 57), `go` (page 57) to an outer level, or `throw` (page 64).

> **Compatibility note:** MACLISP and related dialects also permit the end-test clause to be `()` (as opposed to `(())`), meaning to perform exactly one iteration of the body. This is an obsolete crock, and should no longer be in use. BUT IT IS UPWARD-COMPATIBLE, RIGHT? I'D LIKE TO KEEP THIS STUPID CROCK, FOR COMPATIBILITY, THOUGH I COULD BE CONVINCED.

The remainder of the `do` form constitutes a `prog` body. The function `return` (page 57) 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. `declare` forms may also appear at the beginning of the *result* forms list, and apply only to the *result* forms.

A `do` loop may be given a name for use in `return-from` (page 58) statements by placing the name after the keyword `"do"` and before the variable specifications.

> **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.

Except you just broke that simple conversion by making it return the value of end-test if there are no results. Have to say nil explicitly now.

For example:

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

put something here other than x -
I think is confusing.

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.

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 110) 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)))
```

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.)

Note

do* *bindspecs endtest* **&rest** *body* [*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.

5.6.2. Simple Iteration Constructs

The constructs `dolist`, `dovector`, `dostring`, 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, `dovector` examines successive elements of a vector, `dostring` examines successive characters of a string, and `dotimes` examines integers from *0* to *n-1*, for some specified positive integer *n*.

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 57) or `return-from` (page 58) statement may be used to return immediately from a `dolist`, `dovector`, `dostring`, 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.

word choice?

dolist *bindspec* **&rest** *progbody* [*Special form*]

`dolist` provides straightforward iteration over the elements of a list. The expression `(dolist (`*var list result*`) . `*progbody*`)` evaluates the form *list*, 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 *result* is evaluated, and the result is the value of the `dolist` form. If *result* is omitted, the result is `()`.

For example:

Is result just one form, leaving space for possible future expansion (I am dubious), or is it an implicit progn? Say explicitly because users (and implementors!) [illegible] assume one or the other

```
(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 *result* part of a `dolist` is not currently supported in Lisp Machine LISP. It seems to improve the utility of the construct markedly.

`dovector` *bindspec* `&rest` *progbody* [*Special form*]

`dovector` provides straightforward iteration over the elements of a vector. The expression `(dovector (`*var vector result*`) . `*progbody*`)` evaluates the form *vector*, which should produce a vector. It then performs *progbody* once for each element in the vector, in order, with the variable *var* bound to the element. Then *result* is evaluated, and the result is the value of the `dovector` form. If *result* is omitted, the result is `()`.

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.

`dostring` *bindspec* `&rest` *progbody* [*Special form*]

`dostring` provides straightforward iteration over the characters of a string. The expression `(dostring (`*var string result*`) `*progbody*`)` evaluates the form *string*, which should produce a string. It then performs *progbody* once for each character in the string, in order, with the variable *var* bound to the character. Then *result* is evaluated, and the result is the value of the `dostring` form. If *result* is omitted, the result is `()`.

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.

`dotimes` *bindspec* `&rest` *progbody* [*Special form*]

`dotimes` provides straightforward iteration over a sequence of integers. The expression `(dotimes (`*var count result*`) `*progbody*`)` evaluates the form *count*, which should produce a positive 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. Then *result* is evaluated, and the result is the value of the `dotimes` form. If *result* is omitted, the result is `()`.

What about zero? Should negative be defined to iterate no times? probably better than error.

ZERO AND NEGATIVE SHOULD BE DEFINED TO ITERATE NO TIMES.

Altering the value of *var* in the body of the loop (by using `setq` (page 37), for example) will not affect the number of times the loop is performed or the values of *var* on succeeding iterations.

Complete loss — define it to be undefined. This complicates the implementation to provide something useless to the user.

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))))
```

Poor indentation

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.

5.6.3. 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.

`mapcar` *function* `&rest` *lists* [*Function*]
`maplist` *function* `&rest` *lists* [*Function*]
`mapc` *function* `&rest` *lists* [*Function*]
`mapl` *function* `&rest` *lists* [*Function*]
`mapcan` *function* `&rest` *lists* [*Function*]
`mapcon` *function* `&rest` *lists* [*Function*]

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 `forlists` (page 54) 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) (not (member (car x) (cdr x))))
         '(a b a c d b c))
   => (() () t () t t t)
```

; An entry is `t` 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.

> Compatibility note: In all LISP systems since LISP 1.5, `mapl` 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.

OK

These functions are used when the function is being called merely for its side-effects, rather than its returned values. The value returned by `mapl` or `mapc` is `t`.

This seems like a pointless incompatibility.

> Compatibility note: In MACLISP and Lisp Machine LISP, these functions return the first argument. This is almost never useful, and makes them inconvenient to use at top level.

Often `dolist` (page 50) is more convenient to use than `mapc`.

Schizoid

`mapcan` and `mapcon` are like `mapcar` and `maplist` respectively, except that they combine the results of the function using `nconc` (page 128) 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)
```

In this case the function serves as a filter; this is a standard LISP idiom using `mapcan`. (The function `rem-if-not` (page 113) 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.

I REALLY THINK THE KEYWORD-ORIENTED "LOOP" IS PREFERABLE TO HAVING ALL THESE RANDOM SPECIAL CASES. BUT IF YOU REALLY WANT THEM IN, IT'S OK.

`forlist` *bindspec* `&rest` *body* [*Special form*]
`forvector` *bindspec* `&rest` *body* [*Special form*]
`forstring` *bindspec* `&rest` *body* [*Special form*]

`forlist` provides mapping over the elements of a single list, accumulating the results of an expression. The expression `(forlist (`*var list*`) . `*body*`)` evaluates the form *list*, which should produce a list. It then performs *body* (an implicit `progn`) once for each element in the list, in order, with the variable *var* bound to the element. The values of the last expression in the body on each iterations are made into a list, and this list of results is the value of the `forlist` expression.

For example:

```
(forlist (x '(1 2 3)) (print x) (* x (+ x 3))) => (4 10 18)
    after printing the numbers 1, 2, and 3.
```

The `forlist` construct is closely related to the `dolist` (page 50) construct. Unlike the `dolist` construct, however, `forlist` does not permit an explicit result form, and may not be exited using the `return` construct (the body of a `forlist` is a `progn` body, not a `prog` body).

`forvector` is similar, but accepts a vector and returns a vector. The result vector is always a general vector (that is, of type `(vector t)`.) That seems like an unnecessary restriction.

`forstring` is similar, but accepts a string and returns a string.

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

In general, these are crocks. Suppose I want to map down a string and return a bitvector? (certainly a frequent thing to want to do)

`forlists` *bindspecs* `&rest` *body* [*Special form*]
`forvectors` *bindspecs* `&rest` *body* [*Special form*]
`forstrings` *bindspecs* `&rest` *body* [*Special form*]

`forlists` provides mapping over the elements of *several* lists, accumulating the results of an expression. The expression

```
(forlists ((var1 list1)
           (var2 list2)
           ...)
           (varn listn))
   . body)
```

?

What's the point of having both forlist and forlists?

evaluates the forms *listj*, which should each produce a list. It then performs *body* (an implicit `progn`) once for each element in the lists, in order, with each variable *var* bound to an element of the corresponding list. The values of the last expression in the body on each iterations are made into a list, and this list of results is the value of the `forlist` expression. If the input lists are of different lengths, the iteration terminates as soon as the shortest one runs out.

For example:

```
(forlists ((x '(a b c)) (y '(1 2 3))) (list 'foo x y))
  => ((foo a 1) (foo b 2) (foo c 3))
```

`forvectors` is similar, but accepts vectors and returns a vector. The result vector is always a general vector (that is, of type `(vector t)`.)

`forstrings` is similar, but accepts strings and returns a string.

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

5.6.4. 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 47), have bodies with the characteristics of a `prog`.

`prog` *[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 41) also provide local variable bindings, and `progn` (page 40) also evaluates forms sequentially.

A typical `prog` looks like:

```
(prog (var1 var2 ((&special 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 41) 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 `eql` (page 30) 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 62) and `throw` (page 64) and other related operations.

What about
labels functions —
returning from progs
in their parent?

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`. A `prog` may be given a *name* by which it may be referenced by a function called `return-from` (page 58), which is similar to `return` but allows a particular `prog` to be specified. A name is a symbol which is written after the keyword `prog` and before the list of variable bindings.

For example:

```
(prog outer (foo bar greps)
      ...
      (prog inner (foo baz snert)
            ...
            (return-from outer (cons baz greps))
            ...)
      ...)
```

See the description of `return-from` (page 58) for more information on the use of named `prog` forms.

poor indentation

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

```
(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*` [*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 42) is to `let` (page 41).

For example:

poor indent

```
(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 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 or nested conditional forms. 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 47) and `prog` (page 55) for examples.

5.7. 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.

5.7.1. 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 89)) return multiple values; those which do are so documented.

The special forms for receiving multiple values are `multiple-value-setq` (page 59), `multiple-value-let` (page 59), `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*]
`values-vector` *vector* [*Function*]

Returns as multiple values all the elements of *list* or *vector*, as the case may be.

For example:

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

`multiple-value-let` *lambda-list form* `&rest` *body* [*Special form*]

I would prefer with-values ((var...) form) body ...)

`multiple-value-let` evaluates *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 `multiple-value-let`.

(`multiple-value-let` *bindings form* `.` *body*)

does exactly the same thing as

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

but using `multiple-value-let` is much more efficient.

`multiple-value-setq` *lambda-list form* [*Special form*]

This name is not an improvement!

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 37).

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 `multiple-value-setq` form is a single value, that assigned to the *first* variable, or `()` if no variables are mentioned in the *lambda-list* (an odd thing to do, but legal).

??? Query: Fooey. Why not just say it returns ( )?

`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 vector containing the multiple values.

For example:

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

This is similar to the example of `multiple-value` (page MULTIPLE-VALUE-FUN) above.

It's not similar to anything anywhere nearby

5.7.2. 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.

> ??? Query: The Lisp Machine LISP manual states: "The exact rule governing passing-back of multiple values is as follows: If *X* is a form, and *Y* is a sub-form of *X*, then if the value of *Y* is unconditionally returned as the value of *X*, with no intervening computation, then all the multiple values returned by *Y* are returned by *X*. In all other cases, multiple values or only single values may be returned at the discretion of the implementation; users should not depend on this. The reason we don't guarantee non-transmission of multiple values is because such a guarantee would not be very useful and the efficiency cost of enforcing it would be high. Even `setq`'ing a variable to the result of a form, then returning the value of that variable might be made to pass multiple values by an optimizing compiler which realized that the `setq`ing of the variable was unnecessary."
>
> I'm not sure the implementation should be allowed this caprice. In particular, a compiler smart enough to optimize out a `setq` can just as well leave behind code to enforce the single-value-returning semantics. I believe it is more important to have a dependable definition here.
>
> Opinions? For now the following documentation makes some clear requirements. These are not incompatible with Lisp Machine LISP, but merely requirements on implementations to make certain choices which Lisp Machine LISP leaves open.

Does this mean no implementation is allowed to have an optimizing compiler?

It's debatable that these restrictions are useful. ~~Why not [illegible] that [illegible] only [illegible]~~

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 39), `funcall` (page 39), `funcall*` (page 40), `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 40) itself passes back multiple values from its last form, as does any construct defined to be an "implicit `progn`"; these include `progv` (page 43), `let` (page 41), `let*` (page 42), `when` (page 44), `unless` (page 45), `selectq` (page 45), `caseq` (page 46), `catch` (page 62), `*catch` (page 62), and `catchall` (page 63).

  > ??? Query: Should `prog1` (page 40) and `prog2` (page 41) return multiple values or not? It can be tricky to compile. Lisp Machine LISP causes them to return single values only. In SPICE LISP it happens to be easier to return multiple values. On the S-1 the issue is unclear.

  This is an example of why it should be undefined.

- `unwind-protect` (page 63) returns multiple values if the form it protects does.

- `catch` (page 62) and `*catch` returns multiple values if the result form in a `throw` (page 64) or `*throw` (page 64) exiting from such a catch produces multiple values.

  ~~[illegible] if not~~

- `cond` (page 43) 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`.

> **Compatibility note:** Lisp Machine LISP permits the implementation to return either one value or multiple values for a singleton `cond` clause.

- `if` (page 44) passes back multiple values from whichever form is selected (the *then* form or the *else* form).

- `and` (page 32) and `or` (page 33) pass back multiple values from the last form, but not from forms other than the last.

- `do` (page 47), `prog` (page 55), `prog*` (page 56), and other constructs from which `return` (page 57) can return, each pass back the multiple values of the form appearing in the `return` (page 57) or `return-from` (page 58) that returns from it.

> **Compatibility note:** Lisp Machine LISP permits the implementation to return one value or multiple values in this case. To force several values to be returned from a `prog` (page 55), one must use `return-list`, `multiple-value-return`, or `return` or `return-from` with several arguments. With the rule laid down here, one can get these effects as follows:
>
> | Lisp Machine LISP | COMMON LISP |
> |---|---|
> | `(return-list x)` | `(return (values-list x))` |
> | `(multiple-value-return x)` | `(return x)` |
> | `(return x y z)` | `(return (values x y z))` |
>
> Actually, Lisp Machine LISP may soon go this way also anyway?

Obsolete — flush.

LISP MACHINE LISP IS UPWARD COMPATIBLE WITH COMMON LISP AS OF ABOUT A MONTH AGO.

- `do` (page 47), as mentioned above, behaves like `prog` with respect to `return`. 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 37), `psetq` (page 38), and `setf` (page SETF-FUN). 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 that

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.

5.8. Non-local Exits

COMMON LISP provides a facility for exiting from a complex process in a non-local 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 `prog` (page 55) and `return` (page 57), which allow for so exiting a `prog` form from any point lexically within the body of the `prog`, 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.

must? MUST!

A catch may have a *tag* (a symbol) associated with it to name it, in which case it will catch only throws with a matching tag, and be invisible to all other throws.

The catch/throw facility is the basis on which the error handling machinery is built (see ???).

NO! I DISAGREE WITH THIS PHILOSOPHY. IT'S TOO COMPLEX FOR ME TO EXPOUND UPON HERE, SORRY.

5.8.1. Catch Forms

`catch` *tag* `&rest` *forms* [*Special form*]
`*catch` *tag* `&rest` *forms* [*Special form*]

The `catch` special form is the simplest catcher. The *tag* must be a symbol or `()`. The *forms* are evaluated as an implicit `progn`, 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.

Why must the tag be a symbol?

I'D PREFER TO RESTRICT IT TO SYMBOLS IF THERE'S NO REASON NOT TO.

superfluous paragraph

The *tag* is used to match up throws with catches (using `eq`). `(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.

Flush this crockishness — this is historical accident & the Lisp machine

The values `t` and `()` for *tag* are special and mean that all throws are to be caught; the value `t` is used by `unwind-protect`, for example. The only difference between `t` and `()` is in the error checking; `t` implies that after a "cleanup handler" is executed control will be thrown again to the same tag, and therefore it is an error if a specific catch for this tag does not exist higher up in the stack. Some implementations may wish to check for this error before beginning the throwing process. With a tag of `()` the error check need not be performed.

`*catch` differs from `catch` in that it evaluates *tag* as a form, whose value should be a symbol; for *catch* the tag is written explicitly and is not evaluated. This is the only difference between `catch` and `*catch`.

Seems like a pointless duplication of functions.

I AGREE.

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 `()`.

This is historical accident + irrelevant (the 3rd and 4th values are no longer documented)

Lisp Machine Lisp defines `*catch` to return four values. This seems complicated and not terribly useful. The few specialized uses of this feature can be achieved with `catchall`. Here we simply define `catch` to be consistent with the standard convention on the interaction of multiple values with implicit `progn` forms, and with a throw "tail-recursing" out of the matching catch, by analogy with `return` and `prog`.

`catchall` *catch-function* `&rest` *forms* [*Special form*]
`unwindall` *catch-function* `&rest` *forms* [*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*. `catchall` 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 `catchall`. 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).

Evidently the result of the catch-function is to be returned as the result of the catchall. Say so.

The `catchall` 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 `catchall`. This is useful because the *catch-function* can examine the tag, and if it is not of interest can relay the throw by using ~~`*throw*` (page *THROW*-FUN):~~

```
(catchall #'(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))
```

`unwindall` is just like `catchall` 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 `()`, to indicate that no throw occurred, and the other arguments are the results from the last of the *forms*. Often `unwind-protect` is more suitable for a given task than `unwindall`, however.

`unwind-protect` *protected-form* `&rest` *cleanup-forms* [*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))
```

water ??

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*.

> ??? Query: The Lisp Machine LISP manual regards it as a bug that Lisp Machine LISP doesn't handle multiple values from `unwind-protect`. I agree. So if we can do it for `unwind-protect`, why not for `prog1`?

Obsolete— That was fixed some time ago

5.8.2. Throw Forms

Flush this nil, t garbage

|| `throw` *tag result* [*Special form*]

`*throw` *tag result* [*Special form*]

The `throw` special form is the simplest thrower. The *tag* must be a symbol, and may not be `t`. The most recent outstanding catch whose tag matches *tag* or is `()` or `t` is exited. 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 executed before the unwinding process commences, and whatever results it produces are returned from the catch (or given to the *catch-function*, if appropriate).

> Compatibility note: Here there is a requirement that `throw` deliver multiple values from the *result* form; this is not compatible with present MACLISP and Lisp Machine LISP usage. The model intended here is that `throw` "tail-recurses" out of the `catch`, by analogy with `return` and `prog`.

`*throw` differs from `throw` in that it evaluates *tag* as a form, whose value should be a symbol; for *throw* the tag is written explicitly and is not evaluated. This is the only difference between `throw` and `*throw`.

Gratuitous duplication of functions. I mark use

This makes throw not a function and introduces a possible hidden efficiency cost. I want to think about this before agreeing to it. Analogy with RETURN makes some sense.

`*unwind-stack` *tag result active-frame-count action* [*Special form*]

`*unwind-stack` is a generalization of `*throw` provided for program-manipulating programs such as the error handler. Some of its actions are implementation-dependent.

All of the argument forms are evaluated; note, however, that multiple values are used from *result*. *tag* and *result* are the same as the corresponding arguments to `*throw`.

If *active-frame-count* is not `()`, it must be a non-negative integer, the number of frames to be unwound; the definition of a "frame" is implementation-dependent. If this counts down to zero before a suitable catch is found, the `*unwind-stack` operation terminates and *that frame* returns the values from *result* to whoever called it. (This is similar to Maclisp's `freturn` function.)

If *action* is non-`()`, whenever the `*unwind-stack` would be ready to terminate (either due to *active-frame-count* or due to *tag* being matched by a catch), instead *action* is called as a function, giving it the values from *result* as its arguments. It is called with the stack unwound to the specified point; if *action* returns, its results become the results of the selected frame.

Note that if both *active-frame-count* and *action* are `()`, `*unwind-stack` is identical to `*throw`.

??? Query: Perhaps this belongs not here but in a chapter on semi-compatible low-level stuff?

This does not belong in the core language.
Especially when you see what else
you need to make it usefl.

Chapter 6

FUNC

Chapter 7

MACRO

Chapter 8

Declarations

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

Declarations may be specified by either of two special forms: `declare` and `global-declare`. The `global-declare` form makes globally applicable declarations, whereas `declare` has its effects confined to a limited piece of program.

Rationale: The reason for distinguishing `declare` and `global-declare` is robustness. In MACLISP and Lisp Machine LISP, one can accidentally put a `declare` form in the wrong place, and you never find out because `declare` is a special form which doesn't do much of anything, and so the declaration is evaluated and discarded. Here it is proposed that `declare` be a special form that signals an error "misplaced declaration", but which is snarfed by the surrounding special form when appropriate. All such special forms are implicit `progn` or implicit `prog` situations, and so it isn't difficult to have a centralized handler in the interpreter.

On the other hand, given this specification, `local-declare` is not needed; one need only use `declare` within a `progn`. This strengthens the analogy between `progn` and the begin-end constructs of algebraic languages.

`global-declare &rest` *declaration-list* [*Special form*]

This is a fairly poor name

The declarations in *declaration-list* are put into effect globally, and henceforth are in force. This form should not occur anywhere but at "top level". The compiler will issue a warning if a global declaration is found elsewhere. It is a good idea in a file of code to state all global declarations before other parts of the program.

For example:

```
(global-declare
  (:special *offset*)          ; Declare a special variable.
  (:inline calibrate))         ; Always open-code the calibrate function.
```

Note that it is usually unnecessary to make explicit `:special` declarations if one uses `defvar` (page 21) or `defconst` (page 22) to declare global special variables.

`declare &rest` *declaration-list* [*Function*]

No

This form may occur only at the beginning of the bodies of (implicit or explicit) `progn` or `prog` forms; that is, a `declare` form 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.

The declarations in *declaration-list* apply to all of the code in the body of the `progn` or `prog` form. Moreover, if the construct binds variables, then any declarations in *declaration-list* which affect variable bindings will apply to those bindings; however, they will *not* apply to any executable code in the binding part of the construct.

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.

8.2. Declaration Keywords

??? Query: It seems to be that declaration types should be keywords. The old MacLisp crock of just evaluating declaration forms is not *necessary* now that `eval-when` exists, and it may not be *desirable* because it makes it harder to deal with arbitrary implementation-dependent declarations. On the other hand, all those colons are pretty ugly. What do people think?

If you don't want the colons have declaration names be looked up by string-equal.

Here is a list of valid declaration forms for use in `global-declare` and `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. You need an unspecial declaration
`:type`	(`:type` *type var1 var2* ...) declares that the specified variables will take on values only of the specified type. The :type should probably be elidable when the type name does not conflict.
`:ftype`	(`:ftype` *type function1 function2* ...) declares that the specified functions will be of the functional type *type*. DID YOU EVER EXPLAIN "FUNCTIONAL TYPES" IN GENERAL? For example:

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

`: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.

`:notinline` (`notinline` *function1 function2* `...`) declares that it is *undesirable* to compile the specified functions in-line. Remember that a compiler is free to ignore this declaration.

Implementation note: For this, and other declarations, each compiler should have a mode in which it will provide warnings of declarations it intends to ignore. This should probably be the default mode? Yes.

Note that implementations will add their own declarations keywords in most cases.

So how am I supposed to do:

```
(local-declare ((special ---))
(defun fn-1 ---)
(defun fn-2 ---))
```

if there is no local-declare?

Is there some awful variant of progn 'compile?

Chapter 9

Symbols

A LISP symbol is a data object which has three user-visible components:

- The *property list* is a list which effectively provides each symbol with many modifiable named components.

- The *print name* must be a string, which is the sequence of characters used to identify the symbol. Symbols are of great use because a symbol can be located given its name (typed, say, on a keyboard). It is ordinarily not permitted to alter a symbol's print name.

- The *package cell* must refer to a package object. A package is a data structure used to locate a symbol given its name. A symbol is uniquely identified by its name only when considered relative to a package. A symbol may be in many packages, but it can be *owned* by at most one package. The package cell points to the owner, if any.

A symbol may actually have other components as well for use by the implementation. One of the more important uses of symbols is as names for program variables; it is frequently desirable for the implementor to use certain components of a symbol to implement the semantics of variables. However, there are several possible implementation strategies, and so such possible components are not described here.

The three components named above and the functions related to them are described more individually and in more detail in the following sections.

9.1. The Property List

Since its inception, LISP has associated with each symbol a kind of tabular data structure called a *property list* (*plist* for short). A property list contains zero or more entries; each entry associates from a keyword symbol (called the *indicator*) to a Lisp object (called the *value* or, sometimes, the *property*). There are no duplications among the indicators; a property-list may only have one property at a time with a given name. In this way, given a symbol and an indicator (another symbol), an associated value can be retrieved.

A property list is very similar in purpose to an association list. The difference is that a property list is an object with a unique identity; the operations for adding and removing property-list entries are destructive operations which alter the property-list rather than making a new one. Association lists, on the other hand, are normally augmented non-destructively (without side effects), by adding new entries to the front (see `acons` (page 142) and `pairlis` (page 142)).

A property list is implemented as a memory cell (the property list cell) in a symbol containing a list with an even number (possibly zero) of elements. Each pair of elements constitutes an entry; the first item is the indicator and the second is the value. Because property-list functions are given the symbol and not the list itself, modifications to the property list can be recorded by storing back into the property-list cell of the symbol.

When a symbol is created, its property list is initially empty. Properties are created by `putprop` (page 77) and related functions.

Common Lisp does not use a symbol's property list as extensively as earlier Lisp implementations did. Less-used data, such as compiler, debugging, and documentation information, is kept on property lists in Common Lisp.

> **Compatibility note:** In older Lisp implementations, the print name, value, and function definition of a symbol were kept on its property list. The value cell was introduced into MacLisp and InterLisp to speed up access to variables; similarly for the print-name cell and function cell (MacLisp does not use a function cell). Recent Lisp implementations such as Spice Lisp, Lisp Machine Lisp, and Nil have introduced all of these cells plus the package cell. None of the MacLisp system property names (`expr`, `fexpr`, `macro`, `array`, `subr`, `lsubr`, `fsubr`, and in former times `value` and `pname`) exist in Common Lisp.

> **Compatibility note:** In Common Lisp, the notion of "disembodies property list" introduced in MacLisp is eliminated. It tended to be used for rather kludgy things, and in Lisp Machine Lisp is often associated with the use of locatives (to make it "off by one" for searching alternating keyword lists). In Common Lisp special `setf`-like property list functions are introduced: `getf` (page GETF-FUN), `putpropf` (page PUTPROPF-FUN), and `rempropf` (page REMPROPF-FUN).

`get` *symbol* *indicator* `&optional` *default* [*Function*]

Font

get searches the property list of *symbol* for an indicator `eq` to *indicator*. If one is found, then the corresponding value is returned; otherwise *default* is returned. If *default* is not specified, then `()` is used for *default*. Note that there is no way to distinguish an absent property from one whose value is *default*.

Suppose that the property list of `foo` is `(bar t baz 3 hunoz "Huh?")`. Then, for example:

```
(get 'foo 'baz) => 3
(get 'foo 'hunoz) => "Huh?"
(get 'foo 'zoo) => ()
                    ; a isn't even part of the (disembodied) property list
```

— ??

`getl` *symbol* *indicator-list* [*Function*]

`getl` is like `get`, except that the second argument is a list of indicators. `getl` searches the property list of *symbol* for any of the indicators in *indicator-list*, until it finds a property whose indicator is one of the elements of *indicator-list*.

`getl` returns that tail of the property list which begins with the first such property found. So the *car* of the returned list is an indicator, and the *cadr* is the property value. If none of the indicators on *indicator-list* are on the property list, `getl` returns `()`.

For example:

If the property list of `foo` were

```
   (bar (1 2 3) baz (3 2 1) color blue height six-two)
```

then

```
   (getl 'foo '(baz height))
      => (baz (3 2 1) color blue height six-two)
```

When more than one of the indicators in *indicator-list* is present in *symbol*, which one `getl` returns depends on the order of the properties. `getl` is the only function that depends on that order. The order in which properties appear on a property list is implementation-dependent. [Programs should avoid examining the *cddr* of a result returned by `getl`.] Non-sequitur -- motivate.

`putprop` *symbol value indicator* [*Function*]

This causes *symbol* to have a property whose indicator is *indicator* and whose value is *value*. If the property list already already has had a property with an indicator `eq` to *indicator*, then the value previously ~~associated with~~ that indicator is removed from the property list and replaced by *value*. The property list is destructively altered by using side effects. After a `putprop` is done, `(get` *symbol indicator*`)` will return *value*. `putprop` returns the new *value*.

For example:

```
(putprop 'Nixon 'not 'crook) => not
(get 'Nixon 'crook) => not
```

`defprop` *symbol value indicator* [*Special form*]

`defprop` is a form of `putprop` with unevaluated arguments, which is sometimes more convenient for typing.

For example:

```
(defprop foo bar next-to) <=> (putprop 'foo 'bar 'next-to)
```

Often it is convenient to represent a data base by using property lists, and to initialize it by evaluating a file of `defprop` forms.

For example:

```
(defprop and 0 pdp-8-opcode)
(defprop tad 1 pdp-8-opcode)
(defprop dca 2 pdp-8-opcode)
(defprop isz 3 pdp-8-opcode)
(defprop jms 4 pdp-8-opcode)
(defprop jmp 5 pdp-8-opcode)
(defprop iot 6 pdp-8-opcode)
```

Normally it doesn't make sense to use a disembodied property list rather than a symbol as the *symbol* argument. Especially in Lisp's that don't have them!

Also defprop returns its first argument, whereas putprop returns its second

ENGLISH SYNTAX

`remprop` *symbol indicator* [*Function*]

This removes from *symbol* the property with an indicator `eq` to *indicator*, by splicing it out of the property list. It returns that portion of the property list of which value of the former *indicator*-property was the `car`. `car` of what `remprop` returns is what `get` would have returned with the same arguments.

? FONT?

For example:

If the property list of `foo` was
  `(color blue height 6.3 near-to bar)`
then
  `(remprop 'foo 'height) => (6.3 near-to bar)`
and `foo`'s property list would have been altered to be
  `(color blue near-to bar)`

If *symbol* has no *indicator*-property, then `remprop` has no side-effect and returns `()`.

`plist` *symbol* [*Function*]

This returns the list which contains the property pairs of *symbol*. For a disembodied property list, this simply performs a `cdr` operation; for a symbol, the contents of the property list cell are extracted and returned.

Note that using `get` on the result of `plist` does *not* work. One must give the symbol itself to `get`.

9.2. The Print Name

Every symbol has an associated string called the *print-name*, or *pname* for short. This string is used as the external representation of the symbol: if the characters in the string are typed in to `read` (with suitable escape conventions for certain characters), it is interpreted as a reference to that symbol (if it is interned); and if the symbol is printed, `print` types out the print-name. For more information, see the section on the *reader* (see page READER) and *printer* (see page PRINTER).

`get-pname` *sym* [*Function*]

This returns the print-name of the symbol *sym*.

For example:

```
(get-pname 'XYZ) => "XYZ"
```

It is an extremely bad idea to modify a string being used as the print name of a symbol. Such a modification may confuse the function `read` (page 211) and the package system tremendously.

`samepnamep` *sym1 sym2* [*Function*]

This predicate returns `t` if the two symbols *sym1* and *sym2* have `equal` print-names; that is, if their printed representation is the same. Upper and lower case letters are considered to be different. ||

Compatibility note: In Lisp Machine Lisp, `samepnamep` normally considers upper and lower case to be the same. However, in MacLisp, which originated this function, the cases are distinguished: Lisp Machine Lisp introduced the incompatibility. COMMON LISP is compatible with MACLISP here.

If either or both of the arguments is a string instead of a symbol, then that string is used in place of the print-name. `samepnamep` is useful for determining if two symbols would be the same except that they are not in the same package.

For example:

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

9.3. Creating Symbols

Symbols can be used in two rather different ways. An *interned* symbol is one which is indexed by its print-name in a catalog called a *package*. Every time anyone asks for a symbol with that print-name, he gets the same (`eq`) symbol. Every time input is read with the function `read` (page 211), and that print-name appears, it is read as the same symbol. This property of symbols makes them appropriate to use as names for things and as hooks on which to hang permanent data objects (using the property list, for example; it is no accident that symbols are both the only LISP objects which are cataloged and the only LISP objects which have property lists).

Interned symbols are normally created automatically; the first time someone (such as the function `read`) asks the package system for a symbol with a given print-name, that symbol is automatically created. The function to use to ask for an interned symbol is `intern` (page INTERN-FUN), or one of the functions related to `intern`.

Although interned symbols are the most commonly used, they will not be discussed further here. For more information, turn to the chapter on packages.

An *uninterned* symbol is a symbol used simply as a data object, with no special cataloging (it belongs to no particular package). An uninterned symbol prints in the same way as an interned symbol with the same print-name, but cannot be read back in. The following are some functions for creating uninterned symbols.

`make-symbol` *pname* [*Function*]

(`make-symbol` *pname*) creates a new uninterned symbol, whose print-name is the string *pname*. The value and function bindings will be unbound and the property list will be empty.

Compatibility note: Lisp Machine LISP uses the second argument for an odd flag related to areas. It is unclear what NIL does about this.

`copysymbol` *sym* `&optional` *copy-props* [*Function*]

This returns a new uninterned symbol with the same print-name as *sym*. If *copy-props* is non-`()`, then the initial value and function-definition of the new symbol will be the same as those of *sym*, and the property list of the new symbol will be a copy of *sym*'s. If *copy-props* is `()` (the default), then the new symbol will be unbound and undefined, and its property list will be empty.

`gensym &optional` *x* [*Function*]

`gensym` invents a print-name, and creates a new symbol with that print-name. It returns the new, uninterned symbol.

The invented print-name consists of a prefix character (the value of `si:*gensym-prefix` (page SI:*GENSYM-PREFIX-VAR), initially `#\G`) followed by the low four digits of the decimal representation of a number (the value of `si:*gensym-counter` (page SI:*GENSYM-COUNTER-VAR)). The number is increased by one every time `gensym` is called.

You really want: Document thes names?

If the argument *x* is present and is a fixnum, then *x* must be non-negative, and `si:*gensym-counter` is set to *x*. If *x* is a string or a symbol, then `si:*gensym-prefix` is set to first character of the string or of the symbol's print-name. After handling the argument, `gensym` creates a symbol as it would with no argument.

the

For example:

```
(gensym) => G0007
(gensym 'FOO) => F0008
(gensym 32) => F0032
(gensym) => F0033
(gensym "GARBAGE") => G0034
```

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

`gensym` is usually used to create a symbol which should not normally be seen by the user, and whose print-name is unimportant, except to allow easy distinction by eye between two such symbols. The optional argument is rarely supplied. The name comes from "generate symbol", and the symbols produced by it are often called "gensyms".

If it is crucial that no two generated symbols have the same print name (rather than merely being distinct data structures), or if it is desirable for the generated symbols to be interned, then the function `gentemp` (page GENTEMP-FUN) may be more appropriate to use.

`get-package` *sym* [*Function*]

Given a symbol *sym*, `get-package` returns the contents of the package cell of that symbol.

should be called symbol-package?

That's what we call it.

(we'll probably be changing this name, though)

Chapter 10

Numbers

COMMON LISP provides several different representations for numbers. These representations may be divided into two categories: integers and floating-point numbers. Most numeric functions will accept any kind of number; they are *generic*. Those functions which accept only certain special numbers are so described below.

In general, numbers in COMMON LISP are not true objects; `eq` cannot be counted upon to operate on them reliably. In particular, it is possible that the expression

```
(let ((x z) (y z)) (eq x y))
```

may return `()` rather than `t`, if the value of `z` is a number.

> Rationale: This odd breakdown of `eq` in the case of numbers allows the implementor enough design freedom to produce exceptionally efficient numerical code on conventional architectures. MACLISP requires this freedom, for example, in order to produce compiled numerical code equal in speed to FORTRAN. If not for this freedom, then at least for the sake of compatibility, COMMON LISP makes this same restriction.

If two objects are to be compared for "identity", but either might be a number, then the predicate `eql` (page 30) is probably appropriate; if both objects are known to be numbers, then `=` (page 82) may be preferable.

As a rule, computations with floating-point numbers are only approximate. The *precision* of a floating-point number is not necessarily correlated at all with the *accuracy* of that number. The precision refers to the number of bits retained in the representation. When an operation combines a short floating-point number with a long one, the result will be a long floating-point number. This rule is made to ensure that as much accuracy as possible is preserved; however, it is by no means a guarantee. COMMON LISP numerical routines do assume, however, that the accuracy of an argument does not exceed its precision. Therefore when two short floating-point numbers are combined, the result will be a short floating-point number. This assumption can be altered by first explicitly converting a short floating-point number to long representation. (COMMON LISP never converts automatically from long size to short in an effort to save space.)

Integer computations cannot overflow in the usual sense (though of course there may not be enough storage to represent one), as integers may in principle be of any magnitude. Floating-point computations may get exponent overflow or underflow, in which case an error is signalled.

10.1. Predicates on Numbers

`zerop` *number* [*Function*]

True (returning `t`) if *number* is zero (either the integer zero or a floating-point zero); otherwise `()` is returned. If the argument *number* is not a number, `zerop` signals an error.

`plusp` *number* [*Function*]

True (returning `t`) if *number* is strictly greater than zero; otherwise `()` is returned. If the argument *number* is not a number, `plusp` signals an error.

`minusp` *number* [*Function*]

True (returning `t`) if *number* is strictly less than zero; otherwise `()` is returned. If the argument *number* is not a number, `plusp` signals an error.

what happens on complex?

`oddp` *integer* [*Function*]

Returns `t` if the argument *integer* is odd (not divisible by two), and otherwise returns `()`. It is an error if the argument is not an integer.

`evenp` *integer* [*Function*]

Returns `t` if the argument *integer* is even (divisible by two), and otherwise returns `()`. It is an error if the argument is not an integer.

comma

See also the data-type predicates `integerp` (page 27), `fixnump` (page 28), `bigp` (page 27), `bitp` (page 29), `ratiop` (page 28), `rationalp` (page 28) `floatp` (page 28), `short-float` (page 28), `single-floatp` (page 28), `double-floatp` (page 28), `long-floatp` (page 28), `scalarp` (page SCALARP-FUN), `complexp` (page COMPLEXP-FUN), and `numberp` (page 27).

10.2. Comparisons on Numbers

All of the functions in this section require that their arguments be numbers, and signal an error if given a non-number. They work on all types of numbers, automatically performing any required coercions.

RWG REALLY WANTS = TO WORK ON &REST MANY-NUMBERS.
MAYBE THAT WOULD BE MORE USEFUL. YOU COULD HAVE
A SEPARATE FUNCTION CALLED FUZZY-= ?

`=` *number1* *number2* `&optional` *fuzz* [*Function*]

Returns `t` if *number1* and *number2* are numerically equal. This is used by `equalp` (page 32) when both its arguments are numbers. The optional argument *fuzz* allows nearly-equal floating-point numbers to be considered equal: two numbers *x* and *y* 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, if $abs(x-y) \le fuzz*max(abs(x), abs(y))$. If no third argument is supplied, then *fuzz* defaults to `0.0`, and in this case *x* and *y* must be exactly equal for `=` to return `t`.

Compatibility note: In COMMON LISP, `=` performs "mixed-mode" comparisons. In MACLISP the arguments must be either both fixnums or both floating-point numbers, and moreover there is no *fuzz* argument.

complex?

`<` *number* `&rest` *more-numbers* [*Function*]
`>` *number* `&rest` *more-numbers* [*Function*]
`<=` *number* `&rest` *more-numbers* [*Function*]
`>=` *number* `&rest` *more-numbers* [*Function*]

These functions each take one or more arguments. If the sequence of arguments satisfies a certain condition:

`<`	monotonically increasing
`>`	monotonically decreasing
`<=`	monotonically nondecreasing
`>=`	monotonically nonincreasing

then the result is `t`, and otherwise `()`.

For example:

```
(< 3 5) => t
(< 3 -5) => ()
(< 3 3) => ()
(<= 3 3) => t
(< 0 3 4 6 7) => t
(< 0 3 4 4 6) => ()
(<= 0 3 4 4 6) => t
(> 4 3) => t
(> 4 3 2 1 0) => t
(> 4 3 1 2 0) => ()
```

With two arguments, these functions perform the usual arithmetic comparison tests. With three arguments, they are useful for range checks.

For example:

```
(<= 0 x 9)          ;true iff x is between 0 and 9
(< 0.0 x 1.0)       ;true iff x is between 0.0 and 1.0, exclusive
(< -1 j (string-length s))       ;true iff j is a valid index for string s
```

Compatibility note: In COMMON LISP, the comparison operations perform "mixed-mode" comparisons. In MACLISP, the arguments must be either both fixnums or both floating-point numbers.

Complex?

`max` *number* `&rest` *more-numbers* [*Function*]

`max` returns the argument which is greatest (closest to positive infinity).

For example:

```
(max 1 3 2 -7) => 3
(max -2 3 0 7) => 7
(max 3) => 3
(max 3.0 7 1) => 7 or 7.0
```

If the arguments are a mixture of integers and floating-point numbers, and the largest is an integer, then the implementation is free to produce either that integer or its floating-point equivalent.

rational

omplex?

`min` *number* `&rest` *more-numbers* [*Function*]

`min` returns the argument which is least (closest to negative infinity).

For example:

```
(max 1 3 2 -7) => -7
(max -2 3 0 7) => -2
(min 3) => 3
(min 3.0 7 1) => 1 or 1.0
```

If the arguments are a mixture of integers and floating-point numbers, and the smallest is an integer, then the implementation is free to produce either that integer or its floating-point equivalent.

rational

10.3. Arithmetic Operations

All of the functions in this section require that their arguments be numbers, and signal an error if given a non-number. They work on all types of numbers, automatically performing any required coercions.

`+` `&rest` *numbers* [*Function*]

Returns the sum of the arguments. If there are no arguments, the result is 0, which is an identity for this operation.

Compatibility note: While + is compatible with its use in Lisp Machine LISP, it is incompatible with MACLISP, which uses + for fixnum-only addition.

`-` *number* `&rest` *more-numbers* [*Function*]

The function `-`, when given one argument, returns the negative of that argument.

The function `-`, when given more than one argument, subtracts from the first argument all the others, and returns the result.

Compatibility note: While - is compatible with its use in Lisp Machine LISP, it is incompatible with MACLISP, which uses - for fixnum-only subtraction. Also, - differs from `difference` as used in most LISP systems in the case of one argument.

`abs` *number* [*Function*]

Returns the absolute value of the argument.

```
(abs x) <=> (if (minusp x) (- x) x)
```

What happens on complex?

`*` `&rest` *numbers* [*Function*]

Returns the product of the arguments. If there are no arguments, the result is 1, which is an identity for this operation.

Compatibility note: While * is compatible with its use in Lisp Machine LISP, it is incompatible with MACLISP, which uses * for fixnum-only multiplication.

`/` *number* `&rest` *more-numbers* [*Function*]

The function `/`, when given more than one argument, divides the first argument by all the others, and returns the result.

With one argument, `/` reciprocates the argument.

`/` strives always to produce something near the mathematically correct result. `/` will produce a rational or floating-point number if the mathematical quotient of two integers is not an exact integer.

Should never produce a flonum for quotient of two integers — that way lies many numerical pitfalls. Make rationals a required language feature (they are at least as easy as bignums)

For example:

```
(/ 12 4) => 3
(/ 13 4) => 3.25
(/ -8) => -0.125
```

No! 13/4
-1/8

To divide one integer by another producing an integer result, use one of the functions `floor`, `ceil`, `trunc`, or `round` (page 89).

YEAH!

Compatibility note: What `/` does is totally unlike what the usual `//` or `quotient` operator does. In most LISP systems, `quotient` behaves like `/` except when dividing integers, in which case it behaves like `trunc` (page 89) of two arguments; this behavior is mathematically intractable, and in practice `quotient` is used only when one is sure that both argument are integers, *or* when one is sure that at least one argument is a floating-point number. `/` is tractable for its purpose, and "works" for *any* numbers. For "integer division", `trunc` (page 89) and its relatives are available in COMMON LISP.

Why are these here if plus, etc. are flushed?

`1+` *number* [*Function*]
`add1` *number* [*Function*]

`(1+ x)` is the same as `(+ x 1)`. `add1` does the same thing.

`1-` *number* [*Function*]
`sub1` *number* [*Function*]

`(1- x)` is the same as `(- x 1)`. `sub1` does the same thing. Note that the short name may be confusing: `(1- x)` does *not* mean $1-x$; rather, it means $x-1$.

`gcd` `&rest` *integers* [*Function*]

Returns the greatest common divisor of all the arguments, which must be integers. The result is always a non-negative integer. If no arguments are given, `gcd` returns 0, which is an identity for this operation.

10.4. Irrational and Transcendental Functions

Except as noted, these functions accept any kind of argument, but always return a floating-point result. If the argument is floating, the result will be of the same precision. (For multi-arg case, what?)

`exp` *number* [*Function*]

Returns *e* raised to the power `number`, where *e* is the base of the natural logarithms.

font

`expt` *base-number power-number* [*Function*]

Returns *base-number* raised to the power *power-number*. If both arguments are integers and *power-number* is non-negative, the result will be an integer; otherwise a floating-point number may result.

will?

Implementation note: If the exponent is an integer a repeated-squaring algorithm may be used, while if the exponent is a floating-point number the result may be calculated as:

`(exp (* power-number (log base-number)))`

or in any other reasonable manner.

Does expt of integer to a negative integer return a ratio? Does expt of a ratio to a positive integer return a ratio? ... or a flonum.

`ln` *scalar* [*Function*]

Returns the natural logarithm of *scalar*. The argument must be strictly positive.

`log` *base scalar* [*Function*]

Returns the logarithm of *scalar* in the base *base*. Both arguments must be strictly positive scalars.

For example:

```
(log 2 8.0) => 3.0
(log 10 0.01) => -2.0
```

??? Query: Most LISP implementations, as well as other programming languages (such as FORTRAN), call the natural-logarithm function `log`. Mathematicians usually call this `ln`, however. It would be useful to have a two-argument logarithm function. One could let `log` serve for both the one-argument and two-argument versions, but then `&optional` arguments could not be used in the obvious way if one puts the arguments in the order normally used in mathematical notation, because it would be the first argument which is optional. Opinions?

`sqrt` *scalar* [*Function*]

Returns the positive square root of *scalar*, which must be non-negative.

`isqrt` *integer* [*Function*]

Integer square-root: the argument must be a non-negative integer, and the result is the greatest integer less than or equal to the exact positive square root of the argument.

`sin` *radians* [*Function*]
`sind` *degrees* [*Function*]

Returns the sine of the argument. `sin` assumes its argument to be in radians; `sind` assumes it to be in degrees.

`cos` *radians* [*Function*]
`cosd` *degrees* [*Function*]

Returns the cosine of the argument. `cos` assumes its argument to be in radians; `cosd` assumes it to be in degrees.

The Lisp machine introduced these degree functions, but I think they probably ought to be flushed. NOT CLEAR.

`atan` *y* `&optional` *x* [*Function*]

`atand` *y* `&optional` *x* [*Function*]

An arctangent is calculated and the result is returned in radians (`atan`) or degrees (`atand`).

With two arguments *y* and *x*, the result is the arctangent of the quantity y/x. The signs of *y* and *x* are used to derive quadrant information; moreover, *x* may be zero provided *y* is not zero. The value of `atan` is always between $-\pi$ (exclusive) and π (inclusive). The following table details various special cases.

(but see note below)

Condition		Cartesian locus	Range of result
$y = 0$	$x > 0$	Positive *x*-axis	0
$y > 0$	$x > 0$	Quadrant I	$0 < \text{result} < \pi/2$
$y > 0$	$x = 0$	Positive *y*-axis	$\pi/2$
$y > 0$	$x < 0$	Quadrant II	$\pi/2 < \text{result} < \pi$
$y = 0$	$x < 0$	Negative *x*-axis	π
$y < 0$	$x < 0$	Quadrant III	$-\pi < \text{result} < -\pi/2$
$y < 0$	$x = 0$	Negative *y*-axis	$-\pi/2$
$y < 0$	$x > 0$	Quadrant IV	$-\pi/2 < \text{result} < 0$
$y = 0$	$x = 0$	Origin	error

Actually, the $<$ signs in the above table ought to be $\le$ signs, because of rounding effects; if *y* is greater than zero but nevertheless very small, then the floating-point approximation to $\pi/2$ might be a more accurate result than any other floating-point number. (For that matter, when $y = 0$ the exact value $\pi/2$ cannot be produced anyway, but instead only an approximation.)

With only one argument *y*, the result is the arctangent of *y*, and lies between $-\pi/2$ and $\pi/2$ (both exclusive). I.e. x defaults to 1.

Compatibility note: MacLisp has a function called `atan` which ranges from 0 to 2π. Every other language in the world (ANSI FORTRAN, IBM PL/I, InterLISP) has an arctangent function with range $-\pi$ to π. Lisp Machine Lisp provides two functions, `atan` (compatible with MacLisp) and *atan2* (compatible with everyone else). font

Common Lisp makes `atan` the standard one with range $-\pi$ to π. Observe that this makes the one-argument and two-argument versions of `atan` compatible in the sense that the branch cuts do not fall in different places, which is probably why most languages use this definition. (An aside: the InterLisp one-argument function `arctan` has a range from 0 to π, while every other language in the world provides the range $-\pi/2$ to $\pi/2$! Nevertheless, since InterLisp uses the standard two-argument version, its branch cuts are inconsistent anyway.)

`pi` [*Variable*]

`short-pi` [*Variable*]

`single-pi` [*Variable*]

`double-pi` [*Variable*]

`long-pi` [*Variable*]

These five global variables have as their initial values floating-point approximations to π. `short-pi` contains the best possible short-format approximation, and similarly for the other three formats: single, double, and long. `pi` contains the same value as `long-pi`.

Is short-pi ≠ (short-float pi) ?
I would assume that if the rounding is done
properly, these would be =, in which case
the other four variable names are superfluous

10.5. Type Conversions on Numbers

While most arithmetic functions will operate on any kind of number, coercing types if necessary, the following functions are provided to allow specific conversions of data types to be forced, when desired.

`float` *scalar* [*Function*]

Converts any kind of scalar to a floating-point number. If a given format of floating-point number is sufficiently precise to represent the result, then the result may be of that format or of any larger format, depending on the implementation. If no fixed format is sufficiently precise, then long format is used. To force a particular size of floating-point number to be produced, use one of the more specific float functions below.

float should take an optional second argument. If present, the first arg is floated to the same precision as the second. Among other things, this is used for functions which are supposed to return results of the same precision as their argument (but do the intermediate computations in extended precision.)

`short-float` *scalar* [*Function*]

Converts any kind of scalar to a short floating-point number.

`single-float` *scalar* [*Function*]

Converts any kind of scalar to a single floating-point number.

`double-float` *scalar* [*Function*]

Converts any kind of scalar to a double floating-point number.

`long-float` *scalar* [*Function*]

Converts any kind of scalar to a long floating-point number.

IS THIS USEFUL FOR ANYTHING, VI
VIS RATIONALI

`rational` *scalar* [*Function*]
`rationalize` *scalar* [*Function*]

Each of these functions converts any kind of scalar to be a rational number. If the argument is already rational, that argument is returned. The two functions differ in their treatment of floating-point numbers. `rational` assumes that the floating-point number is completely accurate, and returns a rational number mathematically equal to the precise value of the floating-point number. `rationalize` assumes that the floating-point number is accurate only to the precision of the floating-point representation, and may return any rational number for which the floating-point number is the best available approximation; in doing this it attempts to keep both numerator and denominator small.

Should rationalize take an optional second argument of the number of bits of precision? (If negati
the num
of least
signifi
bits to
discar
It would
often be
with a
second
of -1

There is no `fix` function in COMMON LISP, because there are several interesting ways to convert non-integral values to integers. These are provided by the functions below, which perform not only type-conversion but also some non-trivial calculations.

Explicate this with an example, e.g. one where rationalize returns 1/3 while rational returns something gross.

`floor` *scalar* `&optional` *divisor* [*Function*]
`ceil` *scalar* `&optional` *divisor* [*Function*]
`trunc` *scalar* `&optional` *divisor* [*Function*]
`round` *scalar* `&optional` *divisor* [*Function*]

In the simple, one-argument case, each of these functions converts its argument *scalar* to be an integer. If the argument is already an integer, it is returned directly. If the argument is a ratio or floating-point number, the functions use different algorithms for the conversion.

`floor` converts its argument by truncating towards negative infinity; that is, the result is the largest integer which is not larger than the argument.

`ceil` converts its argument by truncating towards positive infinity; that is, the result is the smallest integer which is not smaller than the argument.

`trunc` converts its argument by truncating towards zero; that is, the result is the integer of the same sign as the argument and which has the greatest integral magnitude not greater than that of the argument., or zero when no such integer exists.

`round` converts its argument by rounding to the nearest integer; if *number* is exactly halfway between two integers (that is, of the form *integer*+0.5) then it is rounded to the one which is even (divisible by two).

Here is a table showing what the four functions produce when given various arguments.

Argument	`floor`	`ceiling`	`trunc`	`round`
2.6	2	3	2	3
2.5	2	3	2	2
2.4	2	3	2	2
0.7	0	1	0	1
0.3	0	1	0	0
-0.3	-1	0	0	0
-0.7	-1	0	0	-1
-2.4	-3	-2	-2	-2
-2.5	-3	-2	-2	-2
-2.6	-3	-2	-2	-3

If a second argument *divisor* is supplied, then the result is the appropriate type of rounding or truncation applied to the result of dividing the *number* by the *divisor*. For example, `(floor 5 2) = (floor (/ 5 2))`, but is potentially more efficient. The *divisor* may be any kind of scalar. The one-argument case is exactly like the two-argument case where the second argument is `1`.

Each of the functions actually returns *two* values; the second result is the remainder, and may be obtained using `multiple-value-let` (page 59) and related constructs. If any of these functions is given two arguments x and y and produces results q and r, then $q*y+r=x$. The remainder r is an integer if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. (In the one-argument case the remainder is a number of the same type as the argument.) The first result is always an integer.

Compatibility note: The names of the functions `floor`, `ceil`, `trunc`, and `round` are more accurate than names like `fix` which have heretofore been used in various LISP systems. The names used here are compatible not exactly.

with standard mathematical terminology (and with PL/I, as it happens). In FORTRAN `ifix` means `trunc`. ALGOL 68 provides `round`, and uses `entier` to mean `floor`. In MACLISP, `fix` and `ifix` both mean `floor` (one is generic, the other flonum-in/fixnum-out). In INTERLISP, `fix` means `trunc`. In Lisp Machine LISP, `fix` means `floor` and `fixr` means `round`. STANDARD LISP provides a `fix` function, but does not accurately specify what it does exactly. The existing usage of the name `fix` is so confused that it seems best to avoid it altogether.

The names and definitions given here have recently been adopted by Lisp Machine LISP, and MACLISP and NIL seem likely to follow suit.

`mod` *number* `&optional` *divisor* [*Function*]
`remainder` *number* `&optional` *divisor* [*Function*]

In Lisp machine proposa
there also take an optio
argument which is the
minimum number of bi
of precision accept
in the resu
An error i
signalled i
the result w
have fewer
Consider
which sto
by mod'i
its argume
with 27

I sure wish this could be called "rem"

`mod` performs the operation `floor` (page 89) on its arguments, and returns the *second* result of `floor` as its only result. Similarly, `remainder` performs the operation `trunc` (page 89) on its arguments, and returns the *second* result of `trunc` as its only result.

`mod` and `remainder` are therefore the usual modulus and remainder functions when applied to two integer arguments. In general, however, the arguments may be integers or floating-point numbers.

With one argument, these functions perform the "mod 1" or "fractional part" operation, differing in the direction of rounding: the result of `mod` of one argument is always non-negative, while the result of `remainder` of one argument always has the same sign as the argument.

```
(mod 13 4) => 1                (remainder 13 4) => 1
(mod -13 4) => 3               (remainder -13 4) => -1
(mod 13 -4) => -3              (remainder 13 -4) => 1
(mod -13 -4) => -1             (remainder -13 -4) => -1
(mod 13.4) => 0.4              (remainder 13.4) => 0.4
(mod -13.4) => 0.6             (remainder -13.4) => -0.4
```

`ffloor` *number* `&optional` *divisor* [*Function*]
`fceil` *number* `&optional` *divisor* [*Function*]
`ftrunc` *number* `&optional` *divisor* [*Function*]
`fround` *number* `&optional` *divisor* [*Function*]

These functions are just like `floor`, `ceil`, `trunc`, and `round`, except that the result (the first result of two) is always a floating-point number rather than an integer. It is roughly as if `ffloor` gave its arguments to `floor`, and then applied `float` to the first result before passing them both back. In practice, however, `ffloor` may be implemented much more efficiently. Similar remarks apply to the other three functions. If the first argument is a floating-point number, and the second agrument is not a floating-point number of shorter format, then the first result will be a floating-point number of the same type as the first argument.

For example:

```
(ffloor -4.7) => -5.0 and 0.3
(ffloor 3.5d0) => 3.0d0 and 0.5d0
```

10.6. Logical Operations on Numbers

The logical operations in this section treat integers as if they were represented in two's-complement notation.

> **Implementation note:** Internally, of course, an implementation of COMMON LISP may or may not use a two's-complement representation. All that is necessary is that the logical operations perform calculations so as to give this appearance to the user.

The logical operations provide a convenient way to represent an infinite vector of bits. Let such a conceptual vector be indexed by the non-negative integers. Then bit j is assigned a "weight" 2^j. Assume that only a finite number of bits are ones, or that only a finite number of bits are zeros. A vector with only a finite number of one-bits is represented as the sum of the weights of the one-bits, a positive integer. A vector with only a finite number of zero-bits is represented as `-1` minus the sum of the weights of the zero-bits, a negative integer.

This method of using integers to represent bit vectors can in turn be used to represent sets. Suppose that some (possibly countably infinite) universe of discourse for sets is mapped into the non-negative integers. Then a set can be represented as a bit vector; an element is in the set if the bit whose index corresponds to that element is a one-bit. In this way all finite sets can be represented (by positive integers), as well as all sets whose complements are finite (by negative integers). The functions `logior`, `logand`, and `logxor` defined below then compute the union, intersection, and symmetric difference operations on sets represented in this way.

`logior &rest` *integers* [*Function*]

Returns the bit-wise logical *inclusive or* of its arguments. If no argument is given, then the result is zero, which is an identity for this operation.

`logxor &rest` *integers* [*Function*]

Returns the bit-wise logical *exclusive or* of its arguments. If no argument is given, then the result is zero, which is an identity for this operation.

`logand &rest` *integers* [*Function*]

Returns the bit-wise logical *and* of its arguments. If no argument is given, then the result is `-1`, which is an identity for this operation.

`logeqv &rest` *integers* [*Function*]

Returns the bit-wise logical *equivalence* (also known as *exclusive nor*) of its arguments. If no argument is given, then the result is `-1`, which is an identity for this operation.

`lognand` *integer1* *integer2* [*Function*]
`lognor` *integer1* *integer2* [*Function*]
`logandc1` *integer1* *integer2* [*Function*]
`logandc2` *integer1* *integer2* [*Function*]
`logorc1` *integer1* *integer2* [*Function*]
`logorc2` *integer1* *integer2* [*Function*]

These are the other six non-trivial bit-wise logical operations on two arguments. Because they are not commutative or associative, they take exactly two arguments rather than any non-negative number of arguments.

```
 (lognand n1 n2) <=> (lognot (logand n1 n2))
  (lognor n1 n2) <=> (lognot (logor n1 n2))
(logandc1 n1 n2) <=> (logand (lognot n1) n2)
(logandc2 n1 n2) <=> (logand n1 (lognot n2))
 (logorc1 n1 n2) <=> (logor (lognot n1) n2)
 (logorc2 n1 n2) <=> (logor n1 (lognot n2))
```

The ten bit-wise logical operations on two integers are summarized in this table:

Argument 1	0	0	1	1	
Argument 2	0	1	0	1	*Operation name*
`logand`	0	0	0	1	and
`logior`	0	1	1	1	inclusive or
`logxor`	0	1	1	0	exclusive or
`logeqv`	1	0	0	1	equivalence (exclusive nor)
`lognand`	1	1	1	0	not-and
`lognor`	1	0	0	0	not-or
`logandc1`	0	1	0	0	and complement of arg1 with arg2
`logandc2`	0	0	1	0	and arg1 with complement of arg2
`logorc1`	1	1	0	1	or complement of arg1 with arg2
`logorc2`	1	0	1	1	or arg1 with complement of arg2

`lognot` *integer* [*Function*]

Returns the bit-wise logical *not* of its argument. Every bit of the result is the complement of the corresponding bit in the argument.

```
(logbitp j (lognot x)) <=> (not (logbitp j x))
```

`logtest` *integer1* *integer2* [*Function*]

`logtest` is a predicate which returns `t` if any of the bits designated by the 1's in *integer1* are 1's in *integer2*.

```
(logtest x y) <=> (not (zerop (logand x y)))
```

`logbitp` *index integer* [*Function*]

logbitp returns `t` if the bit in *integer* whose index is *index* (that is, its weight is 2^{index}) is a one-bit; otherwise it returns `()`.

For example:

```
(logbitp 2 6) => t
(logbitp 0 6) => ()
(logbitp k n) <=> (ldb-test (byte k 1) n)
```

`ash` *integer count* [*Function*]

Shifts *integer* arithmetically left by *count* bit positions if *count* is positive, or right -*count* bit positions if *count* is negative. The sign of the result is always the same as the sign of *integer*.

Arithmetically, this operation performs the computation $floor(integer*2^{count})$.

Logically, this moves all of the bits in *integer* to the left, adding zero-bits at the bottom, or moves them to the right, discarding bits. (In this context the question of what gets shifted in on the left is irrelevant; integers, viewed as strings of bits, are "half-infinite", that is, conceptually extend infinitely far to the left.)

For example:

```
(logbitp j (ash n k))
   <=> (and (>= j k) (logbitp (- j k) n))
```

`logcount` *integer* [*Function*]

This sentence is very misleading

The number of bits in *integer* is determined and returned. If *integer* is positive, then 1 bits in its binary representation are counted. If *integer* is negative, then the 0 bits in its two's-complement binary representation are counted. The result is always a non-negative integer.

For example:

```
(logcount 13) => 3          ; Binary representation is ...0001101
(logcount -13) => 2         ; Binary representation is ...1110011
(logcount 30) => 4          ; Binary representation is ...0011110
(logcount -30) => 4         ; Binary representation is ...1100010
```

As a rule,

```
(logcount x) <=> (logcount (- (+ x 1)))
```

`haulong` *integer* [*Function*]

This returns the number of significant bits in the absolute value of *integer*. The precise computation performed is $ceiling(log2(abs(integer)+1))$.

equivalent to

maybe this expression should be written in Lisp?

~~This expression is also incorrect, since 6 and 7 have the same haulong (3), but the expression~~

Ah, expression is in fact correct. But unnecessarily obscure?

For example:

```
(haulong 0) => 0
(haulong 3) => 2
(haulong 4) => 3
(haulong -7) => 3
```

`haipart` *integer count* [*Function*]

Returns the high *count* bits of the binary representation of the absolute value of *integer*, or the low -*count* bits if *count* is negative. A possible definition of `haipart`:

```
(defun haipart (integer count)
       (let ((x (abs integer)))
            (if (minusp count)
                (ldb (byte (- count) 0) x)
                (ldb (byte count (max (- (haulong x) n) 0))
                     x))))
```

Poor indentation

↑ You have the arguments to byte reversed, which bolsters my argument that size, position is a better order!

10.7. Byte Manipulation Functions

Several functions are provided for dealing with an arbitrary-width field of contiguous bits appearing anywhere in an integer. Such a contiguous set of bits is called a *byte*. Here the term *byte* does not imply some fixed number of bits (such as eight), but a field of arbitrary and user-specifiable width.

The byte-manipulation functions use objects called *byte specifiers* to designate a specific byte position within an integer. The representation of a byte specifier is implementation-dependent; it is sufficient to know that the function *byte* will construct one, and that the byte-manipulation functions will accept them. The function *byte* accepts two integers representing the *position* and *size* of the byte, and returns a byte specifier. Such a specifier designates a byte whose width is *size*, and whose right-hand bit has weight $2^{position}$, in the terminology of integers used as logical bit vectors.

`byte` *position size* [*Function*]

byte takes two integers representing the position and size of a byte, and returns a byte specifier suitable for use as an argument to byte-manipulation functions.

`byte-position` *bytespec* [*Function*]
`byte-size` *bytespec* [*Function*]

Given a byte specifier, `byte-position` returns the position specified as an integer; `byte-size` similarly returns the size.

For example:

```
(byte-position (byte j k)) <=> j
(byte-size (byte j k)) <=> k
```

`ldb` *bytespec integer* [*Function*]

bytespec specifies a byte of *integer* to be extracted. The result is returned as a positive integer.

For example:

```
(logbitp j (ldb (byte p s) n))
   <=> (and (< j s) (logbitp (+ j p) n))
```

The name of the function "`ldb`" means "load byte".

Also have a less obscure example (just give numbers as arguments and show the numerical result) `ldb-test` *bytespec integer* [*Function*]

`ldb-test` is a predicate which returns `t` if any of the bits designated by the byte specifier *bytespec* are 1's in *integer*; that is, it returns `t` if the designated field is non-zero.

```
(ldb-test bytespec n) <=> (not (zerop (ldb bytespec n)))
```

`mask-field` *bytespec integer* [*Function*]

This is similar to `ldb`; however, the result contains the specified byte of *integer* in the position specified by *bytespec*, rather than in position 0 as with `ldb`. The result therefore agrees with *integer* in the byte specified, but has zero bits everywhere else.

For example:

```
(ldb bs (mask-field bs n)) <=> (ldb bs n)
(logbitp j (mask-field (byte p s) n))
   <=> (and (>= j p) (< j s) (logbitp j n))
(mask-field bs n) <=> (logand n (ldb bs -1))
```

`dpb` *newbyte bytespec integer* [*Function*]

Returns a number which is the same as *integer* except in the bits specified by *bytespec*. Let *s* be the size specified by *bytespec*; then the low *s* bits of *newbyte* appear in the result in the byte specified by *bytespec*. The integer *newbyte* is therefore interpreted as being right-justified, as if it were the result of `ldb`. Superflous and a little confusing

For example:

```
(logbitp j (dpb m (byte p s) n))
   <=> (if (and (>= j p) (< j (+ p s)))
             (logbitp (- j p) m)
             (logbitp j n))
```

`deposit-field` *newbyte bytespec integer* [*Function*]

This function is to `mask-field` as `dpb` is to `ldb`. The result is an integer which contains the bits of *newbyte* within the byte specifried by *bytespec*, and elsewhere contains the bits of *integer*.

For example:

```
(logbitp j (dpb m (byte p s) n))
   <=> (if (and (>= j p) (< j (+ p s)))
             (logbitp j m)
             (logbitp j n))
```

wrong example

10.8. Random Numbers

`random &optional` *integer* [*Function*]

`(random)` returns a random integer, which may be positive or negative. The range of the result is implementation-dependent but reasonably large.

Implementation note: In practice the result should range over all the fixnums.

(`random` *n*) accepts a positive integer *n* and returns a non-negative integer less than *n*. Each of the possible results occurs with (approximate) frequency 1/*n*; that is, the implementation attempts to provide an (approximately) equal-chance draw from the *n* integers between 0 (inclusive) and *n* (exclusive).

Is this <=> (mod (random) n)

Need a way to reset the random number generator to a known state, and a way to seed it from a random source, to do anything nontrivial with it.

THE NAMES FOR SAID FUNCTIONS IN L.M. LISP ARE LOSERS BUT THE IDEA IS RIGHT.

Chapter 11

Characters

COMMON LISP provides a character data type; objects of this type represent printed symbols such as letters.

Every character 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).

EITHER HERE OR IN 2.2, MENTION THAT NO PARTICULAR CHAR SET IS NEEDED AND THAT SOME IMPLEMENTATIONS MAY USE FIXNUMS TO REPRESENT CHARACTERS.

`char-code-limit` [*Variable*]

The initial global value of `char-code-limit` is a non-negative integer which is the upper exclusive bound on values produced by the function `char-code` (page 101), which returns the *code* component of a given character; that is, the values returned by `char-code` are non-negative and strictly less than the value of `char-code-limit`.

Implementation note: For the PERQ, the value will be 256; for the S-1, 512.

`char-font-limit` [*Variable*]

The initial global value of `char-font-limit` is a non-negative integer which is the upper exclusive bound on values produced by the function `char-font` (page 101), which returns the *font* component of a given character; that is, the values returned by `char-font` are non-negative and strictly less than the value of `char-font-limit`.

Implementation note: No COMMON LISP implementation is required to support non-zero font attributes; if it does not, then `char-font-limit` should be 1. For the PERQ, the value will be 256; for the S-1, 512.

`char-bits-limit` [*Variable*]

The initial global value of `char-bits-limit` is a non-negative integer which is the upper exclusive bound on values produced by the function `char-bits` (page 101), which returns the *bits* component of a given character; that is, the values returned by `char-bits` are non-negative and strictly less than the value of `char-bits-limit`. Note that the value of `char-bits-limit` will be a power of two.

Implementation note: No COMMON LISP implementation is required to support non-zero bits attributes; if it does not, then `char-bits-limit` should be 1. For the PERQ, the value will be 256; for the S-1, 512.

It really does not make sense to have bits + fonts in the same character; keyboard characters & display characters are two rather different things.

11.1. Predicates on Characters

The predicate `characterp` (page 29) may be used to determine whether any LISP object is a character object.

Even in implementations that don't have them?

`standard-charp` *char* [*Function*]

The argument *char* must be a character object. `standard-charp` returns `t` if the argument is a "standard character", that is, one of the ninety-five ASCII printing characters or one of <tab>, <form>, <return>, or <rubout>. If the argument is a non-standard character, then `standard-charp` returns `()`.

I don't
understan
the motivat
for picking t
four special ch

Note in particular that any character with a non-zero *bits* or *font* attribute is non-standard.

DO YOU MEAN TO SAY TH
ALL IMPLEMENTATIONS,
ASCII OR NOT, MUST HA
THESE CHARS IN THEIR
CHAR SET, AND OTHERS
ARE OPTIONAL?

`graphicp` *char* [*Function*]

The argument *char* must be a character object. `graphicp` returns `t` if the argument is a "graphic" (printing) character, and `()` if it is a "non-graphic" (formatting or control) character. Graphic characters have a standard textual representation as a single glyph, such as "A" or "*" or "=". By convention, the space character is considered to be graphic. Of the standard characters (as defined by `standard-charp`), all but <tab>, <form>, <return>, and <rubout> are graphic.

Graphic characters of font 0 may be assumed all to be of the same width when printed; programs may depend on this for purposes of columnar formatting. Non-graphic characters and characters of other fonts may be of varying widths.

Isn't this
rather ati
the domai
a langua
definiti

Any character with a non-zero bits attribute is non-graphic.

`string-charp` *char* [*Function*]

The argument *char* must be a character object. `string-charp` returns `t` if *char* can be stored into a string (see the functions `char` (page 151) and `rplachar` (page 152)), and otherwise returns `()`. Any character which satisfies `standard-charp` and `graphicp` also satisfies `string-charp`; others may also.

`alphap` *char* [*Function*]

The argument *char* must be a character object. `alphap` returns `t` if the argument is an alphabetic character, and otherwise returns `()`.

Of the standard characters (as defined by `standard-charp`), the letters "A" through "Z" and "a" through "z" are alphabetic.

`uppercasep` *char* [*Function*]
`lowercasep` *char* [*Function*]
`bothcasep` *char* [*Function*]

The argument *char* must be a character object. `uppercasep` returns `t` if the argument is an upper-case (majuscule) character, and otherwise returns `()`. `lowercasep` returns `t` if the argument is an lower-case (minuscule) character, and otherwise returns `()`.

`bothcasep` returns `t` if the argument is upper-case *and* there is a corresponding lower-case character (which can be obtained using `char-downcase` (page 102)), or if the argument is lower-case and there is a corresponding upper-case character (which can be obtained using `char-upcase` (page 102)).

If a character is either upper-case or lower-case, it is necessarily alphabetic. However, it is permissible in theory for an alphabetic character to be neither uppercase nor lowercase.

Of the standard characters (as defined by `standard-charp`), the letters "A" through "Z" are upper-case and "a" through "z" are lower-case.

`digitp` *char* `&optional (`*radix* `10.)` [*Function*]

The argument *char* must be a character object, and *radix* must be a non-negative integer. `digitp` is a pseudo-predicate: if *char* is not a digit of the radix specified by *radix*, then it returns `()`; otherwise it returns a non-negative integer which is the "weight" of *char* in that radix.

Digits are necessarily graphic characters.

Of the standard characters (as defined by `standard-charp`), the characters "0" through "9", "A" through "Z", and "a" through "z" are digits. The weights of "0" through "9" are the integers 0 through 9, and of "A" through "Z" (and also "a" through "z") are 10 through 35. `digitp` returns the weight for one of these digits if and only if its weight is strictly less than *radix*. Thus, for example, the digits for radix 16 are "0123456789ABCDEF".

IT WOULD BE NICE IF ALL EXAMPLES GOT EVALUATED AND TESTED. WANNA BUY A LISP MACHINE?

```
(defun convert-string-to-integer (str &optional (radix 10))
  "Given a digit string and optional radix, return an integer."
  (do ((j 0 (+ j 1))
       (n 0 (+ (* n radix)
               (or (digitp (char str j) radix)
                   (ferror "Bad radix-~D digit: ~C"
                           radix
                           (char str i))))))
      ((= j (string-length str)) n)))
```

`alphanumericp` *char* [*Function*]

The argument *char* must be a character object. `alphanumericp` returns `t` if *char* is either alphabetic or numeric. By definition,

`(alphanumericp x) <=> (or (alphap x) (digitp x))`

is there a "not null" around this.
(or "and ... t")

Alphanumeric characters are therefore are necessarily graphic (as defined by `graphicp` (page 98)).

Of the standard characters (as defined by `standard-charp`), the characters "0" through "9", "A" through "Z", and "a" through "z" are alphanumeric.

Is this a convoluted way of saying that alphap and digitp return nil for characters with non-zero bits? It says something equivalent to that under digitp, but nothing under alphap.

`char=` *char1 char2* [*Function*]

The arguments *char1* and *char2* must be character objects. `char=` returns `t` if *char1* and *char2* are equivalent character objects, having equivalent attributes, and otherwise returns `()`.

case

The function `CHAR=` is the finest discriminator of characters available to the programmer. If `(char= c1 c2)` is true, then any function professing to operate on a character must behave the same whether given `c1` or `c2`.

For non-"funny" characters (those not satisfying `funny-charp` (page FUNNY-CHARP-FUN)),

↑ ???

case

```
(CHAR= C1 C2) <=>
      (AND (= (CHAR-CODE C1) (CHAR-CODE C2))
           (= (CHAR-BITS C1) (CHAR-BITS C2))
           (= (CHAR-FONT C1) (CHAR-FONT C2)))
```

There is no requirement that `(eq c1 c2)` be true merely because `(char= c1 c2)` is true. While `eq` may distinguish two character objects that `char=` does not, it is distinguishing them not as *characters*, but in some sense on the basis of a lower-level implementation characteristic. (Of course, if `(eq c1 c2)` is true then one may expect `(char= c1 c2)` to be true.) However, `eql` (page 30) and `equal` (page 31) compare character objects in the same way that `char=` does.

`char-equal` *char1 char2* [*Function*]

The arguments *char1* and *char2* must be character objects.

The predicate `char-equal` is like `char=`, except that it ignores differences of font and bits attributes and case. By definition,

```
(char-equal c1 c2) <=>
      (char= (char-upcase (character c1))
             (char-upcase (character c2)))
```

For example:

```
(char-equal #\A #\a) => t
(char= #\A #\a) => ()
(char-equal #\A (CONTROL #\A)) t
```

case

`char<` *char1 char2* [*Function*]
`char>` *char1 char2* [*Function*]

The arguments *char1* abd *char2* must be character objects. The predicate `char<` is true if *char1* precedes *char2* in the (implementation-dependent) total ordering on characters. The predicate `char>` is true if *char1* follows *char2* in the (implementation-dependent) total ordering on characters. Neither is true if the arguments satisfy `char=` (page 100).

The total ordering on characters is guaranteed to have the following properties:

- The alphanumeric characters obey the following partial ordering:

```
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
0<1<2<3<4<5<6<7<8<9
either 9<A or Z<0
either 9<a or z<0
```

This implies that alphabetic ordering holds, and that the digits as a group are not interleaved with letters, but that the possible interleaving of upper-case letters and lower-case letters is unspecified.

- If two characters have the same bits and font attributes, then their ordering by `char<` is consistent with the numerical ordering by the predicate `<` (page 83) on their code attributes.

`char-lessp` *char1 char2* [*Function*]

`char-greaterp` *char1 char2* [*Function*]

The arguments *char1* and *char2* must be character objects. The predicate `char-lessp` is like `char<`, except that it ignores differences of font and bits attributes and case; similarly `char-greaterp` is like `char>`. By definition,

```
(char-lessp c1 c2) <=>
     (char< (char-upcase (character c1))
            (char-upcase (character c2)))
```

11.2. Character Construction and Selection

`char-code` *char* [*Function*]

The argument *char* must be a character object. `char-code` returns the *code* attribute of the character object; this will be a non-negative integer less than the (normal) value of the variable `char-code-limit` (page 97).

`char-bits` *char* [*Function*]

The argument *char* must be a character object. `char-bits` returns the *bits* attribute of the character object; this will be a non-negative integer less than the (normal) value of the variable `char-bits-limit` (page 97).

`char-font` *char* [*Function*]

The argument *char* must be a character object. `char-font` returns the *font* attribute of the character object; this will be a non-negative integer less than the (normal) value of the variable `char-font-limit` (page 97).

`code-char` *code* `&optional` `(`*bits* `0)` `(`*font* `0)` [*Function*]

All three arguments must be non-negative integers. If it is possible in the implementation to construct a character object whose code attribute is *code*, whose bits attribute is *bits*, and whose font attribute is *font*, then such an object is returned; otherwise `( )` is returned.

For any integers *c*, *b*, and *f*, if `(code-char` *c b f*`)` is not `( )` then

```
(char-code (code-char c b f)) => c
(char-bits (code-char c b f)) => b
(char-font (code-char c b f)) => f
```

If the font and bits attributes of a character object *x* are zero, then it is the case that

```
(char= (code-char (char-code c)) c) => t
```

For both of these,
it would be better
to signal an error
than to return nil.
Both for better robustnes
& carelessly-written
and to permit type
declaration and
associated
optimization
convention ma

`character` *char* `&optional` `(`*bits* `0)` `(`*font* `0)` [*Function*]

`character` is similar to `code-char` (page 102) except that the first argument is already a character object.

Why not just make code-char accept either?

The argument *char* must be a character object, and *bits* and *font* non-negative integers. If it is possible in the implementation to construct a character object whose code attribute is that of *char*, whose bits attribute is *bits*, and whose font attribute is *font*, then such an object is returned; otherwise `( )` is returned.

If *bits* and *font* are zero, then `character` will not return `( )`. This implies that for every character object one can "turn off" its bits and font attributes.

This function is incompatible
with usual interpretation as a
type-coercion function.
Character of a non-character is an error
and furthermore character of a
character does not necessarily return
The same character.

||

11.3. Character Conversions

`char-upcase` *char* [*Function*]
`char-downcase` *char* [*Function*]

The argument *char* must be a character object. `char-upcase` attempts to convert its argument to an upper-case equivalent; `char-downcase` attempts to convert to lower case.

`char-upcase` returns a character object with the same font and bits attributes as *char*, but with possibly a different code attribute. If the code is different from *char*'s, then the predicate `lowercasep` (page 98) is true of *char*, and `uppercasep` (page 98) is true of the result character. Moreover, if `(char= (char-upcase x) x)` is *not* true, then it is true that

```
(char= (char-downcase (char-upcase x)) x)
```

Similarly, `char-downcase` returns a character object with the same font and bits attributes as *char*, but with possibly a different code attribute. If the code is different from *char*'s, then the predicate `uppercasep` (page 98) is true of *char*, and `lowercasep` (page 98) is true of the result character. Moreover, if `(char= (char-downcase x) x)` is *not* true, then it is true that

```
(char= (char-upcase (char-downcase x)) x)
```

`digit-char` *weight* `&optional` (*radix* `10.`) (*bits* `0`) (*font* `0`) [*Function*]

All arguments must be integers. `digit-char` returns a character object whose bits attribute is *bits*, whose font attribute is *font*, and whose *code* is such that the result character has the weight *weight* when considered as a digit of the radix *radix* (see the predicate `digitp` (page 99)), if that is possible; if that cannot be done, `digit-char` returns `()`. `digit-char` does not return `()` if *bits* and *font* are zero, *radix* is between 2 and 36 inclusive, and *weight* is non-negative and less than *radix*. If more than one character object can encode such a weight in the given radix, one shall be chosen consistently by any given implementation; moreover, among the standard characters upper-case letters are preferred to lower-case letters).

Some comment applies

For example:

```
(digit-char 7) => #\7
(digit-char 12) => ()
(digit-char 12 16) => #\C          ;not #\c
(digit-char 6 2) => ()
(digit-char 1 2) => #\1
```

WHAT'S THAT? WHAT TYPE?

`char-int` *char* [*Function*]

The argument *char* must be a character object, or the object `#\EOF`. If *char* is a character object, `char-int` returns a non-negative integer; if *char* is `#\EOF`, the result is `-1`.

If the font and bits attributes of *char* are zero, then `char-int` returns the same integer `char-code` would. Also,

```
(char= c1 c2) <=> (= (char-int c1) (char-int c2))
```

for characters `c1` and *c2*. font

This function is provided primarily for the purpose of hashing characters. Also, the function `tyi` (page 213) is defined in terms of `char-int`.

`int-char` *integer* [*Function*]

ital?

The argument must be a non-negative integer. `int-char` returns a character object `c` such that `(char-int c)` is equal to *integer*, if possible; otherwise `int-char` returns `()`. Note that *integer* may not be `-1`.

DOES THIS MEAN THAT THE MAPPING MUST BE ONE-TO-ONE ONTO? IF SO, SAY SO EXPLICITLY.

Some comment applies

ARE THERE MANY?

`char-name` *char* [*Function*]

The argument *char* must be a character object or an end-of-file object. If the character has a name, then that name (a symbol) is returned; for an end-of-file object the name `eof` is returned; otherwise `()` is returned. All characters which have zero font and bits attributes and which are non-graphic (do not satisfy the predicate `graphicp` (page 98)) have names. Graphic characters may or may not have names.

The standard characters `<tab>`, `<form>`, `<return>`, `<rubout>`, and `<space>` have the respective names `tab`, `form`, `return`, `rubout`, and `space`.

Characters which have names can be notated as "#\" followed by the name: `#\Space`.

`name-char` *sym* [*Function*]

The argument `sym` must be a symbol. If the symbol is the name of a character object, that object is returned; if the symbol is `eof`, an end-of-file object is returned; and otherwise `()` is returned.

11.4. Character Control-Bit Functions

COMMON LISP provides explicit names for four bits of the bits attribute: *Control*, *Meta*, *Hyper*, and *Super*. The following definitions are provided for manipulating these. Each COMMON LISP implementation provides these functions for compatibility, even if it does not support any or all of the bits named below.

`char-control-bit` [*Variable*]
`char-meta-bit` [*Variable*]
`char-super-bit` [*Variable*]
`char-hyper-bit` [*Variable*]

The initial values of these variables are the "weights" for the four named control bits. The weight of the control bit is `1`; of the meta bit, `2`; of the super bit, `4`; and of the hyper bit, `8`.

If a given implementation of COMMON LISP does not support a particular bit, then the corresponding variable is zero instead.

`controlp` *char* [*Function*]
`metap` *char* [*Function*]
`superp` *char* [*Function*]
`hyperp` *char* [*Function*]

The argument *char* must be a character object. If the control bit is set within the bits attribute of *char*, then `controlp` returns `t`, and otherwise returns `()`. Similarly `metap` tests the meta bit, `superp` the super bit, and `hyperp` the hyper bit.

`control` *char* [*Function*]
`meta` *char* [*Function*]
`super` *char* [*Function*]
`hyper` *char* [*Function*]

The argument *char* must be a character object or `()`. If the argument is `()`, the result is `()`. Otherwise, consider the function `control`; the other operate similarly. If `controlp` is true of *char*, then *char* is returned. Otherwise, if it is possible to construct a character object with the same code and font attributes, and with the same bits attribute but with the control bit "turned on", then such a character object is returned, and otherwise `()` is returned.

s

Motivatio
I BET IT'S
YOU CAN NE
TWO OF THESE
I AGREE TH
THIS IS A JO
EXCEPTIONAL
CONDITIONS.

I don't think these words should be used for these functions by themselves. Put in "char-" or "-char"

Same comment as on code-char

`uncontrol` *char* [*Function*]
`unmeta` *char* [*Function*]
`unsuper` *char* [*Function*]
`unhyper` *char* [*Function*]

The argument *char* must be a character object or `()`. If the argument is `()`, the result is `()`. Otherwise, consider the function `uncontrol`; the other operate similarly. If `controlp` is false of *char*, then *char* is returned. Otherwise, a character object is returned with the same code and font attributes, and with the same bits attribute but with the control bit "turned off" (this is always possible).

Same comments as above.

Chapter 12

Sequences

NOT REALLY: CAN HAVE DUPLICATES. "SEQUENCE" IS REALLY MORE ACCURATE

The type `sequence` encompasses objects of type `list`, `vector`, and `array`. 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. In the case of lists and vectors, the ordering of the elements is "natural", following the total ordering on the integer indexes of the elements. In the case of arrays, the ordering of the elements follows the lexicographic ordering of the index sequences for the elements, and so the elements are considered to be arranged in "row-major" order. If an array is given to a generic sequence function, then any indices involved are not array indices (unless the array is one-dimensional), but rather indices in the row-major ordering. GIVE AN EXAMPLE OF ROW-MAJOR, EG. (0 0) (0
(0 2) (1 0

There are some operations which are useful on lists, vectors, and 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. There are type-specific versions of these operations as well, which may be used for declarative or error-checking purposes. DITT

These are the operations defined on sequences:

`elt`
`setelt`
`subseq`
`copyseq`
`length`
`fill`
`replace`

`reverse`
`nreverse`
`concat`
`reduce`
`left-reduce`
`right-reduce`
`sort`

`map`
`some`
`every`
`notany`
`notevery`
`merge`
`nmerge`

`remove`
`position`
`scan-over`
`count`
`mismatch`
`maxprefix`
`maxsuffix`
`search`

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>
`remove`	`remove-from-end`	Compare elements using `equal`
`remq`	`remq-from-end`	Compare elements using `eq`
`rem`	`rem-from-end`	Compare elements with user predicate
`rem-if`	`rem-from-end-if`	Test elements with user predicate
`rem-if-not`	`rem-from-end-if-not`	Test elements with inverse user predicate

I sent you (GLS) a message some time back suggesting that there are too many names, the name-abbreviation is not always consistent enough to be easy to use, and optional arguments can be used. I repeat the suggestion. (Although I can see arguments against it.)

As a rule, for each of these names *x* there is a generic function named *x* which operates on sequences. There are also type-specific functions as follows:

Name	Type of sequence operated upon
`list-`*x*	Lists
`bit-`*x*	Bit vectors
`string-`*x*	Strings
`v`*x*	General vectors (those of type `(vector t)`)
`v`*x*`@`	Vectors of a type indicated by the first argument

Use of such a type-specific function implies that any sequence arguments must be of the specified type, any arguments stored into or compared with elements of a sequence must be of an appropriate type, and that the result will be a sequence or element of the appropriate type.

> **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. `list-rem-from-end-if-not` seems to be a conceptually atomic operation, for example, despite the fact that its name is made from four 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. OK

`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`.

See `list-elt` (page 138), `nth` (page 125), `aref`, `vref` (page 162), `bit`, `char` (page 151), and `vref@` (page 170).

FOO. ARG ORDER COMPLETELY
INCOMPATIBLE WITH ASET.
THIS WOULD BE CONFUSING.

`setelt` *sequence index newvalue* Unfortunate argument order. [*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 vector, then the *newvalue* must be an object which that vector can contain.

See `list-setelt` (page 138), `setnth` (page SETNTH-FUN), `aset` (page 177), `vset` (page 162), `rplacbit`, `rplachar` (page 152), and `vset@` (page 170).

`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` (as with all its type-specific variants) *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*.

See `sublist` (page 138), `subvector` (page SUBVECTOR-FUN), `subvec`, `substring`, `sub-bits` (page 166), and `subvec@` (page 171).

is thi real hyphen, or a mistake by Scri

Compatibility note: Although this function and most of the others in this chapter take their names from those proposed for NIL, they use the *start* and *end* convention for delimiting substrings as in Lisp Machine LISP, rather than the *start* and *count* convention. While the latter seems to be somewhat more convenient for certain contemporary hardware such as the VAX and S-1, and therefore for their compilers, the former seems to be far more convenient for the user (according to an informal poll). This would seem to be an overriding consideration.

`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.

See `copylist` (page 126), `copyvector` (page COPYVECTOR-FUN), `copyvec`, `copystring`, `copybits` (page 166), and `copyvec@` (page 171).

`length` *sequence* [*Function*]

The number of elements in *sequence* is returned as a non-negative integer. Note that `length` and `list-length` behave differently when given a vector; `length` returns the length of the vector, while `list-length` always returns zero.

e

See `list-length` (page 124), `array-length` (page 177), `vlength` (page 163), `bit-length` (page 166), `string-length` (page 155), and `vlength@` (page 171).

`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)
```

See `list-fill` (page 138), `vfill` (page 163), `bit-fill` (page 166), `string-fill` (page 155), and `vfill@` (page 171).

`replace` *target-sequence source-sequence* `&optional` *target-start source-start target-end 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))
```

In this and all others that take a double start/end pair, it is necessary that nil as an end be the same as omitting it.

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 the results are undefined.

See `list-replace` (page 138), `vreplace` (page 163), `bit-replace` (page 166), `string-replace` (page 155), and `vreplace@` (page 171).

`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. SEE p127 — IS REVERSE OF A DOTTED LIST THE SAME AS LIST-REVERSE OF IT?

See `list-reverse` (page 127), `vreverse` (page 163), `bit-reverse` (page 166), `string-reverse` (page 155), and `vreverse@` (page 171).

`nreverse` *sequence* [*Function*]

The result is a sequence containing the same elements as *sequence* but in reverse order. The argument is 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`.

See `list-nreverse` (page 127), `vnreverse` (page 163), `bit-nreverse` (page 166), `string-nreverse` (page 155), and `vnreverse@` (page 171).

enate

← avoid useless abbreviation, it only makes names harder to remember. This is not used so frequently that it needs to be shorter.

`concat` `&rest` *sequences* [*Function*]

I AGREE. ABBREVIATIO ARE INSIDIO

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 `concat` 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 `concat` 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, `concat` returns `()`. — I am dubious about this; is () better than #() or "" or #"" or...

See `list-concat` (page 138), `vconcat` (page 163), `bit-concat` (page 166), `string-concat` (page 155), and `vconcat@` (page 171).

`reduce` *function sequence* `&optional` *start-value* [*Function*]
`left-reduce` *function sequence* `&optional` *start-value* [*Function*]
`right-reduce` *function sequence* `&optional` *start-value* [*Function*]

These functions are similar to the reduction operator of APL. The *function* must be a function of two arguments which can operate on elements of the *sequence*. In general, the result is produced by using *function* to accumulate the elements of the *sequence*. If the argument *start-value* is provided, it is used to initialize the accumulation; in this case the *sequence* may be empty (in which case the result is *start-value*). If *start-value* is not provided, the *sequence* may not be empty. (Does this m
The first element is used as the star
value?
Evide
but the w
that par
is writ
is conf

For brevity, let the *function* be called *f*, and let the elements of the *sequence* be called *x1*, *x2*, . . . , *xn*. Then the result of `left-reduce` with two arguments is:

```
(f ... (f (f x1 x2) x3) ... xn)
```

That is, the function *f* is applied to the elements left-associatively. If a *start-value* is provided, then the result is:

```
(f ... (f (f (f start-value x1) x2) x3) ... xn)
```

The result of `right-reduce` is similar, but the elements are right-associated:

```
(f x1 (f x2 ... (f xn-1 xn) ...))
(f x1 (f x2 ... (f xn-1 (f xn start-value) ...))
```

The result of `reduce` is similar to these, but the *function* is assumed to be associative (and additionally assumed to be commutative if a *start-value* is provided), and so the elements may be associated in any manner the implementation desires. that's totally random.

For example:

```
(reduce #'+ '(1 2 3 4 5)) => 15
(reduce #'* x) => product of elements of x, which must be non-empty
(reduce #'* x 1) => product of elements of x, which may be non-empty
```

Note that frequently the *start-value* ought to be the identity for the *function*.

```
(right-reduce #'cons "abcd") => (#\a #\b #\c . #\d)
(right-reduce #'cons "abcd" '()) => (#\a #\b #\c #\d)
(left-reduce #'cons "abcd") => (((#\a . #\b) . #\c) . #\d)
(left-reduce #'cons "abcd" '())
  => (((( () . #\a) . #\b) . #\c) . #\d)
(reduce #'cons "abcd") => unpredictable (cons is not associative)
```

See `list-reduce` (page 139), `vreduce` (page 163), `bit-reduce` (page 166), `string-reduce` (page 156), and `vreduce@` (page 171).

`map` *function* `&rest` *sequences* [*Function*]

MODULO &OPTIONAL AND &REST OF COURSE

The *function* must take as many arguments as there are *sequences* 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*. As a boundary case, if no sequences are given, the *function* must take no arguments, and it is called indefinitely many times; the call to `map` will normally never terminate.

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 implementation-dependent; a type-specific function can be used to specify the argument and result sequence types. See `list-map` (page 139), `vmap` (page 163), `bit-map` (page 166), `string-map` (page 156), and `vmap@` (page 171).

The result sequence type should not be tied to the argument sequence 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 52).

OK

`some` *predicate* `&rest` *sequences* [*Function*]
`every` *predicate* `&rest` *sequences* [*Function*]
`notany` *predicate* `&rest` *sequences* [*Function*]
`notevery` *predicate* `&rest` *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 `t`. 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 `t`. Thus as a predicate it is true if *no* invocation of *predicate* is true.

`notevery` returns `t` as soon as any invocation of *predicate* returns `()`. If the end of a sequence is reached, `notevery` 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.

If no *sequences* are given, then the *predicate* must be able take no arguments. In this case, the *predicate* is called repeatedly; `some` and `notany` return only if *predicate* ever returns a non-`()` value, and `every` and `notevery` return only if *predicate* ever returns `()`.

See `list-some` (page 139), `vsome` (page 163), `bit-some` (page 166), `string-some` (page 156), and `vsome@` (page 171), and related functions.

`remove` *item sequence* `&optional` *count* [*Function*]
`remq` *item sequence* `&optional` *count* [*Function*]
`rem` *predicate item sequence* `&optional` *count* [*Function*]
`rem-if` *predicate sequence* `&optional` *count* [*Function*]
`rem-if-not` *predicate sequence* `&optional` *count* [*Function*]
`remove-from-end` *item sequence* `&optional` *count* [*Function*]
`remq-from-end` *item sequence* `&optional` *count* [*Function*]
`rem-from-end` *predicate item sequence* `&optional` *count* [*Function*]
`rem-from-end-if` *predicate sequence* `&optional` *count* [*Function*]
`rem-from-end-if-not` *predicate sequence* `&optional` *count* [*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 is *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.

NOT IF FROM-END SAY "FIRST"?

The `-from-end` variants differ from the others only when *count* is provided; in that case only the rightmost *count* elemnts 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.

See `list-remove` (page 139), `vremove` (page 163), `bit-remove` (page 167), `string-remove` (page 156), and `vremove@` (page 171), and related functions.

`position` *item sequence* `&optional` *start end* [*Function*]
`posq` *item sequence* `&optional` *start end* [*Function*]
`pos` *predicate item sequence* `&optional` *start end* [*Function*]
`pos-if` *predicate sequence* `&optional` *start end* [*Function*]
`pos-if-not` *predicate sequence* `&optional` *start end* [*Function*]
`position-from-end` *item sequence* `&optional` *start end* [*Function*]
`posq-from-end` *item sequence* `&optional` *start end* [*Function*]
`pos-from-end` *predicate item sequence* `&optional` *start end* [*Function*]
`pos-from-end-if` *predicate sequence* `&optional` *start end* [*Function*]
`pos-from-end-if-not` *predicate sequence* `&optional` *start end* [*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 is *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.

See `list-position` (page 139), `vposition` (page 164), `bit-position` (page 167), `string-position` (page 156), and `vposition@` (page 172), and related functions.

`scan-over` *item sequence* `&optional` *start end* [*Function*]
`scanq` *item sequence* `&optional` *start end* [*Function*]
`scan` *predicate item sequence* `&optional` *start end* [*Function*]
`scan-if` *predicate sequence* `&optional` *start end* [*Function*]
`scan-if-not` *predicate sequence* `&optional` *start end* [*Function*]

Here is a typical example where the equal/eq/user naming convention breaks down hopelessly. Make the predicate always an argument.

`scan-over-from-end` *item sequence* `&optional` *start end* [*Function*]
`scanq-from-end` *item sequence* `&optional` *start end* [*Function*]
`scan-from-end` *predicate item sequence* `&optional` *start end* [*Function*]
`scan-from-end-if` *predicate sequence* `&optional` *start end* [*Function*]
`scan-from-end-if-not` *predicate sequence* `&optional` *start end* [*Function*]

If the *sequence* contains an element *failing* a certain test, then the index within the sequence of the leftmost such element is returned as a non-negative integer; otherwise `()` is returned. In other words, elements *satisfying* the test are scanned over.

For `scan-over`, an element passes the test if *item* is `equal` to it; therefore `scan-over` scans for an element to which *item* is not `equal`.

For `scanq`, an element passes the test if *item* is `eq` to it; therefore `scanq` scans for an element to which *item* is not `eq`.

For `scan`, an element passes the test is *predicate* is true when applied to *item* and an element (in that order); therefore `scan` scans for an element for which the *predicate* is false when so applied.

For `scan-if`, an element passes the test if *predicate* is true of it; therefore `scan-if` is the same as `pos-if-not` (page 114).

For `scan-if-not`, an element passes the test if *predicate* is not true of it; therefore `scan-if-not` is the same as `pos-if` (page 114).

The `-from-end` variants differ in that the index of the *rightmost* element failing 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, `scan-over-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.

See `list-scan-over` (page 140), `vscan-over` (page 164), `bit-scan-over` (page 168), `string-scan-over` (page 157), and `vscan-over@` (page 172), and related functions.

??? Query: I am not excited at all over these names. In NIL these were called `skip`, `skpq`, `skp`, and so on; - YECCH!! Fahlman and others have objected to those names. One idea is to can them and just use `pos`:

```
(scanq x s) <=> (pos #'(lambda (a b) (not (eq a b))) x s)
```

Any other suggestions?

`count` *item sequence* `&optional` *start end* [*Function*]
`cntq` *item sequence* `&optional` *start end* [*Function*]
`cnt` *predicate item sequence* `&optional` *start end* [*Function*]
`cnt-if` *predicate sequence* `&optional` *start end* [*Function*]
`cnt-if-not` *predicate sequence* `&optional` *start end* [*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 is *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-vectors 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.

font

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.

See `list-count` (page 140), `vcount` (page 164), `bit-count` (page 168), `string-count` (page 157), and `vcount@` (page 172), and related functions.

`mismatch` *sequence1 sequence2* `&optional` *start1 start2 end1 end2* [*Function*]
`mismatq` *sequence1 sequence2* `&optional` *start1 start2 end1 end2* [*Function*]
`mismat` *predicate sequence1 sequence2* `&optional` *start1 start2 end1 end2* [*Function*]
`mismatch-from-end` *sequence1 sequence2* `&optional` *start1 start2 end1 end2* [*Function*]
`mismatq-from-end` *sequence1 sequence2* `&optional` *start1 start2 end1 end2* [*Function*]
`mismat-from-end` *predicate sequence1 sequence2* `&optional` *start1 start2 end1 end2* [*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 `mismatq`, elements are compared using `eq`.

For `mismat`, 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

Again, would be better to have only mismat and mismat-from-end, and Flush the other four. Call them mismatch and mismatch-from-end, of course.

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.

See `list-mismatch` (page 140), `vmismatch` (page 164), `bit-mismatch` (page 168), `string-mismatch` (page 157), and `vmismatch@` (page 172), and related functions.

`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.

See `list-maxprefix` (page 140), `vmaxprefix` (page 165), `bit-maxprefix` (page 168), `string-maxprefix` (page 158), and `vmaxprefix@` (page 173), and related functions.

`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 `srchq`, elements are compared using `eq`.

For `srch`, 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.

See `list-search` (page 141), `vsearch` (page 165), `bit-search` (page 169), `string-search` (page 158), and `vsearch@` (page 173), and related functions.

`sort` *sequence predicate* [*Function*]
`sortcar` *sequence predicate* [*Function*]
`sortslot` *sequence key-function predicate* [*Function*]

For sort + merge, it might be better to make key-functi an optional.

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 the elements to the *predicate*. `sortcar` assumes that all elements of the *sequence* are lists, and gives the *car* of each element to the *predicate*; we say that the *car* of each element is the *sort key*, and the *cdr* is other data associated with the key. `sortslot` allows an arbitrary *key-function* to determine the key, given an element. The *key-function* should not have any side effects. A useful example of a key function would be a component selector function for a `defstruct` (page 181) structure, for sorting a sequence of structures.

```
(sortslot a f p)
   <=> (sort a #'(lambda (x y) (p (f x) (f y))))
```

While the above two expressions are equivalent, `sortslot` may be more efficient in some implementations for certain types of arguments. For example, an implementation may choose to apply *key-function* to each item just once, putting the resulting keys into a separate table, and then sort the parallel tables, as opposed to applying *key-function* to an item every time just before applying the *predicate*.

If the *predicate* always returns, 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 predicate does reflect some total ordering criterion, then the elements of the result sequence will conform to that ordering. The sorting operation is not guaranteed *stable*, however; elements considered equal by the *predicate* may or may not stay in their original order.

The sorting operation is destructive in all cases. In the case of an array or vector argument, this is accomplished by permuting the elements; sorting an array means rearranging the elements so that they are sorted with respect to row-major order. In the case of a list, the list is destructively reordered in the same manner as for `nreverse` (page 110). Thus if the argument should not be destroyed, the user must sort a copy of the argument. Also must look at result, not guaranteed eq to arg although it sometime is.

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

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
      #'(lambda (x y)
          (string-lessp (mostcar x) (mostcar y))))
```

If `fooarray` contained these items before the sort:

```
(Tokens (The lion sleeps tonight))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
((Beach Boys) (I get around))
(Beatles (I want to hold your hand))
```

then after the sort `fooarray` would contain:

Should also have
stable-sort
(+ 2 friends)

```
((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))
```

See `list-sort` (page 141), `vsort` (page 165), `bit-sort` (page 169), `string-sort` (page 158), and `vsort@` (page 173), and related functions.

`merge` *sequence1 sequence2 predicate* [*Function*]
`mergecar` *sequence1 sequence2 predicate* [*Function*]
`mergeslot` *sequence1 sequence2 key-function predicate* [*Function*]

The sequences *sequence1* and *sequence2* are nondestructively 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 the elements to the *predicate*. `mergecar` assumes that all elements of the *sequence* are lists, and gives the *car* of each element to the *predicate*; we say that the *car* of each element is the *merge key*, and the *cdr* is other data associated with the key. `mergeslot` allows an arbitrary *key-function* to determine the key, given an element. The *key-function* should not have any side effects. A useful example of a key function would be a component selector function for a `defstruct` (page 181) structure, for merging a sequence of structures.

If the *predicate* always returns, then the merging operation will always terminate, producing a sequence containing the same elements as the two input sequences (that is, the result is a permutation of the concatenation of *sequence1* and *sequence2*). This is guaranteed even if the *predicate* does not really consistently represent a total order. If the predicate does reflect some total ordering criterion, and each of the input sequences was already sorted according to this ordering, then the elements of the result sequence will conform to that ordering. The merging operation is not guaranteed *stable*, however; if two or more elements are considered equal by the *predicate*, then the elements from *sequence1* may or may not precede those from *sequence2* in the result.

The merging operation is non-destructive; however, the result may share structure with the inputs.

For example:

```
(merge '(1 3 4 6 7) '(2 5 8) #'<) => (1 2 3 4 5 6 7 8)
```

See `list-merge` (page 141), `vmerge` (page 165), `bit-merge` (page 169), `string-merge` (page 158), and `vmerge@` (page 173), and related functions.

`nmerge` *sequence1 sequence2 predicate* [*Function*]
`nmergecar` *sequence1 sequence2 predicate* [*Function*]
`nmergeslot` *sequence1 sequence2 key-function predicate* [*Function*]

These functions are exactly like `merge`, `mergecar`, and `mergeslot` (page 120), except that these may perform the merging operation destructively. The input sequences may be destroyed, and/or the result may share structure with the input sequences.

See `list-nmerge` (page 141), `vnmerge` (page 165), `bit-nmerge` (page 169), `string-nmerge` (page 158), and `vnmerge@` (page 173), and related functions.

I DON'T THINK I UNDERSTAND WHAT IT IS TO MERGE — WHAT WOULD BE THE RESULT IF ONE OR BOTH INPUT SEQUENCES WERE UNSORTED?

Specify what happens with dotted lists and be consistent everywhere

Chapter 13

Manipulating List Structure

A *cons*, or dotted pair, is a compound data object having two components, called the *car* and *cdr*. Each component may be any LISP object. A *list* is a chain of conses linked by *cdr* fields; the chain is terminated by some atom (a non-cons object). An ordinary list is terminated by `()`, the empty list. A list whose *cdr*-chain is terminated by some non-`()` atom is called a *dotted list*, }

13.1. Conses

`car` *x* [*Function*]

Returns the *car* of *x*, which must be a cons or `()`; that is, *x* must satisfy the predicate `listp` (page 27). By definition, the *car* of `()` is `()`. If the cons is regarded as the first cons of a list, then `car` returns the first element of the list.

For example:

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

`cdr` *x* [*Function*]

Returns the *cdr* of *x*, which must be a cons or `()`; that is, *x* must satisfy the predicate `listp` (page 27). By definition, the *cdr* of `()` is `()`. If the cons is regarded as the first cons of a list, then `cdr` returns the rest of the list, which is a list with all elements but the first of the original list.

For example:

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

`c...r` *x* [*Function*]

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

For example:

`(cddadr x)` is the same as `(cdr (cdr (car (cdr x))))`

If the argument is regarded as a list, then `cadr` returns the second element of the list, `caddr` the third, and `cadddr` the fourth. If the first element of a list is a list, then `caar` is the first element of the sublist, `cdar` is the rest of that sublist, and `cadar` is the second element of the sublist; and so on.

As a matter of style, it is often preferable to define a function or macro to access part of a complicated data structure, rather than to use a long `car/cdr` string:

```
(defmacro imag-part (complexnum) '(cadr ,complexnum))
    ; then use imag-part everywhere instead of cadr
```

`cons` *x y* [*Function*]

`cons` is the primitive function to create a new *cons*, whose *car* is *x* and whose *cdr* is *y*.

For example:

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

`cons` may be thought of as creating a *cons*, or as adding a new element to the front of a list.

`tree-equal` *x y* [*Function*]

This is a predicate which returns `t` if *x* and *y* are isomorphic trees with identical leaves; that is, if *x* and *y* are `eq`, or if they are both conses and their *cars* are `tree-equal` and their *cdrs* are `tree-equal`. Thus `tree-equal` recursively compares conses (but not any other objects which have components). See `equal` (page 31), which does recursively compare other structured objects.

13.2. Lists

`list-length` *list* `&optional` *limit* [*Function*]

`list-length` returns, as an integer, the length of *list*. The length of a list is the number of top-level conses in it. If the argument *limit* is supplied, it should be an integer; if the length of the *list* is greater than *limit* (possibly because the *list* is circular!), then some integer no smaller than *limit* and no larger than the length of the list is returned.

Rationale: Allowing this vague definition of the meaning of *limit* allows certain tricky fast implementations.

For example:

```
(length '()) => 0
(length '(a b c d)) => 4
(length '(a (b c) d)) => 3
(length '(a b c d e f g) 4) => 4 or 5 or 6 or 7
```

`length` could be implemented by:

```
(defun list-length (x &optional (limit () limitp))
    (do ((n 0 (+ n 1))
         (y x (cdr y)))
        ((atom y) n)
      (when (and limitp (>= n limit))
        (return n))))
```

See `length` (page 109), which will return the length of any sequence. One difference between `length` and `list-length` is that `length` of a vector returns the length of the vector, while `list-length` of a vector returns 0.

Should be an error.

YES! THIS "DOTTED LIST" BUSINESS SHOULD NOT BE USURPING ALL OF LISP'S ERROR CHECKING!

All of these should barf if list arg is not list

`nth` *n list* [*Function*]

(`nth` *n list*) returns the *n*'th element of *list*, where the zeroth element is the *car* of the list. *n* must be a non-negative integer. If the length of the list is not greater than *n*, then the result is `()`. (This is consistent with the idea that the *car* and *cdr* of `()` are each `()`.)

For example:

```
(nth 0 '(foo bar gack)) => foo
(nth 1 '(foo bar gack)) => bar
(nth 3 '(foo bar gack)) => ()
```

This function is slightly different from `list-elt` (page 138); note also that the order of arguments is reversed.

> Compatibility note: This is not the same as the INTERLISP function called `nth`, which is similar to but not exactly the same as the COMMON LISP function `nthcdr`. This definition of `nth` is compatible with Lisp Machine LISP and NIL. Also, some people have used macros and functions called `nth` of their own in their old MACLISP programs, which may not work the same way; be careful.

PUT IN SETNTH HERE ...

`nthcdr` *n list* [*Function*]

(`nthcdr` *n list*) performs the `cdr` operation *n* times on *list*, and returns the result.

For example:

```
(nthcdr 0 '(a b c)) => (a b c)
(nthcdr 2 '(a b c)) => (c)
(nthcdr 4 '(a b c)) => ()
```

In other words, it returns the *n*'th *cdr* of the list.

> Compatibility note: This is similar to the INTERLISP function `nth`, except that the INTERLISP function is one-based instead of zero-based.

```
(car (nthcdr n x)) <=> (nth n x)
```

`last` *list* [*Function*]

`last` returns the last cons (*not* the last element!) of *list*. If *list* is `()`, it returns `()`.

For example:

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

`list &rest` *args* [*Function*]

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

For example:

Form

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

`list*` `arg` `&rest` *others* [*Function*]

`list*` is like `list` except that the last *cons* of the constructed list is "dotted". The last argument to `list*` is used as the *cdr* of the last cons constructed; this need not be an atom. If it is not an atom, then the effect is to add several new elements to the front of a list.

For example:

```
(list* 'a 'b 'c 'd) => (a b c . d)
This is like
(cons 'a (cons 'b (cons 'c 'd)))
Also:
(list* 'a 'b 'c '(d e f)) => (a b c d e f)
(list* x) <=> x
```

|| `make-list` *size* `&optional` *value* [*Function*]

This creates and returns a list containing *size* elements, each of which is *value* (which defaults to `()`). *size* should be a non-negative integer.

Make this compatible with Lisp machine (takes keyword argum

Yes!

For example:

```
(make-list 5) => (() () () () ())
(make-list 3 '() 'rah) => (rah rah rah)
```

Compatibility note: The Lisp Machine LISP function `make-list` takes arguments *area* and *size*. Areas are not relevant to COMMON LISP. The argument order used here is compatible with NIL.

It doesn't (it once used to.)

`append` `&rest` *lists* [*Function*]

The arguments to `append` are lists. The result is a list which is the concatenation of the arguments. The arguments are not destroyed.

For example:

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

Note that `append` copies the top-level list structure of each of its arguments *except* the last. The function `list-concat` (page 138) performs a similar operation, but copies all its arguments. See also `nconc` (page 128), which is like `append` but destroys all arguments but the last.

`(append` *x* `'())` is an idiom once frequently used to copy the list *x*, but the `copylist` function is more appropriate to this task.

`copylist` *list* [*Function*]

Returns a list which is `equal` to *list*, but not `eq`. Only the top level of list-structure is copied; that is, `copylist` copies in the *cdr* direction but not in the *car* direction. If the list is "dotted", that is, `(cdr (last` *list*`))` is a non-`()` atom, this will be true of the returned list also. See also `copyseq` (page 109).

`copyalist` *list* [*Function*]

`copyalist` is for copying association lists. The top level of list structure of *list* is copied, just as `copylist` does. In addition, each element of *list* which is a cons is replaced in the copy by a new cons with the same car and cdr.

`copytree` *object* [*Function*]

`copytree` is for copying trees of conses. The argument *object* may be any LISP object. If it is not a cons, it is returned; otherwise the result is a new cons of the results of calling `copytree` on the car and cdr of the argument. In other words, all conses in the tree are copied recursively, stopping only when non-conses are encountered. Circularities and the sharing of substructure are *not* preserved.

It isn't that circularities aren't preserved, they cause it to los

`list-reverse` *list* [*Function*]

`list-reverse` creates a new list whose elements are the elements of *list* taken in reverse order. `list-reverse` does not modify its argument, unlike `list-nreverse` (page 127) which is faster but does modify its argument. If the *list* is dotted, the non-`()` atom at the end is discarded; `reverse` always produces a list ending in `()`. — BUT, AS MOON SAID, DON'T CONSIDER RANDOM OBJECTS TO BE 0-LENGTH DOTTED LISTS. (LIST-REVERSE 'A) SHOULD BE AN ERROR. AND SO SHOULD (REVERSE 'A).

For example:

```
(list-reverse '(a b (c d) e)) => (e (c d) b a)
```

See `reverse` (page 110), which can reverse any kind of sequence.

`revappend` *x y* [*Function*]

`(revappend` *x y*`)` is exactly the same as `(append (reverse` *x*`)` *y*`)` except that it is more efficient. Both *x* and *y* should be lists. The argument *x* is copied, not destroyed. Compare this with `nreconc` (page 128), which destroys its first argument.

Is this a list function or a sequence functio

`list-nreverse` *list* [*Function*]

`nreverse` reverses its argument, which should be a list. The argument is destroyed by `rplacd`'s all through the list (see `reverse`, which creates a new list rather than destroying its argument). If the *list* is dotted, the non-`()` atom at the end is discarded; `nreverse` always produces a list ending in `()`.

For example:

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

At this point the precise value of x is implementation-dependent.

See `nreverse` (page 110), which can destructively reverse any kind of sequence.

`nconc &rest` *lists* [*Function*]

`nconc` takes lists as arguments. It returns a list which is the arguments concatenated together. The arguments are changed, rather than copied. (Compare this with `append` (page 126), which copies arguments rather than destroying them.)

For example:

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

In the Lisp machine there are nconc operations possible on sequences other than lists. Should this keep the name nconc (rather than list-nconc) It's ok with me if it does.

Note, in the example,that the value of x is now different, since its last cons has been `rplacd`'d to the value of `y`. If one were then to evaluate `(nconc x y)` again, it would yield a piece of "circular" list structure, whose printed representation would be `(a b c d e f d e f d e f ...)`, repeating forever.

`nreconc` *x y* [*Function*]

`(nreconc` *x y*`)` is exactly the same as `(nconc (nreverse` *x*`)` *y*`)` except that it is more efficient. Both *x* and *y* should be lists. The argument *x* is destroyed. Compare this with `revappend` (page 127).

`push` *item place* [*Macro*]

place should be a reference to a cell containing a list; *item* may be any LISP object. Usually *place* is the name of a variable. *item* is consed onto the front of the list, and the augmented list is stored back into *place*. If the list held in *place* is viewed as a push-down stack, then `push` pushes an element onto the top of the stack.

The form

```
(push (hairy-function x y z) variable)
```

replaces the commonly-used construct

```
(setq variable (cons (hairy-function x y z) variable))
```

and is intended to be more explicit and esthetic.

In general,

`(push` *item place*`) ==> (setf` *place* `(cons` *item place*`))`

(See `setf` (page SETF-FUN).)

`pop` *place* [*Macro*]

place should be a reference to a cell containing a list. Usually *place* is the name of a variable. The result of `pop` is the `car` of the contents of *place*, and as a side-effect the `cdr` of the contents is stored back into *place*. If the list held in *place* is viewed as a push-down stack, then `pop` pops an element from the top of the stack and returns it.

For example:

Lisp machine pop allows an optional second argument where to store the car. If this has been decided to be flushed, put a compatibility note

```
(setq stack '(a b c))
(pop stack) => a
stack => (b c)                  ;The stack was popped.
```

In general,

```
(pop place) ==> (prog1 (car place) (setf place (cdr place)))
```

(See `setf` (page SETF-FUN).)

`butlast` *list* [*Function*]

This creates and returns a list with the same elements as *list*, excepting the last element. The argument is not destroyed. If the argument is `()`, then `()` is returned.

For example:

```
(butlast '(a b c d)) => (a b c)
(butlast '((a b) (c d)) => ((a b))
(butlast '(a)) => ()
(butlast nil) => ()
```

The name is from the phrase "all elements but the last".

should take &optional (n 1)

`nbutlast` *list* [*Function*]

This is the destructive version of `butlast`; it changes the *cdr* of the second-to-last cons of the list to `()`. If there is no second-to-last cons (that is, if the list has fewer than two elements) it returns `()`, and the argument is not modified. (Therefore one normally writes `(setq a (nbutlast a))` rather than simply `(nbutlast a)`.)

For example:

```
(setq foo '(a b c d))
(nbutlast foo) => (a b c)
foo => (a b c)
(nbutlast '(a)) => ()
(nbutlast '()) => ()
```

??? Query: Do we really want `firstn`, `lastn`, and `ldiff`, given the existence of `sublist` (page 138)?

RMS THREW THESE IN BECAUSE HE SAW THEM IN THE INTERLISP MANUAL AND HE THOUGHT THEY MIGHT BE GOOD. I BELIEVE I ONCE SAW A USE OF LDIFF (IT WAS BY RMS).

sublist is orthogonal to ldiff. Note that firstn + lastn beha differently from sub if you ask for more elements there are. I think they importa how

`firstn` *n list* [*Function*]

`firstn` returns a list of length *n*, whose elements are the first *n* elements of `list`. If *list* is fewer than *n* elements long, the remaining elements of the returned list will be `()`. The argument *list* is not destroyed.

For example:

```
(firstn 2 '(a b c d)) => (a b)
(firstn 0 '(a b c d)) => ()
(firstn 6 '(a b c d)) => (a b c d () ())
```

`lastn` *n list* [*Function*]

`lastn` returns a list of length *n*, whose elements are the last *n* elements of `list`. If *list* is fewer than *n* elements long, the leading elements of the returned list will be `()`. The argument *list* is not destroyed, nor is it copied.

For example:

```
(lastn 2 '(a b c d)) => (c d)
(lastn 0 '(a b c d)) => ()
(lastn 6 '(a b c d)) => (() () a b c d)
```

`ldiff` *list sublist* [*Function*]

list should be a list, and *sublist* should be a sublist of *list*, i.e. one of the conses that make up *list*. `ldiff` (meaning List Difference) will return a new list, whose elements are those elements of *list* that appear before *sublist*. If *sublist* is not a tail of *list*, then a copy of *list* is returned. The argument *list* is not destroyed.

For example:

```
(setq x '(a b c d e))
(setq y (cdddr x)) => (d e)
(ldiff x y) => (a b c)
```

but

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

since the sublist was not `eq` to any part of the list.

`list-to-vector` *list* [*Function*]

`list-to-vector` constructs a vector of the same length as *list* and with the same corresponding elements, and returns the new vector. The inverse of this operation is `vector-to-list` (page VECTOR-TO-LIST-FUN).

`list-to-string` *list* [*Function*]

`list-to-string` constructs a string of the same length as *list* and with the same corresponding elements (which must all be characters satisfying `string-charp` (page 98)), and returns the new string. The inverse of this operation is `string-to-list` (page 155).

13.3. Alteration of List Structure

The functions `rplaca` and `rplacd` are used to make alterations in already-existing list structure; that is, to change the cars and cdrs of existing conses.

The structure is not copied but is physically altered; hence caution should be exercised when using these functions, as strange side-effects can occur if portions of list structure become shared unbeknownst to the programmer. The `nconc` (page 128), `nreverse` (page 110), `nreconc` (page 128), and `nbutlast` (page 129) functions already described, and the `delete` (page 134) family described later, have the same property. However, they are normally not used for this side-effect; rather, the list-structure modification is purely for

I GREATLY DISAGREE WITH THIS PHILOSOPHY, BUT WE HAVE ENOUGH TO ARGUE ABOUT AT THE MOMENT...

efficiency and compatible non-modifying functions are provided.

`rplaca` *x y* [*Function*]

(`rplaca` *x y*) changes the car of *x* to *y* and returns (the modified) *x*. *x* should be a cons, but *y* may be any Lisp object.

must

For example:

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

`rplacd` *x y* [*Function*]

(`rplacd` *x y*) changes the cdr of *x* to *y* and returns (the modified) *x*. *x* should be a cons, but *y* may be any Lisp object.

For example:

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

Compatibility note: In COMMON LISP, as in MACLISP and Lisp Machine LISP, `rplacd` can not be used to set the property list of a symbol. The `setplist` (page SETPLIST-FUN) function is provided for this purpose.

13.4. Substitution of Expressions

A number of functions are provided for performing substitutions within a tree. All take a tree and a description of old sub-expressions to be replaced by new ones. The functions form a semi-regular collection, according to these properties:

- Whether comparison of items is by `eq` or `equal`.
- Whether substitution is specified by two arguments or by an association list.
- Whether the tree is copied or modified.

Allow user to supply predicate. I suggest making it an optional argument which uniformly defaults either to eq or eql. Then flush substq, nsubstq

These properties may be summarized as follows:

	Accepts two arguments, old and new		Accepts an association list
	Uses `equal`	Uses `eq`	Uses `eq`
Copies	`subst`	`substq`	`sublis`
Modifies	`nsubst`	`nsubstq`	`nsublis`

`subst` *new old tree* [*Function*]

(`subst` *new old tree*) substitutes *new* for all occurrences of *old* in *tree*, and returns the modified copy of *tree*. The original *tree* is unchanged, as `subst` recursively copies all of *tree* replacing elements `equal` to *old* as it goes.

For example:

In NIL subst is a sequence operation, I think. There doesn't seem to be any predefined function for substitution in sequences in this book.

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

This function is not "destructive"; that is, it does not change the *car* or *cdr* of any already-existing list structure.

Make it clear whether it substitutes in cdrs as well as cars

`(subst () () x)` is an idiom once frequently used to copy all the conses in a tree, but the `copytree` (page 127) function is more appropriate to the task.

(the Lisp machine one currently does. substitute in cd probably a bu

`nsubst` *new old tree* [*Function*]

`nsubst` is a destructive version of `subst`. The list structure of *tree* is altered by replacing each occurrence of *old* with *new*. `equal` is used to decide whether a part of *tree* is the same as *old*.

`substq` *new old tree* [*Function*]

`substq` is just like `subst`, except that `eq`, rather than `equal`, is used to decide whether a part of *tree* is the same as *old*.

There should also be a version of SUBST that copies only what it needs to (like SUBST). Some Lisp machine programs define this for themselve

`nsubstq` *new old tree* [*Function*]

`nsubstq` is a destructive version of `substq`. `nsubstq` is just like `nsubst`, except that `eq`, rather than `equal`, is used to decide whether a part of *tree* is the same as *old*.

`sublis` *alist tree* [*Function*]

`sublis` makes substitutions for symbols in a tree (a structure of conses). The first argument to `sublis` is an association list. The *car* of each a-list entry should be a symbol. The second argument is the tree in which substitutions are to be made. `sublis` looks at all symbols in the tree; if a symbol appears as a key in the association list occurrences of it are replaced by the object it is associated with. The argument is not modified; new conses are created where necessary and only where necessary, so the newly created structure shares as much of its substructure as possible with the old. For example, if no substitutions are made, the result is `eq` to the old tree.

For example:

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

`nsublis` *alist tree* [*Function*]

`nsublis` is like `sublis` but changes the original list structure instead of copying.

13.5. Using Lists as Sets

COMMON LISP includes functions which allow a list of items to be treated as a *set*. Some of the functions usefully allow the set to be ordered; others specifically support unordered sets. There are functions to add,

?

remove, and search for items in a list, based on various criteria. There are also set union, intersection, and difference functions.

Many of the functions described here form a regular pattern according to two criteria:

- Whether elements are compared for equality by `equal`, `eq`, or some other specified predicate of one or two arguments.
- Whether the operation is destructive or not.

As a general rule, a function which uses `equal` is named by an English word; the corresponding function which uses an arbitrary two-argument predicate is named by some short prefix of that word; the function which uses `eq` is named by that prefix plus "`q`"; the function which uses a one-argument predicate is named by the prefix plus "`-if`"; and the function which takes a one-argument predicate but inverts its sense is named by the prefix plus "`-if-not`".

As another general rule, the destructive version of a function is named by prefixing "`n`" to the name of the version which is not destructive. An exception (for historical reasons) to this rule is the pair `delete` (page 134) and `remove` (page 113)/`list-remove` (page 139).

`member` *item list* [*Function*]
`memq` *item list* [*Function*]
`mem` *predicate item list* [*Function*]
`mem-if` *predicate list* [*Function*]
`mem-if-not` *predicate list* [*Function*]

(`member` *item list*) returns `()` if *item* is not one of the elements of *list*. Otherwise, it returns the tail of *list* beginning with the first occurrence of *item*. The comparison is made by `equal`. *list* is searched on the top level only. Because `member` returns `()` if it doesn't find anything, and something non-`()` if it finds something, it is often used as a predicate.

For example: member

```
(memq 'snerd '(a b c d)) => ()
(memq 'a '(g (a y) c a d e a f)) => (a d e a f)
```

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

`memq` is like `member`, except `eq` is used to compare the *item* to the list element, instead of `equal`.

`mem` is like `member`, except *predicate* is used to compare the *item* to the list element, instead of `equal`.

`mem-if` is like `member`, except that *predicate*, a function of one argument, is used to test elements of *list*.

`mem-if-not` is like `mem-if`, except that the sense of *predicate* is inverted; that is, a test succeeds if `predicate` returns `()`.

See also `position` (page 114) and `list-position` (page 139).

+ some + friends
(for mem-if, mem-if-no

`tailp` *sublist list* [*Function*]

Returns `t` if *sublist* is a sublist of *list* (i.e. one of the conses that makes up *list*). Otherwise returns `()`. Another way to look at this is that `tailp` returns `t` if `(nthcdr` *n list*`)` is *sublist*, for some value of *n*. See `ldiff` (page 130).

`delete` *item list* `&optional` *n* [*Function*]
`delq` *item list* `&optional` *n* [*Function*]
`del` *predicate item list* `&optional` *n* [*Function*]
`del-if` *predicate list* `&optional` *n* [*Function*]
`del-if-not` *predicate list* `&optional` *n* [*Function*]

`(delete` *item list*`)` returns the *list* with all top-level occurrences of *item* removed. `equal` is used to compare *item* to elements of the list. The operation may be destructive; the argument *list* may be actually modified (`rplacd`'ed) when instances of *item* are spliced out. `delete` should be used for value, not for effect. That is, use

```
(setq a (delete x a))
```

rather than

```
(delete x a)
```

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

If the optional argument *n* is provided, it should be a non-negative integer; it specifies an upper limit on the number of deletions. `(delete` *item list n*`)` is like `(delq` *item list*`)` except only the first *n* instances of *item* are deleted. *n* is allowed to be zero, in which case no elements are deleted. If *n* is greater than the number of occurrences of *item* in the list, all occurrences of *item* in the list will be deleted.

For example:

```
(delete 'a '(b a c (a b) d a e)) => (b c (a b) d e)
(delete 'a '(b a c (a b) d a e) 1) => (b c (a b) d a e)
```

`delq` is like `delete`, except `eq` is used to compare the *item* to the list element, instead of `equal`.

`del` is like `delete`, except *predicate* is used to compare the *item* to the list element, instead of `equal`.

`del-if` is like `delete`, except that *predicate*, a function of one argument, is used to test elements of *list*.

`del-if-not` is like `del-if`, except that the sense of *predicate* is inverted; that is, a test succeeds if `predicate` returns `()`.

For non-destructive deletion, use `remove` (page 113) or `list-remove` (page 139).

`adjoin` *item list* [*Function*]
`adjq` *item list* [*Function*]
`adj` *predicate item list* [*Function*]

`adjoin` is used to add an element to a set, provided that it is not already a member. `equal` is used to compare *item* to elements of *list*.

`(adjoin` *item list*`)`
means the same as
`(if (member` *item list*`)` *list* `(cons` *item list*`))`

There should be a
PUSH version of this
– sometimes called
PUSHE or ADD2L (lousy
names

`adjq` is like `adjoin`, except `eq` is used to compare the *item* to the list element, instead of `equal`.

`adj` is like `adjoin`, except *predicate* is used to compare the *item* to the list element, instead of `equal`.

For these, I suggest
having only one version
and making the predicate
a required first argument. Sort of like REDUCE.

`union` `&rest` *lists* [*Function*]
`unionq` `&rest` *lists* [*Function*]
`unite` *predicate* `&rest` *lists* [*Function*]

`union` takes any number of lists and returns a new list containing everything that is an element of any of the *lists*. If there is a duplication (as determined by `equal`) between two lists, only one of the duplicate instances will be in the result. If any of the arguments has duplicate entries, the redundant entries may or may not appear in the result.

You also need a wa
to take one list and
eliminate any duplicate
in it (or define union of
one arg to d
that as an
exception)

For example:

```
(union '(a b c) '(f a d)) => (a b c f d)
(union) => ()
```

If `union` is given no arguments, then `()` is returned, for `()` is the identity of the operation.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The implementation is therefore free to use any of a variety of strategies.

`uniteq` is like `union`, except `eq` is used to compare elements of the *lists*, instead of `equal`.

`unite` is like `union`, except *predicate* is used to compare elements of the *lists*, instead of `equal`.

`nunion` `&rest` *lists* [*Function*]
`nunionq` `&rest` *lists* [*Function*]
`nunite` *predicate* `&rest` *lists* [*Function*]

`nunion` is the destructive version of `union`. *nunion* takes any number of lists and returns a new list containing everything that is an element of any of the *lists*. If there is a duplication (as determined by `equal`) between two lists, only one of the duplicate instances will be in the result. If any of the arguments has duplicate entries, the redundant entries may or may not appear in the result. Any of the argument lists may be cannibalized to construct the result.

If `nunion` is given no arguments, then `()` is returned, for `()` is the identity of the operation.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The implementation is therefore free to use any of a variety of strategies.

`nuniteq` is like `nunion`, except `eq` is used to compare elements of the *lists*, instead of `equal`.

`nunite` is like `nunion`, except *predicate* is used to compare elements of the *lists*, instead of `equal`.

`intersection` *firstlist* `&rest` *otherlists* [*Function*]
`intersectq` *firstlist* `&rest` *otherlists* [*Function*]
`intersect` *predicate firstlist* `&rest` *otherlists* [*Function*]

`intersection` takes any number of lists and returns a new list containing everything that is an element of *firstlist* and also of the *otherlists*. If *firstlist* has duplicate entries, the redundant entries may or may not appear in the result.

For example:

```
(intersection '(a b c) '(f a d)) => (a)
```

Unfortunately, the identity for the `intersection` operation is the entire universe. Because there is no defined representation for that, `intersection` requires at least one argument.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The implementation is therefore free to use any of a variety of strategies.

`intersectq` is like `intersection`, except `eq` is used to compare elements of the *lists*, instead of `equal`.

`intersect` is like `intersection`, except *predicate* is used to compare elements of the *lists*, instead of `equal`.

`nintersection` *firstlist* `&rest` *otherlists* [*Function*]
`nintersectq` *firstlist* `&rest` *otherlists* [*Function*]
`nintersect` *predicate firstlist* `&rest` *otherlists* [*Function*]

`nintersection` is the destructive version of `intersection`. Only *firstlist* may be destroyed, however. `nintersection` takes any number of lists and returns a new list containing everything that is an element of *firstlist* and also of the *otherlists*. If *firstlist* has duplicate entries, the redundant entries may or may not appear in the result.

Unfortunately, the identity for the `nintersection` operation is the entire universe. Because there is no defined representation for that, `nintersection` requires at least one argument.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The implementation is therefore free to use any of a variety of strategies.

Could we see the motivation for this (a
contrasted with union
in a note? I assume
it's that intersection

makes things smaller
while union makes them
bigger. By the way,
are the "n—" versions
"guaranteed" not to cons?
This isn't discussed.

`nintersectq` is like `nintersection`, except `eq` is used to compare elements of the *lists*, instead of `equal`.

`nintersect` is like `nintersection`, except *predicate* is used to compare elements of the *lists*, instead of `equal`.

`setdifference` *list1 list2* [*Function*]
`setdiffq` *list1 list2* [*Function*]
`setdiff` *predicate list1 list2* [*Function*]

`setdifference` returns a list of elements of *list1* which do not appear in *list2*. This operation is not destructive. `equal` is used to compare elements of the lists.

`setdiffq` is like `setdifference`, except `eq` is used to compare elements of the lists, instead of `equal`.

`setdiff` is like `setdifference`, except *predicate* is used to compare elements of the lists, instead of `equal`.

`nsetdifference` *list1 list2* [*Function*]
`nsetdiffq` *list1 list2* [*Function*]
`nsetdiff` *predicate list1 list2* [*Function*]

`nsetdifference` is the destructive version of `setdifference`. `nsetdifference` returns a list of elements of *list1* which do not appear in *list2*. This operation may destroy *list1*. `equal` is used to compare elements of the lists.

`nsetdiffq` is like `nsetdifference`, except `eq` is used to compare elements of the lists, instead of `equal`.

`nsetdiff` is like `nsetdifference`, except *predicate* is used to compare elements of the lists, instead of `equal`.

`set-exclusive-or` *list1 list2* [*Function*]
`setxorq` *list1 list2* [*Function*]
`setxor` *predicate list1 list2* [*Function*]

`set-exclusive-or` returns a list of elements which appear in exactly one of *list1* and *list2*. This operation is not destructive. `equal` is used to compare elements of the lists.

`setxorq` is like `set-exclusive-or`, except `eq` is used to compare elements of the lists, instead of `equal`.

`setxor` is like `set-exclusive-or`, except *predicate* is used to compare elements of the lists, instead of `equal`.

`nset-exclusive-or` *list1 list2* [*Function*]
`nsetxorq` *list1 list2* [*Function*]
`nsetxor` *predicate list1 list2* [*Function*]

`nset-exclusive-or` is the destructive version of `set-exclusive-or`. `nset-exclusive-or` returns a list of elements which appear in exactly one of *list1* and *list2*. Both lists may be destroyed in producing the result. `equal` is used to compare elements of the lists.

`nsetxorq` is like `nset-exclusive-or`, except `eq` is used to compare elements of the lists, instead of `equal`.

`nsetxor` is like `nset-exclusive-or`, except *predicate* is used to compare elements of the lists, instead of `equal`.

13.6. List-Specific Sequence Operations

The functions in this section are equivalent in operation to the corresponding generic sequence functions, but require sequence arguments to be lists. Such lists may be terminated by atoms other than `()`, but as a rule such atoms are ignored other than as list terminators. Note that non-list sequences are atoms and will terminate lists.

`list-elt` *list index* [*Function*]

The element of the *list* specified by the integer *index* is returned. The *index* must be non-negative and less than the length of the list. See `elt` (page 108).

This differs from `nth` (page 125) in that `nth` allows the index to be larger than the length of the list. Note also that `nth` takes its arguments in the reverse order.

`list-setelt` *list index newvalue* [*Function*]

The LISP object *newvalue* is stored into the component of the *list* specified by the integer *index*. The *index* must be non-negative and less than the length of the list. See `setelt` (page 108).

`sublist` *list start* `&optional` *end* [*Function*]
`list-fill` *list item* `&optional` *start end* [*Function*]
`list-replace` *target-list source-list* `&optional` *target-start source-start target-end source-end* [*Function*]
`list-concat` `&rest` *lists* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `subseq` (page 108), `fill` (page 109), `replace` (page 110), and `concat` (page 110).

Note especially that `list-concat`, like `concat` and unlike `append` (page 126), copies all of its arguments, rather than letting the result share the last argument as its tail.

`list-reduce` *function list* `&optional` *start-value* [*Function*]
`list-left-reduce` *function list* `&optional` *start-value* [*Function*]
`list-right-reduce` *function list* `&optional` *start-value* [*Function*]
`list-map` *function* `&rest` *lists* [*Function*]
`list-some` *predicate* `&rest` *lists* [*Function*]
`list-every` *predicate* `&rest` *lists* [*Function*]
`list-notany` *predicate* `&rest` *lists* [*Function*]
`list-notevery` *predicate* `&rest` *lists* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `reduce` (page 111), `left-reduce` (page 111), `right-reduce` (page 111), `map` (page 112), `some` (page 112), `every` (page 112), `notany` (page 112), `notevery` (page 112).

Is list-map identic
to mapcar?

`list-remove` *item list* `&optional` *count* [*Function*]
`list-remq` *item list* `&optional` *count* [*Function*]
`list-rem` *predicate item list* `&optional` *count* [*Function*]
`list-rem-if` *predicate list* `&optional` *count* [*Function*]
`list-rem-if-not` *predicate list* `&optional` *count* [*Function*]
`list-remove-from-end` *item list* `&optional` *count* [*Function*]
`list-remq-from-end` *item list* `&optional` *count* [*Function*]
`list-rem-from-end` *predicate item list* `&optional` *count* [*Function*]
`list-rem-from-end-if` *predicate list* `&optional` *count* [*Function*]
`list-rem-from-end-if-not` *predicate list* `&optional` *count* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `remove` (page 113).

`list-position` *item list* `&optional` *start end* [*Function*]
`list-posq` *item list* `&optional` *start end* [*Function*]
`list-pos` *predicate item list* `&optional` *start end* [*Function*]
`list-pos-if` *predicate list* `&optional` *start end* [*Function*]
`list-pos-if-not` *predicate list* `&optional` *start end* [*Function*]
`list-position-from-end` *item list* `&optional` *start end* [*Function*]
`list-posq-from-end` *item list* `&optional` *start end* [*Function*]
`list-pos-from-end` *predicate item list* `&optional` *start end* [*Function*]
`list-pos-from-end-if` *predicate list* `&optional` *start end* [*Function*]
`list-pos-from-end-if-not` *predicate list* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `position` (page 114).

`list-scan-over` *item list* `&optional` *start end* [*Function*]
`list-scanq` *item list* `&optional` *start end* [*Function*]
`list-scan` *predicate item list* `&optional` *start end* [*Function*]
`list-scan-if` *predicate list* `&optional` *start end* [*Function*]
`list-scan-if-not` *predicate list* `&optional` *start end* [*Function*]
`list-scan-over-from-end` *item list* `&optional` *start end* [*Function*]
`list-scanq-from-end` *item list* `&optional` *start end* [*Function*]
`list-scan-from-end` *predicate item list* `&optional` *start end* [*Function*]
`list-scan-from-end-if` *predicate list* `&optional` *start end* [*Function*]
`list-scan-from-end-if-not` *predicate list* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `scan-over` (page 114).

`list-count` *item list* `&optional` *start end* [*Function*]
`list-cntq` *item list* `&optional` *start end* [*Function*]
`list-cnt` *predicate item list* `&optional` *start end* [*Function*]
`list-cnt-if` *predicate list* `&optional` *start end* [*Function*]
`list-cnt-if-not` *predicate list* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `count` (page 115).

`list-mismatch` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-mismatq` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-mismat` *predicate list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-mismatch-from-end` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-mismatq-from-end` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-mismat-from-end` *predicate list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `mismatch` (page 116).

`list-maxprefix` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-maxprefq` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-maxpref` *predicate list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-maxsuffix` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-maxsuffq` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-maxsuff` *predicate list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `maxprefix` (page 117).

`list-search` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-srchq` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-srch` *predicate list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-search-from-end` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-srchq-from-end` *list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]
`list-srch-from-end` *predicate list1 list2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `search` (page 118).

`list-sort` *list predicate* [*Function*]
`list-sortcar` *list predicate* [*Function*]
`list-sortslot` *list key-function predicate* [*Function*].

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `sort` (page 118).

`list-merge` *list1 list2 predicate* [*Function*]
`list-mergecar` *list1 list2 predicate* [*Function*]
`list-mergeslot` *list1 list2 key-function predicate* [*Function*]
`list-nmerge` *list1 list2 predicate* [*Function*]
`list-nmergecar` *list1 list2 predicate* [*Function*]
`list-nmergeslot` *list1 list2 key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`list-`", except that sequence arguments must be lists. See `merge` (page 120) and `nmerge` (page 121).

13.7. Association Lists

An *association list*, or *a-list*, is a data structure used very frequently in LISP. An a-list is a list of pairs (conses); each pair is an association. The *car* of a pair is called the *key*, and the *cdr* is called the *datum*.

An advantage of the a-list representation is that an a-list can be incrementally augmented simply by adding new entries to the front. Moreover, because the searching functions such as `assoc` search the a-list in order, new entries can "shadow" old entries. If an a-list is viewed as a mapping from keys to data, then the mapping can be not only augmented but also altered in a non-destructive manner by adding new entries to the front of the a-list.

Sometimes an a-list represents a bijective mapping, and it is desirable to retrieve a key given a datum. For this purpose "reverse" forms of the a-list functions are provided.

It is permissible to let `()` be an element of an a-list in place of a pair.

`acons` *key datum a-list* [*Function*]

`acons` constructs a new association list by adding the pair `(`*key* `.` *datum*`)` to the old *a-list*.

```
(acons x y a) <=> (cons (cons x y) a)
```

`pairlis` *keys data* `&optional` *a-list* [*Function*]

`pairlis` takes two lists and makes an association list which associates elements of the first list to corresponding elements of the second list. It is an error if the two lists *keys* and *data* are not of the same length. If the optional argument *a-list* is provided, then the new pairs are added to the front of it.

For example:

```
(pairlis '(beef clams kitty) '(roast fried yu-shiang))
   => ((beef . roast) (clams . fried) (kitty . yu-shiang))
(pairlis '(one two) '(1 2) '((three . 3) (four . 19)))
   => ((one . 1) (two . 2) (three . 3) (four . 19))
```

The remaining association-list functions form a regular collection according to three independent criteria:

- The type of operation is indicated by a prefix to the function name:

no prefix	Search, returning an association pair.
`mem`	Search, returning a tail of the a-list.
`pos`	Search, returning a numerical index into the a-list.
`del`	Destructive deletion.

The prefixes indicate that the operations are related to the functions `member` (page 133), `position` (page 114), and `delete` (page 134).

- If the function treats the a-list normally (the *car* of each association pair is treated as the key), then no infix is written. If the function treats the a-list as a reverse mapping (the *cdr* of each association pair is treated as the key), then the letter "`r`" is written.

- The suffix names the a-list operation and indicates the testing criterion:

`assoc`	Compare against an item using `equal`.
`assq`	Compare against an item using `eq`.
`ass`	Compare against an item using a user-specified predicate.
`ass-if`	Use a single-argument user-specified predicate.
`ass-if-not`	Invert a single-argument user-specified predicate.

Again, too many functions.

Thus, for example, the function `posrassq` would perform a search, returning the numerical position, treating the a-list in reverse form, and using `eq` to test the keys.

`assoc` *item a-list* [*Function*]
`assq` *item a-list* [*Function*]
`ass` *predicate item a-list* [*Function*]
`ass-if` *predicate a-list* [*Function*]
`ass-if-not` *predicate a-list* [*Function*]

`(assoc` *item alist*`)` looks up *item* in the association list *a-list*. The value is the first pair in the a-list whose `car` is `equal` to *x*, or `()` if there is none such.

For example:

```
(assoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
        =>  (r . x)
(assoc 'goo '((foo . bar) (zoo . goo))) => ()
(assoc '2 '((1 a b c) (2 b c d) (-7 x y z))) => (2 b c d)
```

It is possible to `rplacd` the result of `assoc` *provided* that it is not `()`, if your intention is to "update" the "table" that was `assoc`'s second argument. (However, it is often better to update an a-list by adding new pairs to the front, rather than altering old pairs.)

For example:

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

A typical trick is to say `(cdr (assoc x y))`. Because the *cdr* of `()` is guaranteed to be `()`, this yields `()` if no pair is found *or* if a pair is found whose *cdr* is `()`. This is useful if `()` serves its usual role as a "default value".

`assq` is like `assoc`, except `eq` is used to compare the *item* to each key, instead of `equal`.

`ass` is like `assoc`, except *predicate* is used to compare the *item* to each key, instead of `equal`.

`ass-if` is like `assoc`, except that *predicate*, a function of one argument, is used to test keys of *a-list*.

`ass-if-not` is like `ass-if`, except that the sense of *predicate* is inverted; that is, a test succeeds if `predicate` returns `()`.

`rassoc` *item a-list* [*Function*]
`rassq` *item a-list* [*Function*]
`rass` *predicate item a-list* [*Function*]
`rass-if` *predicate a-list* [*Function*]
`rass-if-not` *predicate a-list* [*Function*]

`rassoc` is the reverse form of `assoc`; it compares *item* to the *cdr* of each successive pair in *a-list*, rather than to the *car*. Similarly, `rassq` is the reverse form of `assq`, and so on.

For example:

```
(rassoc 'a '((a . b) (b . c) (c . a) (z . a))) => (c . a)
```

`memassoc` *item a-list* [*Function*]
`memassq` *item a-list* [*Function*]
`memass` *predicate item a-list* [*Function*]
`memass-if` *predicate a-list* [*Function*]
`memass-if-not` *predicate a-list* [*Function*]

`memassoc` is a synthesis of `assoc` (page 143) and `member` (page 133).

(`memassoc` *item alist*) looks up *item* in the association list *a-list*. The value is the portion of the a-list whose first pair is the first pair in *a-list* whose `car` is `equal` to *x*, or `()` if there is none such. Thus `memassoc` performs its search like `assoc`, but returns a value like `member`.

For example:

```
(memassoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
   => ((r . x) (s . y) (r . z))
(memassoc 'goo '((foo . bar) (zoo . goo))) => ()
(memassoc '2 '((1 a b c) (2 b c d) (-7 x y z)))
   => ((2 b c d) (-7 x y z))
```

`memassq` is like `memassoc`, except `eq` is used to compare the *item* to each key, instead of `equal`.

`memass` is like `memassoc`, except *predicate* is used to compare the *item* to each key, instead of `equal`.

`memass-if` is like `memassoc`, except that *predicate*, a function of one argument, is used to test keys of *a-list*.

`memass-if-not` is like `memass-if`, except that the sense of *predicate* is inverted; that is, a test succeeds if `predicate` returns `()`.

`memrassoc` *item a-list* [*Function*]
`memrassq` *item a-list* [*Function*]
`memrass` *predicate item a-list* [*Function*]
`memrass-if` *predicate a-list* [*Function*]
`memrass-if-not` *predicate a-list* [*Function*]

`memrassoc` is the reverse form of `memassoc`; it compares *item* to the *cdr* of each successive pair in *a-list*, rather than to the *car*. Similarly, `memrassq` is the reverse form of `memassq`, and so on.

For example:

```
(memrassoc 'a '((a . b) (b . c) (c . a) (z . a) (p . q)))
   => ((c . a) (z . a) (p . q))
```

`posassoc` *item a-list* [*Function*]
`posassq` *item a-list* [*Function*]
`posass` *predicate item a-list* [*Function*]
`posass-if` *predicate a-list* [*Function*]
`posass-if-not` *predicate a-list* [*Function*]

`posassoc` is a synthesis of `assoc` (page 143) and `position` (page 114).

(`posassoc` *item alist*) looks up *item* in the association list *a-list*. The value is the zero-origin numerical position of the first pair in the a-list whose `car` is `equal` to *x*, or `()` if there is none such.

For example:

```
(posassoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
   => 2
(posassoc 'goo '((foo . bar) (zoo . goo))) => ()
(posassoc '2 '((1 a b c) (2 b c d) (-7 x y z))) => 1
```

`posassq` is like `posassoc`, except `eq` is used to compare the *item* to each key, instead of `equal`.

`posass` is like `posassoc`, except *predicate* is used to compare the *item* to each key, instead of `equal`.

`posass-if` is like `posassoc`, except that *predicate*, a function of one argument, is used to test keys of *a-list*.

`posass-if-not` is like `posass-if`, except that the sense of *predicate* is inverted; that is, a test succeeds if `predicate` returns `()`.

`posrassoc` *item a-list* [*Function*]
`posrassq` *item a-list* [*Function*]
`posrass` *predicate item a-list* [*Function*]
`posrass-if` *predicate a-list* [*Function*]
`posrass-if-not` *predicate a-list* [*Function*]

`posrassoc` is the reverse form of `posassoc`; it compares *item* to the *cdr* of each successive pair in *a-list*, rather than to the *car*. Similarly, `posrassq` is the reverse form of `posassq`, and so on.

For example:

```
(posrassoc 'a '((a . b) (b . c) (c . a) (z . a))) => 2
```

`delassoc` *item a-list* `&optional` *n* [*Function*]
`delassq` *item a-list* `&optional` *n* [*Function*]
`delass` *predicate item a-list* `&optional` *n* [*Function*]
`delass-if` *predicate a-list* `&optional` *n* [*Function*]
`delass-if-not` *predicate a-list* `&optional` *n* [*Function*]

`delassoc` is a synthesis of `assoc` (page 143) and `delete` (page 134).

(`delassoc` *item alist*) looks up *item* in the association list *a-list*. Any and all pairs to whose key *item* is `equal` are destructively spliced out of *a-list*. The value is the modified *a-list*.

For example:

```
(delassoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
   => ((a . b) (c . d) (s . y))

(delassoc 'goo '((foo . bar) (zoo . goo)))
   => ((foo . bar) (zoo . goo))    (The argument was not modified.)

(delassoc '2 '((1 a b c) (2 b c d) (-7 x y z)))
   => ((1 a b c) (-7 x y z))
```

If the optional argument *n* is provided, it should be a non-negative integer; it specifies an upper bound on the number of pairs to be removed. (In this `delassoc` behaves exactly like `delete`.)

`delassq` is like `delassoc`, except `eq` is used to compare the *item* to each key, instead of `equal`.

`delass` is like `delassoc`, except *predicate* is used to compare the *item* to each key, instead of `equal`.

`delass-if` is like `delassoc`, except that *predicate*, a function of one argument, is used to test keys of *a-list*.

`delass-if-not` is like `delass-if`, except that the sense of *predicate* is inverted; that is, a test succeeds if `predicate` returns `()`.

`delrassoc` *item a-list* `&optional` *n* [*Function*]
`delrassq` *item a-list* `&optional` *n* [*Function*]
`delrass` *predicate item a-list* `&optional` *n* [*Function*]
`delrass-if` *predicate a-list* `&optional` *n* [*Function*]
`delrass-if-not` *predicate a-list* `&optional` *n* [*Function*]

`delrassoc` is the reverse form of `delassoc`; it compares *item* to the *cdr* of each successive pair in *a-list*, rather than to the *car*. Similarly, `delrassq` is the reverse form of `delassq`, and so on.

For example:

```
(delrassoc 'a '((a . b) (b . c) (c . a) (z . a)))
   => ((a . b) (b . c))
```

Compatibility note: The functions `sassoc` and `sassq` have been omitted. They were useless hangovers from LISP 1.5 days.

13.8. Hash Tables

A hash table is a LISP object that works something like a property list and something like an association list. Each hash table has a set of *entries*, each of which associates a particular *key* with a particular *value*. The basic functions that deal with hash tables can create entries, delete entries, and find the value that is associated with a given key. Finding the value is very fast even if there are many entries, because hashing is used; this is an important advantage of hash tables over property lists.

A given hash table can only associate one *value* with a given *key*; if you try to add a second *value* it will replace the first. Also, adding a value to a hash table is a destructive operation; the hash table is modified. By contrast, association lists can be augmented non-destructively.

Hash tables come in two kinds, the difference being whether the keys are compared with `eq` or with `equal`. In other words, there are hash tables which hash on Lisp *objects* (using `eq`) and there are hash tables which hash on abstract *S-expressions* (using `equal`).

Hash tables of the first kind are created with the function `make-hash-table`, which takes various options. New entries are added to hash tables with the `puthash` function. To look up a key and find the associated value, use `gethash`; to remove an entry, use `remhash`. Here is a simple example.

```
(setq a (make-hash-table))
(puthash 'color 'brown a)
(puthash 'name 'fred a)
(gethash 'color a) => brown
(gethash 'name a) => fred
(gethash 'pointy a) => ()
```

In this example, the symbols `color` and `name` are being used as keys, and the symbols `brown` and `fred` are being used as the associated values. The hash table has two items in it, one of which associates from `color` to `brown`, and the other of which associates from `name` to `fred`.

Keys do not have to be symbols; they can be any Lisp object. Likewise values can be any Lisp object. Hash tables are properly interfaced to the relocating garbage collector so that garbage collection will have no perceptible effect on the functionality of hash tables. ?

When a hash table is first created, it has a *size*, which is the maximum number of entries it can hold. Usually the actual capacity of the table is somewhat less, since the hashing is not perfectly collision-free. With the maximum possible bad luck, the capacity could be very much less, but this rarely happens. If so many entries are added that the capacity is exceeded, the hash table will automatically grow, and the entries will be *rehashed* (new hash values will be recomputed, and everything will be rearranged so that the fast hash lookup still works). This is transparent to the caller; it all happens automatically.

> **Compatibility note:** This hash table facility is compatible with Lisp Machine Lisp. It is similar to the hasharray facility of InterLisp, and some of the function names are the same. However, it is *not* compatible with InterLisp. The exact details and the order of arguments are designed to be consistent with the rest of MacLisp rather than with InterLisp. For instance, the order of arguments to `maphash` is different, there is no "system hash table", and there is not the InterLisp restriction that keys and values may not be `()`. Note, however, that the order of arguments to `gethash`, `puthash`, and `remhash` is not consistent with `get`, `putprop`, and `remprop`, either. This is an unfortunate result of the haphazard historical development of Lisp.

13.8.1. Hashing on EQ

This section documents the functions for `eq` hash tables, which use *objects* as keys and associate other objects with them.

make-hash-table &rest *options* [*Function*]

This creates a new hash table. The number of arguments should be even. Each pair of arguments specifies an option; the first is a keyword symbol, and the second a value for that option. Valid option keywords are:

|| no colon?

`size`	Set the initial size of the hash table, in entries, as a fixnum. The default is 64. The actual size is rounded up from the size you specify to the next "good" size. You won't necessarily be able to store this many entries into the table before it overflows and becomes bigger; but except in the case of extreme bad luck you will be able to store almost this many.
`rehash-size`	Specifies how much to increase the size of the hash table when it becomes full. This can be an integer greater than zero, which is the number of entries to add, or it can be a floating-point number greater than one, which is the ratio of the new size to the old size. The default is `1.3`, which causes the table to be made 30% bigger each time it has to grow.
`rehash-threshold`	Specifies how full the hash table can get before it must grow. This can be an integer greater than zero and less than the rehash-size (in which case it will be scaled whenever the table is grown), or it can be a floating-point number between zero and one. The default is `0.8`, which means the table is enlarged when it becomes over 80% full.

This parameter is not meaningful for some hash algorithms.

For example:

```
(make-hash-table 'rehash-size 1.5
                 'size (* number-of-widgets 43))
```

gethash *key hash-table* **&optional** *default* // [*Function*]

Find the entry in *hash-table* whose key is *key*, and return the associated value. If there is no such entry, return *default*, which is `()` if not specified.

`gethash` actually returns two values; the second is `t` if an entry was found, and `()` if no entry was found.

puthash *key value hash-table* [*Function*]

Create an entry in *hash-table* associating *key* to *value*; if there is already an entry for *key*, then replace the value of that entry with *value*. Returns *value*.

swaphash key value hash-table & optional default

remhash *key hash-table* [*Function*]

Remove any entry for *key* in *hash-table*. Returns `t` if there was an entry or `()` if there was not.

maphash *function hash-table* [*Function*]

For each entry in *hash-table*, call *function* on two arguments: the key of the entry and the value of the entry. If entries are added to or deleted from the hash table while a `maphash` is in progress, the results are unpredictable. `maphash` returns `t`.

`clrhash` *hash-table* [*Function*]

Remove all the entries from *hash-table*. Returns the hash table itself.

13.8.2. Hashing on EQUAL

Don't perpetuate this mistake! Have one set of functions, and let them behave according to type of hash tab given

This section documents the functions for `equal` hash tables, which use *S-expressions* as keys and associate objects with them. They are entirely analogous to the functions for `eq` hash tables.

`make-equal-hash-table &rest` *options* [*Function*]

This creates a new hash table of the `equal` kind. The number of arguments should be even. Each pair of arguments specifies an option; the first is a keyword symbol, and the second a value for that option. The valid option keywords are the same as for `make-hash-table` (page 148).

`gethash-equal` *key* *hash-table* `&optional` *default* [*Function*]

Find the entry in *hash-table* whose key is `equal` to *key*, and return the associated value. If there is no such entry, return *default*, which is `()` if not specified.

`gethash-equal` actually returns two values; the second is `t` if an entry was found, and `()` if no entry was found.

`puthash-equal` *key* *value* *hash-table* [*Function*]

Create an entry in *hash-table* associating *key* to *value*; if there is already an entry with a key `equal` to for *key*, then replace the value of that entry with *value*. Returns *value*.

`remhash-equal` *key* *hash-table* [*Function*]

Remove any entry with a key `equal` to *key* in *hash-table*. Returns `t` if there was an entry or `()` if there was not.

`maphash-equal` *function* *hash-table* [*Function*]

For each entry in *hash-table*, call *function* on two arguments: the key of the entry and the value of the entry. If entries are added to or deleted from the hash table while a `maphash-equal` is in progress, the results are unpredictable. `maphash-equal` returns `t`.

`clrhash-equal` *hash-table* [*Function*]

Remove all the entries from *hash-table*. Returns the hash table itself.

13.8.3. Primitive Hash Function

`sxhash` *S-expression* [*Function*]

`sxhash` computes a hash code of an S-expression, and returns it as an integer, which may be positive or negative. A property of `sxhash` is that `(equal` *x y*`)` implies `(= (sxhash` *x*`) (sxhash` *y*`))`.

Implementation note: The integer returned by `sxhash` should be a fixnum, that is, an integer with an immediate representation.

In Lisp machine, This now guaranteed to return a non-negative integer.

Chapter 14

Strings

YOU APPARENTLY ONLY SUPPORT FIXED STRINGS AND NOT VARYING STRINGS (IN THE PL/1) SENSE, THAT IS THEY CAN'T HAVE FILL POINTERS. THIS GREATLY DETRACTS FROM THEIR UTILITY!

A string is a special kind of vector whose elements are characters. In general, string operations do not work on ordinary vectors.

> **Compatibility note:** Lisp Machine Lisp implements strings as a kind of array, and allows general array operations on strings, even at the user-visible level. One consequence of this is that Lisp Machine Lisp strings can have array leaders. Common Lisp treats strings as vectors, not arrays. In Lisp Machine Lisp one uses the function `aref` (page 177) to access string elements; in Common Lisp one must use the function `char` (page 151).

> **Compatibility note:** Lisp Machine Lisp allows a fixnum to be coerced into a one-character string whose element is a character whose ASCII value is the fixnum. The net effect is that a single character can be automatically coerced to be a one-character string. It would be inconsistent with adherence to the character standard, and possibly also affect efficiency adversely in some implementations, to remain compatible with this. OK

As a rule, any string operation will accept a symbol instead of a string as an argument if the operation never modifies that argument; the print-name of the symbol is used. In this respect the string-specific sequence operations are not simply specializations of the generic versions; the generic sequence operations never accept symbols as sequences. This slight inelegance is permitted in Common Lisp in the name of pragmatic utility. Also, there is a slight non-parallism in the names of string functions. Where the suffixes `equalp` and `eql` would be more appropriate, for historical compatibility the suffixes `equal` and `=` are used instead to indicate case-insensitive and case-sensitive character comparison, respectively.

Any Lisp object may be tested for being a string by the predicate `stringp` (page 29).

14.1. String Access and Modification

`char` *string index* [*Function*]

The given *index* must be a non-negative integer less than the length of *string*. The character at position *index* of the string is returned as a character object. (This character will necessarily satisfy the predicate `string-charp` (page 98).) As with all sequences in Common Lisp, indexing is zero-origin.

For example:

```
(char "Floob-Boober-Bab-Boober-Bubs" 0) => #\F
(char "Floob-Boober-Bab-Boober-Bubs" 1) => #\l
```

See `elt` (page 108).

`rplachar` *string index newchar* [*Function*]

The argument *string* must be a string. The given *index* must be a non-negative integer less than the length of the string. The character at position *index* is altered to be *newchar*, which must be a character object which satisfies the predicate `string-charp` (page 98). `rplachar` returns *newchar* as its value. See `setelt` (page 108).

14.2. String Comparison

`string=` *string1 string2* `&optional` `(`*start1* `0)` `(`*start2* `0)` *end1 end2* [*Function*]

`string=` compares two strings, returning `t` if they are the same (corresponding characters are identical) and `()` if they are not. The function `equal` (page 31) calls `string=` if applied to two strings.

missing word

The optional arguments *start1* and *start2* are the places in the strings to start the comparison. The optional arguments *end1* and *end2* places in the strings to stop comparing; comparison stops just *before* the position specified by a limit. The *start* arguments default to zero (beginning of string), and the *end* arguments default to the lengths of the strings (end of string), so that by default the entirety of each string is examined. These arguments are provided so that substrings can be compared efficiently.

The value of `string=` is necessarily `()` if the (sub)strings being compared are of unequal length; that is, if

```
(not (= (- end1 start1) (- end2 start2)))
```

italics

is `t` then `string=` returns `()`.

For example:

```
(string= "foo" "foo") => t
(string= "foo" "Foo") => ()
(string= "foo" "bar") => ()
(string= "together" "frogs" 1 3 2 4) => t
```

`string-equal` *string1 string2* `&optional` `(`*start1* `0)` `(`*start2* `0)` *end1 end2* [*Function*]

`string-equal` is just like `string=` except that differences in case are ignored; two characters are considered to be the same if `char-equal` (page 100) is true of them.

For example:

```
(string-equal "foo" "Foo") => t
```

`string<` *string1 string2* [*Function*]
`string>` *string1 string2* [*Function*]
`string<=` *string1 string2* [*Function*]
`string>=` *string1 string2* [*Function*]
`string<>` *string1 string2* [*Function*]

should have subset args like string=

The two string arguments are compared lexicographically, and the result is `()` unless *string1* is (less than, greater than, less than or equal to, greater than or equal to, not equal to) *string2*, respectively. If the condition is satisfied, however, then the result is the index within the strings of the first character position at which the strings fail to match; put another way, the result is the length of the longest common prefix of the strings.

A string *a* is less than a string *b* if in the first position in which they differ the character of *a* is less than the corresponding character of *b* according to the function `char<` (page 100), or if string *a* is a proper prefix of string *b* (of shorter length and matching in all the characters of *a*).

`string-lessp` *string1 string2* [*Function*]
`string-greaterp` *string1 string2* [*Function*]
`string-not-lessp` *string1 string2* [*Function*]
`string-not-greaterp` *string1 string2* [*Function*]
`string-not-equal` *string1 string2* [*Function*]

Should have optional args like string-equ

These are exactly like `string<`, `string>`, `string<=`, `string>=`, and `string<>`, respectively, except that distinctions between upper-case and lower-case letters are ignored. It is if `char-lessp` (page 101) were used instead of `char<` (page 100) for comparing characters.

string-compare

14.3. String Construction and Manipulation

Should take keyword arguments like make-list, make-array

`make-string` *count* `&optional` *fill-character* [*Function*]

This returns a string of length *count*, each of whose characters has been initialized to the *fill-character*. If *fill-character* is not specified, then the string will be initialized in an implementation-dependent way.

> **Implementation note:** It may be convenient to initialize the string to null characters, or to spaces, or to garbage ("whatever was there").

`string-repeat` *string count* [*Function*]

The result of `string-repeat` is a string containing *count* copies of *string* appended together. The length of the result is therefore the product of *count* and the length of *string*. The argument *count* must be a non-negative integer.

For example:

```
(string-repeat "Baz! " 4) => "Baz! Baz! Baz! Baz! "
(string-repeat "*" 5) => "*****"
```

Note that `make-string` (page 153) can also produce a string which is a replication of a single character.

`string-trim` *character-bag string* [*Function*]
`string-left-trim` *character-bag string* [*Function*]
`string-right-trim` *character-bag string* [*Function*]

`string-trim` returns a substring (in the sense of the function `substring` (page 155)) of *string*.

with all characters in *character-bag* stripped off of the beginning and end. The function `string-left-trim` is similar, but strips characters off only the beginning; `string-right-trim` strips off only the end. The argument *character-bag* may be a list of characters or a string.

For example:

```
(string-trim '(#\Space #\Tab #\Return) " garbanzo beans
         ") => "garbanzo beans"
(string-trim " ()*" " ( *three (silly) words* ) ")
   => "three (silly) words"
(string-left-trim " ()*" " ( *three (silly) words* ) ")
   => "three (silly) words* ) "
(string-right-trim " ()*" " ( *three (silly) words* ) ")
   => " ( *three (silly) words"
```

`string-upcase` *string* [*Function*]

`string-downcase` *string* [*Function*]

`string-capitalize` *string* [*Function*]

`string-upcase` returns a copy of *string*, with all lower-case alphabetic characters replaced by the corresponding upper-case characters. More precisely, each character of the result string is produced by applying the function `char-upcase` (page 102) to the corresponding character of *string*.

`string-downcase` is similar, except that upper-case characters are converted to lower-case characters (using `char-downcase` (page 102)).

For example:

```
(string-upcase "Dr. Livingston, I presume?")
   => "DR. LIVINGSTON, I PRESUME?"
(string-downcase "Dr. Livingston, I presume?")
   => "dr. livingston, i presume?"
```

`string-capitalize` produces a copy of *string* such that every word (subsequence of case-modifiable characters delimited by non-case-modifiable characters) has its first character in upper-case and any other letters in lower-case.

For example:

```
(string-capitalize " hello ") => " Hello "
(string-capitalize
     "occlUDeD cASEmenTs FOreSTAll iNADVertent DEFenestraTION")
  => "Occluded Casements Forestall Inadvertent Defenestration"
(string-capitalize 'kludgy-hash-search) => "Kludgy-Hash-Search"
```

↑ Should be allowed to return their arg if no characters need changing.

14.4. Type Conversions on Strings

`string` *x* [*Function*]

`string` coerces *x* into a string. Most of the string functions apply this to such of their arguments as are supposed to be strings. If *x* is a string, it is returned. If *x* is a symbol, its print-name is returned. If *x* cannot be coerced to be a string, an error occurs.

To get the string representation of a number or any other LISP object, use `prinstring` (page PRINSTRING-FUN) or `format` (page 217).

`string-to-list` *string* [*Function*]
`string-to-vector` *string* [*Function*]

A list or vector is created with the same length as the argument *string*, and the elements of this new list or vector are the characters of *string*.

For example:

```
(string-to-list "stretch") => (#\s #\t #\r #\e #\t #\c #\h)
(string-to-vector "stretch") => #(#\s #\t #\r #\e #\t #\c #\h)
```

The inverse conversions may be accomplished using the functions `list-to-string` (page 130) and `vector-to-string` (page VECTOR-TO-STRING-FUN).

14.5. Sequence Functions on Strings

The functions in this section are equivalent in operation to the corresponding generic sequence functions, but require sequence arguments to be strings, and sequence elements to be string-characters. As a useful extension, any argument which is supposed to be a string but is never modified may be a symbol instead, in which case the print-name of the symbol is used.

As long as the string functions are not precisely equivalent to the generic versions, the following additional but useful incompatibility is introduced. Where a generic operation uses `equal`, the string operations use `char-equal`, even though `equal` uses `char=`. Also, because `eq` is not guaranteed to work on characters, the "q" versions of the sequence functions are not provided, being replaced by "=" versions which use `char=`.

`substring` *string start* `&optional` *end* [*Function*]
`copystring` *string* [*Function*]
`string-length` *string* [*Function*]
`string-fill` *string string-character* `&optional` *start end* [*Function*]
`string-replace` *target-vector source-vector* `&optional` *target-start source-start target-end source-end* [*Function*]
`string-reverse` *string* [*Function*]
`string-nreverse` *string* [*Function*]
`string-concat` `&rest` *strings* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not

begin with the prefix `"string-"`, except that sequence arguments must be strings or possibly symbols. See `subseq` (page 108), `copyseq` (page 109), `length` (page 109), `fill` (page 109), `replace` (page 110), `reverse` (page 110), `nreverse` (page 110), and `concat` (page 110).

> Compatibility note: In Lisp Machine Lisp, `string-reverse` and `string-nreverse` are advertised to be able to reverse a 1-dimensional array of any type; indeed, in Lisp Machine Lisp they are general array reversers, and strings are merely a special kind of array. In Common Lisp, these functions may reverse only strings.

In Common Lisp version of Lisp Machine Lisp, one uses general sequence functions to do this.

`string-reduce` *function string* `&optional` *start-value* [*Function*]
`string-left-reduce` *function string* `&optional` *start-value* [*Function*]
`string-right-reduce` *function string* `&optional` *start-value* [*Function*]
`string-map` *function* `&rest` *strings* [*Function*]
`string-some` *predicate* `&rest` *strings* [*Function*]
`string-every` *predicate* `&rest` *strings* [*Function*]
`string-notany` *predicate* `&rest` *strings* [*Function*]
`string-notevery` *predicate* `&rest` *strings* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix `"string-"`, except that sequence arguments must be strings or possibly symbols. See `reduce` (page 111), `left-reduce` (page 111), `right-reduce` (page 111), `map` (page 112), `some` (page 112), `every` (page 112), `notany` (page 112), `notevery` (page 112).

> ??? Query: Should the `reduce` functions be omitted as useless, or retained for symmetry?

RETAIN, I GUESS. BUT I
THAT THESE FUNCTIONS SHOUL
EXIST EXPLICITLY.

`string-remove` *string-character string* `&optional` *count* [*Function*]
`string-rem=` *string-character string* `&optional` *count* [*Function*]
`string-rem` *predicate string-character string* `&optional` *count* [*Function*]
`string-rem-if` *predicate string* `&optional` *count* [*Function*]
`string-rem-if-not` *predicate string* `&optional` *count* [*Function*]
`string-remove-from-end` *string-character string* `&optional` *count* [*Function*]
`string-remq-from-end` *string-character string* `&optional` *count* [*Function*]
`string-rem-from-end` *predicate string-character string* `&optional` *count* [*Function*]
`string-rem-from-end-if` *predicate string* `&optional` *count* [*Function*]
`string-rem-from-end-if-not` *predicate string* `&optional` *count* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix `"string-"`, except that sequence arguments must be strings or possibly symbols. See `remove` (page 113).

`string-position` *string-character string* `&optional` *start end* [*Function*] = string-search-char
`string-pos=` *string-character string* `&optional` *start end* [*Function*]
`string-pos` *predicate string-character string* `&optional` *start end* [*Function*]
`string-pos-if` *predicate string* `&optional` *start end* [*Function*]
`string-pos-if-not` *predicate string* `&optional` *start end* [*Function*]
`string-position-from-end` *string-character string* `&optional` *start end* [*Function*]

`string-posq-from-end` *string-character string* `&optional` *start end* [*Function*]

`string-pos-from-end` *predicate string-character string* `&optional` *start end* [*Function*]

`string-pos-from-end-if` *predicate string* `&optional` *start end* [*Function*]

`string-pos-from-end-if-not` *predicate string* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix `"string-"`, except that sequence arguments must be strings or possibly symbols. See `position` (page 114).

> **Compatibility note:** In Lisp Machine LISP, `string-position` is called `string-search-char`. The Lisp Machine LISP function `string-search-set` may be expressed in COMMON LISP as
>
> `(string-pos-if #'(lambda (x) (position x` *character-set*`))` *string*`)`
>
> for example.

`string-scan-over` *string-character string* `&optional` *start end* [*Function*] = string-search-not-char

`string-scan=` *string-character string* `&optional` *start end* [*Function*]

`string-scan` *predicate string-character string* `&optional` *start end* [*Function*]

`string-scan-if` *predicate string* `&optional` *start end* [*Function*]

`string-scan-if-not` *predicate string* `&optional` *start end* [*Function*]

`string-scan-over-from-end` *string-character string* `&optional` *start end* [*Function*]

`string-scanq-from-end` *string-character string* `&optional` *start end* [*Function*]

`string-scan-from-end` *predicate string-character string* `&optional` *start end* [*Function*]

`string-scan-from-end-if` *predicate string* `&optional` *start end* [*Function*]

`string-scan-from-end-if-not` *predicate string* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix `"string-"`, except that sequence arguments must be strings or possibly symbols. See `scan-over` (page 114).

`string-count` *string-character string* `&optional` *start end* [*Function*]

`string-cnt=` *string-character string* `&optional` *start end* [*Function*]

`string-cnt` *predicate string-character string* `&optional` *start end* [*Function*]

`string-cnt-if` *predicate string* `&optional` *start end* [*Function*]

`string-cnt-if-not` *predicate string* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix `"string-"`, except that sequence arguments must be strings or possibly symbols. See `count` (page 115).

`string-mismatch` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

`string-mismat=` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

`string-mismat` *predicate string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

`string-mismatch-from-end` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

`string-mismatq-from-end` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

`string-mismat-from-end` *predicate string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`string-`", except that sequence arguments must be strings or possibly symbols. See `mismatch` (page 116).

`string-maxprefix` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-maxpref=` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-maxpref` *predicate string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-maxsuffix` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-maxsuff=` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-maxsuff` *predicate string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`string-`", except that sequence arguments must be strings or possibly symbols. See `maxprefix` (page 117).

`string-search` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-srch=` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-srch` *predicate string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-search-from-end` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-srchq-from-end` *string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]
`string-srch-from-end` *predicate string1 string2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`string-`", except that sequence arguments must be strings or possibly symbols. See `search` (page 118).

`string-sort` *string predicate* [*Function*]
`string-sortcar` *string predicate* [*Function*]
`string-sortslot` *string key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`string-`", except that sequence arguments must be strings or possibly symbols. See `sort` (page 118).

`string-merge` *string1 string2 predicate* [*Function*]
`string-mergecar` *string1 string2 predicate* [*Function*]
`string-mergeslot` *string1 string2 key-function predicate* [*Function*]
`string-nmerge` *string1 string2 predicate* [*Function*]
`string-nmergecar` *string1 string2 predicate* [*Function*]
`string-nmergeslot` *string1 string2 key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`string-`", except that sequence arguments must be strings or possibly symbols. See `merge` (page 120) and `nmerge` (page 121).

You choose not to provide the string-search-....-set series?

THIS SERIES REALLY IS USEFUL: (STRING-SEARCH-SET '(#\SPACE #\TAB) X)

Chapter 15

Vectors

A vector is a simple one-dimensional sequence of objects. Each vector has a fixed length *n* peculiar to that vector; the elements of the vector are numbered from zero to *n*-1.

Vectors differ from lists in that it takes constant time to make a list longer (using the function `cons` (page 124)) and linear time to access an arbitrary element (using the function `nth` (page 125), for example), while for a vector this is reversed: it takes linear time to extend a vector, but accessing an element takes constant time. Hence the use of lists or vectors for a particular application should be dictated primarily by efficiency considerations.

Vectors are divided into various subtypes, depending on what class of LISP objects they are capable of containing. A vector capable of containing objects of type *type* is said to be of type `(vector` *type*`)`. A request (to the function `make-vector`) to construct a vector of type `(vector` *type1*`)` may or may not succeed, however; it may produce such a vector, or it may produce a vector whose actual type is `(vector` *type2*`)`, where *type1* is a subtype of *type2*. Each implementation will respond to such a request by using the most specific type *type2* for which it provides vectors of that concrete type.

wrong word

All implementations of COMMON LISP must provide three concrete vector types: general vectors, whose type is `(vector t)`, and which can contain any LISP object; bit-vectors, whose type is `(vector (mod 2))`, and which can contain bits (the integers `0` and `1`); and strings, whose type is `(vector string-char)`, and which can contain a certain subset of the character data type. Implementations may choose to provide other specialized concrete vector types as well; a common choice is vectors of type

`(vector (mod` *n*`))` for $n = 2^{2^j}$ for integral *j*

Vectors are a kind of sequence; most of the operations on vectors are merely specialized versions of those which operate on sequences. For most generic sequence functions, five specialized vector versions are provided: for any vectors, for general vectors (those specialized vectors which can hold any LISP object), bit-vectors, strings, and vectors of a specified type. If *x* is the name of a generic sequence function, then as a rule the type-specific functions are named as follows:

Name	Type of vector operated upon
`vector-`*x*	Vectors of any kind
`bit-`*x*	Bit vectors

missing from the manual

`string-`*x*	Strings
`v`*x*	General vectors (those of type `(vector t)`)
`v`*x*`@`	Vectors of a type indicated by the first argument

Use of such a type-specific function implies that any sequence arguments must be of the specified vector type, any arguments stored into or compared with elements of a vector must be of an appropriate type, and that the result will be a vector or element of the appropriate type.

Strings have such important and distinctive uses that there are many functions on strings which are not generalized to arbitrary sequences. String functions are therefore described in another chapter.

15.1. Creating Vectors

Take keyword argum

`make-vector` *length* `&optional` *type initial-value* [*Function*]

A new vector is created and returned. It will contain *length* elements; *length* must be a non-negative integer. It will be of type `(vector` *ctype*`)`, where *ctype* is the most specific type for which the implementation provides a concrete representation of vectors of that type, such that *type* is a subtype of *ctype*. The *type* defaults to `t`. Each element of the vector will be *initial-value*, which must be of type *type*; but if the *initial-value* is not provided, then the initial contents of the vector are implementation-dependent (but each element must nevertheless be of type *type*).

`make-bit-vector` *length* `&optional` *initial-value* [*Function*]

This is precisely equiva;

15.2. Functions on General Vectors (Vectors of LISP Objects)

The functions in this section are equivalent in operation to the corresponding generic sequence functions, but require sequence arguments to be vectors of type `(vector t)`.

`vref` *vector index* [*Function*]

The element of the *vector* specified by the integer *index* is returned. The *index* must be non-negative and less than the length of the vector. See `elt` (page 108).

`vset` *vector index newvalue* [*Function*]

The LISP object *newvalue* is stored into the component of the *vector* specified by the integer *index*. The *index* must be non-negative and less than the length of the vector. The *newvalue* must be suitable for storing into the *vector* if the *vector* is of a specialized type. See `setelt` (page 108).

I am unhappy about using this name for a function whose arguments are not in the same order as aset.

`subvec` *vector start* `&optional` *end* [*Function*]

`copyvec` *vector* [*Function*]

`vlength` *vector* [*Function*]

`vfill` *vector item* `&optional` *start end* [*Function*]

`vreplace` *target-vector source-vector* `&optional` *target-start source-start target-end source-end* [*Function*]

`vreverse` *vector* [*Function*]

`vnreverse` *vector* [*Function*]

`vconcat` `&rest` *vectors* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `subseq` (page 108), `copyseq` (page 109), `length` (page 109), `fill` (page 109), `replace` (page 110), `reverse` (page 110), `nreverse` (page 110), and `concat` (page 110).

`vreduce` *function vector* `&optional` *start-value* [*Function*]

`left-vreduce` *function vector* `&optional` *start-value* [*Function*]

`right-vreduce` *function vector* `&optional` *start-value* [*Function*]

`vmap` *function* `&rest` *vectors* [*Function*]

`vsome` *predicate* `&rest` *vectors* [*Function*]

`vevery` *predicate* `&rest` *vectors* [*Function*]

`vnotany` *predicate* `&rest` *vectors* [*Function*]

`vnotevery` *predicate* `&rest` *vectors* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `reduce` (page 111), `left-reduce` (page 111), `right-reduce` (page 111), `map` (page 112), `some` (page 112), `every` (page 112), `notany` (page 112), `notevery` (page 112).

`vremove` *item vector* `&optional` *count* [*Function*]

`vremq` *item vector* `&optional` *count* [*Function*]

`vrem` *predicate item vector* `&optional` *count* [*Function*]

`vrem-if` *predicate vector* `&optional` *count* [*Function*]

`vrem-if-not` *predicate vector* `&optional` *count* [*Function*]

`vremove-from-end` *item vector* `&optional` *count* [*Function*]

`vremq-from-end` *item vector* `&optional` *count* [*Function*]

`vrem-from-end` *predicate item vector* `&optional` *count* [*Function*]

`vrem-from-end-if` *predicate vector* `&optional` *count* [*Function*]

`vrem-from-end-if-not` *predicate vector* `&optional` *count* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `remove` (page 113).

`vposition` *item vector* `&optional` *start end* [*Function*]
`vposq` *item vector* `&optional` *start end* [*Function*]
`vpos` *predicate item vector* `&optional` *start end* [*Function*]
`vpos-if` *predicate vector* `&optional` *start end* [*Function*]
`vpos-if-not` *predicate vector* `&optional` *start end* [*Function*]
`vposition-from-end` *item vector* `&optional` *start end* [*Function*]
`vposq-from-end` *item vector* `&optional` *start end* [*Function*]
`vpos-from-end` *predicate item vector* `&optional` *start end* [*Function*]
`vpos-from-end-if` *predicate vector* `&optional` *start end* [*Function*]
`vpos-from-end-if-not` *predicate vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `position` (page 114).

`vscan-over` *item vector* `&optional` *start end* [*Function*]
`vscanq` *item vector* `&optional` *start end* [*Function*]
`vscan` *predicate item vector* `&optional` *start end* [*Function*]
`vscan-if` *predicate vector* `&optional` *start end* [*Function*]
`vscan-if-not` *predicate vector* `&optional` *start end* [*Function*]
`vscan-over-from-end` *item vector* `&optional` *start end* [*Function*]
`vscanq-from-end` *item vector* `&optional` *start end* [*Function*]
`vscan-from-end` *predicate item vector* `&optional` *start end* [*Function*]
`vscan-from-end-if` *predicate vector* `&optional` *start end* [*Function*]
`vscan-from-end-if-not` *predicate vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `scan-over` (page 114).

`vcount` *item vector* `&optional` *start end* [*Function*]
`vcntq` *item vector* `&optional` *start end* [*Function*]
`vcnt` *predicate item vector* `&optional` *start end* [*Function*]
`vcnt-if` *predicate vector* `&optional` *start end* [*Function*]
`vcnt-if-not` *predicate vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `count` (page 115).

`vmismatch` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmismatq` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmismat` *predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmismatch-from-end` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmismatq-from-end` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

`vmismat-from-end` *predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `mismatch` (page 116).

`vmaxprefix` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxprefq` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxpref` *predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxsuffix` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxsuffq` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxsuff` *predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `maxprefix` (page 117).

`vsearch` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrchq` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrch` *predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsearch-from-end` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrchq-from-end` *vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrch-from-end` *predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `search` (page 118).

`vsort` *vector predicate* [*Function*]
`vsortcar` *vector predicate* [*Function*]
`vsortslot` *vector key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`. See `sort` (page 118).

`vmerge` *vector1 vector2 predicate* [*Function*]
`vmergecar` *vector1 vector2 predicate* [*Function*]
`vmergeslot` *vector1 vector2 key-function predicate* [*Function*]
`vnmerge` *vector1 vector2 predicate* [*Function*]
`vnmergecar` *vector1 vector2 predicate* [*Function*]
`vnmergeslot` *vector1 vector2 key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`v`", except that sequence arguments must be vectors of type `(vector t)`.

See `merge` (page 120) and `nmerge` (page 121).

15.3. Functions on Bit-Vectors

Most of the functions in this section are equivalent in operation to the corresponding generic sequence functions, but require sequence arguments to be bit-vectors. Because `eq` is not guaranteed to work on integers, the `eq`-versions of the generic sequence functions are not provided.

`bit` *bit-vector index* [*Function*]

The element of the *bit-vector* specified by the integer *index* is returned. The *index* must be non-negative and less than the length of the vector. The result will always be `0` or `1`. See `elt` (page 108).

`rplacbit` *bit-vector index newbit* [*Function*]

The *newbit* is stored into the component of the *bit-vector* specified by the integer *index*. The *index* must be non-negative and less than the length of the vector. The *newvalue* must be `0` or `1`. See `setelt` (page 108).

`sub-bits` *bit-vector start* `&optional` *end* [*Function*]
`copybits` *bit-vector* [*Function*]
`bit-length` *bit-vector* [*Function*]
`bit-fill` *bit-vector bit* `&optional` *start end* [*Function*]
`bit-replace` *target-vector source-vector* `&optional` *target-start source-start target-end source-end* [*Function*]
`bit-reverse` *bit-vector* [*Function*]
`bit-nreverse` *bit-vector* [*Function*]
`bit-concat` `&rest` *bit-vectors* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`" or end with "`bits`", except that sequence arguments must be bit-vectors. See `subseq` (page 108), `copyseq` (page 109), `length` (page 109), `fill` (page 109), `replace` (page 110), `reverse` (page 110), `nreverse` (page 110), and `concat` (page 110).

`bit-reduce` *function bit-vector* `&optional` *start-value* [*Function*]
`bit-left-reduce` *function bit-vector* `&optional` *start-value* [*Function*]
`bit-right-reduce` *function bit-vector* `&optional` *start-value* [*Function*]
`bit-map` *function* `&rest` *bit-vectors* [*Function*]
`bit-some` *predicate* `&rest` *bit-vectors* [*Function*]
`bit-every` *predicate* `&rest` *bit-vectors* [*Function*]
`bit-notany` *predicate* `&rest` *bit-vectors* [*Function*]
`bit-notevery` *predicate* `&rest` *bit-vectors* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not

begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors.

Implementation note: Implementations are free to use the following trick if deemed advisable: determine the effect of the *function* or *predicate* once for each relevant combination of bits, and then use these cached results to perform the operation. For example, one might implement `some` by applying the *predicate* to 0 and to 1 once each, and then dispatching to one of four pieces of code:

Result on 0	Result on 1	Action
`()`	`()`	Return `()`.
`()`	`t`	Search for a 1 bit.
`t`	`()`	Search for a 0 bit.
`t`	`t`	Return `(not (zerop (length` *bit-vector*`)))`.

This seems like an unreasonable restriction on the implementation.

each of which can be implemented in an optimized manner. Actually, one should be more careful than this, to avoid calling *predicate* at all if the bit-vector is empty, and avoid calling it on 1 if the vector is all zeros, and vice versa.

See `reduce` (page 111), `left-reduce` (page 111), `right-reduce` (page 111), `map` (page 112), `some` (page 112), `every` (page 112), `notany` (page 112), `notevery` (page 112).

`bit-remove` *bit bit-vector* `&optional` *count* [*Function*]

`bit-rem` *predicate bit bit-vector* `&optional` *count* [*Function*]

`bit-rem-if` *predicate bit-vector* `&optional` *count* [*Function*]

`bit-rem-if-not` *predicate bit-vector* `&optional` *count* [*Function*]

`bit-remove-from-end` *bit bit-vector* `&optional` *count* [*Function*]

`bit-rem-from-end` *predicate bit bit-vector* `&optional` *count* [*Function*]

`bit-rem-from-end-if` *predicate bit-vector* `&optional` *count* [*Function*]

`bit-rem-from-end-if-not` *predicate bit-vector* `&optional` *count* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `remove` (page 113).

`bit-position` *bit bit-vector* `&optional` *start end* [*Function*]

`bit-pos` *predicate bit bit-vector* `&optional` *start end* [*Function*]

`bit-pos-if` *predicate bit-vector* `&optional` *start end* [*Function*]

`bit-pos-if-not` *predicate bit-vector* `&optional` *start end* [*Function*]

`bit-position-from-end` *bit bit-vector* `&optional` *start end* [*Function*]

`bit-pos-from-end` *predicate bit bit-vector* `&optional` *start end* [*Function*]

`bit-pos-from-end-if` *predicate bit-vector* `&optional` *start end* [*Function*]

`bit-pos-from-end-if-not` *predicate bit-vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `position` (page 114).

`bit-scan-over` *bit bit-vector* `&optional` *start end* [*Function*]
`bit-scan` *predicate bit bit-vector* `&optional` *start end* [*Function*]
`bit-scan-if` *predicate bit-vector* `&optional` *start end* [*Function*]
`bit-scan-if-not` *predicate bit-vector* `&optional` *start end* [*Function*]
`bit-scan-over-from-end` *bit bit-vector* `&optional` *start end* [*Function*]
`bit-scan-from-end` *predicate bit bit-vector* `&optional` *start end* [*Function*]
`bit-scan-from-end-if` *predicate bit-vector* `&optional` *start end* [*Function*]
`bit-scan-from-end-if-not` *predicate bit-vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `scan-over` (page 114).

`bit-count` *bit bit-vector* `&optional` *start end* [*Function*]
`bit-cnt` *predicate bit bit-vector* `&optional` *start end* [*Function*]
`bit-cnt-if` *predicate bit-vector* `&optional` *start end* [*Function*]
`bit-cnt-if-not` *predicate bit-vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `count` (page 115).

`bit-mismatch` *bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`bit-mismat` *predicate bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`bit-mismatch-from-end` *bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]

`bit-mismat-from-end` *predicate bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `mismatch` (page 116).

`bit-maxprefix` *bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`bit-maxpref` *predicate bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`bit-maxsuffix` *bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`bit-maxsuff` *predicate bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `maxprefix` (page 117).

`bit-search` *bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`bit-srch` *predicate bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`bit-search-from-end` *bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]

`bit-srch-from-end` *predicate bit-vector1 bit-vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `search` (page 118).

`bit-sort` *bit-vector predicate* [*Function*]
`bit-sortcar` *bit-vector predicate* [*Function*]
`bit-sortslot` *bit-vector key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `sort` (page 118).

??? Query: These functions are incredibly useless, but have an efficient (linear-time) implementation!

`bit-merge` *bit-vector1 bit-vector2 predicate* [*Function*]
`bit-mergecar` *bit-vector1 bit-vector2 predicate* [*Function*]
`bit-mergeslot` *bit-vector1 bit-vector2 key-function predicate* [*Function*]
`bit-nmerge` *bit-vector1 bit-vector2 predicate* [*Function*]
`bit-nmergecar` *bit-vector1 bit-vector2 predicate* [*Function*]
`bit-nmergeslot` *bit-vector1 bit-vector2 key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with the prefix "`bit-`", except that sequence arguments must be bit-vectors. See `merge` (page 120) and `nmerge` (page 121).

`bit-and` `&rest` *bit-vectors* [*Function*]
`bit-ior` `&rest` *bit-vectors* [*Function*]
`bit-xor` `&rest` *bit-vectors* [*Function*]
`bit-eqv` `&rest` *bit-vectors* [*Function*]
`bit-nand` *bit-vector1 bit-vector2* [*Function*]
`bit-nor` *bit-vector1 bit-vector2* [*Function*]
`bit-andc1` *bit-vector1 bit-vector2* [*Function*]
`bit-andc2` *bit-vector1 bit-vector2* [*Function*]
`bit-orc1` *bit-vector1 bit-vector2* [*Function*]
`bit-orc2` *bit-vector1 bit-vector2* [*Function*]

These functions perform bit-wise logical operations on bit-vectors. All of the arguments to any of these functions must be bit-vectors, all of the same length. The result is a bit-vector matching the argument(s) in length, such that bit *j* of the result is produced by operating on bit *j* of each of the arguments. Indeed, if the arguments are in fact bit-vectors of the same length, then (`bit-`*xxx* `.` *arguments*`)` `<=>` `(bit-map #'log`*xxx* `.` *arguments*`)`

That is, each `bit-` function described here is simply a mapping over bit-vectors of a `log` function which applies to integers (and therefore to the bit values 0 and 1).

The following table indicates what the result bit is for each operation when two arguments are given. (Those operations which accept an indefinite number of arguments are commutative and associative, and require at least one argument.)

text out of place

Argument 1	0	0	1	1	
Argument 2	0	1	0	1	*Operation name*
`bit-and`	0	0	0	1	and
`bit-ior`	0	1	1	1	inclusive or
`bit-xor`	0	1	1	0	exclusive or
`bit-eqv`	1	0	0	1	equivalence (exclusive nor)
`bit-nand`	1	1	1	0	not-and
`bit-nor`	1	0	0	0	not-or
`bit-andc1`	0	1	0	0	and complement of arg1 with arg2
`bit-andc2`	0	0	1	0	and arg1 with complement of arg2
`bit-orc1`	1	1	0	1	or complement of arg1 with arg2
`bit-orc2`	1	0	1	1	or arg1 with complement of arg2

`bit-not` *bit-vector* [*Function*]

The argument must be a bit-vector. A copy of the argument with all the bits inverted is returned. That is, bit *j* of the result is 1 iff bit *j* of the argument is zero.

`(bit-not` *bitvec*`)` `<=>` `(bit-map #'lognot` *bitvec*`)`

15.4. Functions on Vectors of Explicitly Specified Type

The functions in this section are equivalent in operation to the corresponding generic sequence functions, but require sequence arguments to be vectors of type `(vector` *type*`)`, where *type* is specified as the first argument to the function. (If this *type* argument is a quoted constant, then the compiler for some implementations may be able to exploit this type information to produce more efficient code.)

`vref@` *type vector index* [*Function*]

The element of the *vector* specified by the integer *index* is returned. The *index* must be non-negative and less than the length of the vector. See `elt` (page 108).

`vset@` *type vector index newvalue* [*Function*]

The LISP object *newvalue* is stored into the component of the *vector* specified by the integer *index*. The *index* must be non-negative and less than the length of the vector. The *newvalue* must be suitable for storing into the *vector* (it must be of type *type*). See `setelt` (page 108).

`subvec@` *type vector start* `&optional` *end* [*Function*]

`copyvec@` *type vector* [*Function*]

`vlength@` *type vector* [*Function*]

`vfill@` *type vector item* `&optional` *start end* [*Function*]

`vreplace@` *type target-vector source-vector* `&optional` *target-start source-start target-end source-end* [*Function*]

`vreverse@` *type vector* [*Function*]

`vnreverse@` *type vector* [*Function*]

`vconcat@` *type* `&rest` *vectors* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with "v" and end with "@", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `subseq` (page 108), `copyseq` (page 109), `length` (page 109), `fill` (page 109), `replace` (page 110), `reverse` (page 110), `nreverse` (page 110), and `concat` (page 110).

`vreduce@` *type function vector* `&optional` *start-value* [*Function*]

`left-vreduce@` *type function vector* `&optional` *start-value* [*Function*]

`right-vreduce@` *type function vector* `&optional` *start-value* [*Function*]

`vmap@` *type function* `&rest` *vectors* [*Function*]

`vsome@` *type predicate* `&rest` *vectors* [*Function*]

`vevery@` *type predicate* `&rest` *vectors* [*Function*]

`vnotany@` *type predicate* `&rest` *vectors* [*Function*]

`vnotevery@` *type predicate* `&rest` *vectors* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with "v" and end with "@", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `reduce` (page 111), `left-reduce` (page 111), `right-reduce` (page 111), `map` (page 112), `some` (page 112), `every` (page 112), `notany` (page 112), `notevery` (page 112).

`vremove@` *type item vector* `&optional` *count* [*Function*]

`vremq@` *type item vector* `&optional` *count* [*Function*]

`vrem@` *type predicate item vector* `&optional` *count* [*Function*]

`vrem-if@` *type predicate vector* `&optional` *count* [*Function*]

`vrem-if-not@` *type predicate vector* `&optional` *count* [*Function*]

`vremove-from-end@` *type item vector* `&optional` *count* [*Function*]

`vremq-from-end@` *type item vector* `&optional` *count* [*Function*]

`vrem-from-end@` *type predicate item vector* `&optional` *count* [*Function*]

`vrem-from-end-if@` *type predicate vector* `&optional` *count* [*Function*]

`vrem-from-end-if-not@` *type predicate vector* `&optional` *count* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with "v" and end with "@", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `remove` (page 113).

`vposition@` *type item vector* `&optional` *start end* [*Function*]

`vposq@` *type item vector* `&optional` *start end* [*Function*]

`vpos@` *type predicate item vector* `&optional` *start end* [*Function*]

`vpos-if@` *type predicate vector* `&optional` *start end* [*Function*]

`vpos-if-not@` *type predicate vector* `&optional` *start end* [*Function*]

`vposition-from-end@` *type item vector* `&optional` *start end* [*Function*]

`vposq-from-end@` *type item vector* `&optional` *start end* [*Function*]

`vpos-from-end@` *type predicate item vector* `&optional` *start end* [*Function*]

`vpos-from-end-if@` *type predicate vector* `&optional` *start end* [*Function*]

`vpos-from-end-if-not@` *type predicate vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with `"v"` and end with `"@"`, except that sequence arguments must be vectors of type `(vector` *type*`)`. See `position` (page 114).

`vscan-over@` *type item vector* `&optional` *start end* [*Function*]

`vscanq@` *type item vector* `&optional` *start end* [*Function*]

`vscan@` *type predicate item vector* `&optional` *start end* [*Function*]

`vscan-if@` *type predicate vector* `&optional` *start end* [*Function*]

`vscan-if-not@` *type predicate vector* `&optional` *start end* [*Function*]

`vscan-over-from-end@` *type item vector* `&optional` *start end* [*Function*]

`vscanq-from-end@` *type item vector* `&optional` *start end* [*Function*]

`vscan-from-end@` *type predicate item vector* `&optional` *start end* [*Function*]

`vscan-from-end-if@` *type predicate vector* `&optional` *start end* [*Function*]

`vscan-from-end-if-not@` *type predicate vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with `"v"` and end with `"@"`, except that sequence arguments must be vectors of type `(vector` *type*`)`. See `scan-over` (page 114).

`vcount@` *type item vector* `&optional` *start end* [*Function*]

`vcntq@` *type item vector* `&optional` *start end* [*Function*]

`vcnt@` *type predicate item vector* `&optional` *start end* [*Function*]

`vcnt-if@` *type predicate vector* `&optional` *start end* [*Function*]

`vcnt-if-not@` *type predicate vector* `&optional` *start end* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with `"v"` and end with `"@"`, except that sequence arguments must be vectors of type `(vector` *type*`)`. See `count` (page 115).

`vmismatch@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

`vmismatq@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

`vmismat@` *type predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

`vmismatch-from-end@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

`vmismatq-from-end@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*] `vmismat-from-end@` *type predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with "v" and end with "@", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `mismatch` (page 116).

`vmaxprefix@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxprefq@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxpref@` *type predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxsuffix@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxsuffq@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vmaxsuff@` *type predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with "v" and end with "@", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `maxprefix` (page 117).

`vsearch@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrchq@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrch@` *type predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsearch-from-end@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrchq-from-end@` *type vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]
`vsrch-from-end@` *type predicate vector1 vector2* `&optional` *start1 start2 end1 end2* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with "v" and end with "@", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `search` (page 118).

`vsort@` *type vector predicate* [*Function*]
`vsortcar@` *type vector predicate* [*Function*]
`vsortslot@` *type vector key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not begin with "v" and end with "@", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `sort` (page 118).

`vmerge@` *type vector1 vector2 predicate* [*Function*]
`vmergecar@` *type vector1 vector2 predicate* [*Function*]
`vmergeslot@` *type vector1 vector2 key-function predicate* [*Function*]
`vnmerge@` *type vector1 vector2 predicate* [*Function*]
`vnmergecar@` *type vector1 vector2 predicate* [*Function*]
`vnmergeslot@` *type vector1 vector2 key-function predicate* [*Function*]

These functions are exactly like the corresponding generic sequence functions whose names do not

begin with "`v`" and end with "`@`", except that sequence arguments must be vectors of type `(vector` *type*`)`. See `merge` (page 120) and `nmerge` (page 121).

Chapter 16

Arrays

16.1. Array Creation

`make-array` *dimensions* `&rest` *options* [*Function*]

This is the primitive function for making arrays. *dimensions* should be a list of non-negative integers (in fact, fixnums) which are the dimensions of the array; the length of the list will be the dimensionality of the array. For convenience when making a one-dimensional array, the single dimension may be provided as a fixnum rather than a list of one fixnum.

There must be an even number of *options* arguments; they are alternating keywords and values, each keyword having one associated value. Valid keywords are:

`:type`	The value should be the name of the type of the elements of the array; an array is constructed of the most specialized type which can nevertheless accommodate elemments of the given type. The type `t` specifies a general array, one whose elements may be any LISP object; this is the default type.
`:initial`	The value is used to initialize each element of the array. The value must be of the type specified by the `:type` option. If the `:initial` option is omitted, the initial values of the array elements are undefined (unless the `:displaced-to` option is used). The `:initial` option may not be used with the `:displaced-to` option.
`:leader-length`	The value should be a non-negative fixnum. The array will have a leader with that many elements. The elements of the leader will be initialized to `()` unless the `:leader-list` option is also given.
`:leader-list`	The value should be a list. Call the number of elements in the list *n*. The first *n* elements of the leader will be initialized from successive elements of this list. If the `:leader-length` option is not specified, then the length of the leader will be *n*. If the `:leader-length` option is given, and its value is greater than *n*, then all leader elements after the first *n* will be initialized to `()`. If the specified `:leader-length` is less than *n*, an error is signalled. The leader elements are filled in forward order; that is, the *car* of the list will be stored in leader element `0`, the *cadr* in element `1`, and so on.

Not all array types are distinguishable by their element types. The Lisp machine will accept other things here in addition to data types.

:initial-value or something would be more grammatical.

`:displaced-to`

If the value is not `()`, then the array will be a *displaced* array. The value must be an array or vector; `make-array` will create an *indirect* or *shared* array which shares its contents with the specified array. In this case the `:displaced-index-offset` option may be useful. The `:displaced-to` option may not be used with the `:initial` option.

`:displaced-index-offset`

If this is present, the value of the `:displaced-to` option should be an array, and the value of this option should be a non-negative fixnum; it is made to be the index-offset of the created shared array.

Compatibility note: The Lisp Machine LISP `:area` and `:named-structure-symbol` keywords are omitted here, and `:initial` is new.

For example:

```
;; Create a one-dimensional array of five elements.
(make-array 5)

;; Create a two-dimensional array, 3 by 4, with four-bit elements.
(make-array '(3 4) ':type '(mod 16))

;; Create an array of single-floats with a three-element leader.
(make-array 5 ':leader-length 3 ':type ':single-float)

;; The same thing, providing initial values for the leader elements.
(setq a (make-array 100 ':type ':single-float
                        ':leader-list '(0 () foo)))
(array-leader a 0) => 0
(array-leader a 1) => ()
(array-leader a 2) => foo

;; Making a shared array.
(setq a (make-array '(4 3)))
(setq b (make-array 8 ':displaced-to a
                      ':displaced-index-offset 2))
;; Now it is the case that:
        (aref b 0) <=> (aref a 0 2)
        (aref b 1) <=> (aref a 1 0)
        (aref b 2) <=> (aref a 1 1)
        (aref b 3) <=> (aref a 1 2)
        (aref b 4) <=> (aref a 2 0)
        (aref b 5) <=> (aref a 2 1)
        (aref b 6) <=> (aref a 2 2)
        (aref b 7) <=> (aref a 3 0)
```

Do types have colons or don't they?

The last example depends on the fact that arrays are, in effect, stored in row-major order for purposes of sharing. Put another way, the sequences of indices for the elements of an array are ordered lexicographically.

Compatibility note: Both Lisp Machine LISP and FORTRAN store arrays in column-major order.

WE WILL FIX THE LM, AT LEAST THE L-MACHINE. FORTRAN WILL HAVE TO FEND FOR ITSELF.

??? Query: From the Lisp Machine LISP manual: "`make-array` returns the newly-created array, and also returns, as a second value, the number of words allocated in the process of creating the array, i.e. the `%structure-total-size` of the array."

(Ignore this, it is a historical artifact of something like the cold-load generator.

16.2. Array Access

`aref` *array* `&rest` *subscripts* [*Function*]

This accesses and returns the element of *array* specified by the *subscripts*. The number of subscripts must equal the rank of the array, and each subscript must be a non-negative integer less than the corresponding array dimension.

`aset` *new-value array* `&rest` *subscripts* [*Function*]

This stores *new-value* into the element of *array* specified by the *subscripts*. The number of subscripts must equal the rank of the array, and each subscript must be a non-negative integer less than the corresponding array dimension. The result of `aset` is the value *new-value*.

The argument *new-value* must be of a type suitable for storing into *array* if the *array* is of a specialized type.

16.3. Array Information

`array-type` *array* [*Function*]

This returns the type of elements of the array. For a general array, this is `t`; for an array of eight-bit integers, `(mod 256)` might be returned. What is returned is the actual type of the array elements, which may be the same as that specified to `make-array`, or may be more general if the implementatation doesn't support arrays of that specific type.

`array-length` *array* [*Function*]

array may be any array. This returns the total number of elements allocated in *array*. For a one-dimensional array, this is equal to the length of the single axis. (If a fill pointer is in use for the array, however, the function `array-active-length` (page 177) may be more useful.)

`array-active-length` *array* [*Function*]

`array-active-length` returns the fill pointer for the array. (IF NO FILL PTR, RETURNS LENGTH. This is normally the same as the length of the array unless `reset-fill-pointer` (page RESET-FILL-POINTER-FUN) has been used. NOT REALLY TRUE.

`array-rank` *array* [*Function*]

Returns the number of dimensions (axes) of *array*. This will be a non-negative integer.

Compatibility note: In Lisp Machine LISP this is called `array-#-dims`. This name causes problems in MACLISP because of the # character. The problem is better avoided.

`array-dimension` *axis-number array* [*Function*]

The length of dimension number *axis-number* of the *array* is returned. *array* may be any kind of array, and *axis-number* should be a non-negative integer less than the rank of *array*.

> Compatibility note: This is similar to the Lisp Machine LISP function `array-dimension-n`, but is zero-origin for consistency instead of one-origin. Also, in Lisp Machine LISP `(array-dimension-n 0)` returns the length of the array leader; in COMMON LISP `array-leader-length` (page ARRAY-LEADER-LENGTH-FUN) must be used for that purpose.

`array-dimensions` *array* [*Function*]

`array-dimensions` returns a list whose elements are the dimensions of *array*.

`array-in-bounds-p` *array* `&rest` *subscripts* [*Function*]

This function checks whether the *subscripts* are all legal subscripts for *array*, and returns `t` if they are; otherwise it returns `()`. The *subscripts* may be any LISP objects.

Surely it signals an error if they aren't integers (fixnums in most implementations, actually)

16.4. Array Leaders

Any array may have associated with it an extra vector of type `(vector t)` called its *leader*. The following functions are used to manipulate this leader.

`array-has-leader-p` *array* [*Function*]

array may be any array. This predicate returns `t` if *array* has a leader; otherwise it returns `()`.

`array-leader-length` *array* [*Function*]

array may be any array. This returns the length of *array*'s leader (as a non-negative integer) if it has a leader, or `()` if it does not.

`array-leader` *array index* [*Function*]

This returns element number *index* of the leader of *array*. *array* should be an array with a leader, and *index* should be a non-negative integer less than the length of the leader. (This function is like a `vref` (page 162) on the leader vector.)

`store-array-leader` *new-value array index* [*Function*]

The object *new-value* is stored into element number *index* of the leader of *array*. *array* should be an array with a leader, and *index* should be a non-negative integer less than the length of the leader. `store-array-leader` returns *new-value*. (This function is like a `vset` (page 162) on the leader vector.)

16.5. Fill Pointers

To make it easy to incrementally fill in the contents of an array, a set of functions for manipulating a *fill pointer* are defined. The fill pointer is a non-negative integer no larger than the total number of elements in the array (as returned by `array-length` (page 177)); it is the number of "active" or "filled-in" elements in the array. When an array is created, its fill pointer is initialized to the number of elements in the array; the fill pointer should be *reset* before use. The fill pointer constitutes the "active length" of the array. Some functions will ignore elements beyond the fill-pointer index; those that do are so documented.

→ IT'S NOT JUST FOR FILLING; IT'S ALSO FOR VARYING STRINGS AND 1-D ARRAYS.

Multidimensional arrays may have fill pointers; elements are filled in row-major order (last index varies fastest).

??? Query: The following comes from Lisp Machine Lisp, and is somewhat of a crock. Should this be retained for compatability? (If so, fill pointers should be initialized to ( ), not the array-length.)

"By convention, the fill pointer is kept in element number 0 of the array's leader. We say that an array *has a fill pointer* if the array has a leader of non-zero length and element number 0 of the leader is an integer. Normally there is no fill pointer."

It would be nice if fill pointers and named structures did not interact so randomly with the leader. (Then again, what's a leader for?)

EXACTLY...

`array-reset-fill-pointer` *array* `&optional` *index* [*Function*]

The fill pointer of *array* is reset to *index*, which defaults to zero. The *index* must be a non-negative integer not greater than the old value of the fill pointer.

`array-push` *array* *new-element* [*Function*]

array must be an array which has a fill pointer, and *new-element* may be any object. `array-push` attempts to store *new-element* in the element of the array designated by the fill pointer, and increase the fill pointer by one. If the fill pointer does not designate an element of the array (specifically, when it gets too big), it is unaffected and `array-push` returns `( )`. Otherwise, the two actions (storing and incrementing) happen uninterruptibly, and `array-push` returns the *former* value of the fill pointer (one less than the one it leaves in the array); thus the value of `array-push` is the index of the new element pushed.

Compatibility note: In Lisp Machine Lisp the array is required to be one-dimensional; at least, so states the documentation. Is this true? Also, should the requirement of uninterruptibility be retained?

I don't care

2 It is an error in the documentation. However the multi-d

Case is essentially usel

`array-push-extend` *array* *x* `&optional` *extension* [*Function*]

`array-push-extend` is just like `array-push` except that if the fill pointer gets too large, the array is extended (using `adjust-array-size` (page ADJUST-ARRAY-SIZE-FUN)) so that it can contain more elements; it never "fails" the way `array-push` does, and so never returns `( )`. The optional argument *extension*, which must be a positive integer, is the minimum number of elements to be added to the array if it must be extended.

`array-pop` *array* [*Function*]

array must be an array which has a fill pointer. The fill pointer is decreased by one, and the array element designated by the new value of the fill pointer is returned. If the new value does not designate any element of the array (specifically, if it has reached zero), an error occurs. The two operations (decrementing and array referencing) happen uninterruptibly.

> Compatibility note: In Lisp Machine LISP the array is required to be one-dimensional; at least, so states the documentation. Is this true? Also, should the requirement of uninterruptibility be retained?

Currently it is.

16.6. Changing the Size of an Array

`adjust-array-size` *array* *new-size* `&optional` *new-element* [*Function*]

The *array* is adjusted so that it contains (at least) *new-size* elements. The argument *new-size* must be a non-negative integer.

If *array* is a one-dimensional array, its size is simply changed to be *new-size*, by altering its single dimension. If *array* has more than one dimension, then its *first* dimension is adjusted to the smallest possible value which allows the array to have no fewer than *new-size* elements. If any dimension other than the first is zero, however, then the array is not changed, and an error occurs if *new-size* is not `0`. If the array has zero dimensions, then the array is not changed, and an error occurs if *new-size* is not `0` or `1`.

If *array* is made smaller, the extra elements are lost. If *array* is made bigger, the new elements are initialized to *new-element*; if this argument is not provided, then the initial contents of new elements are undefined.

If the *array* is used to share with other arrays, then after the `adjust-array-size` operation it may or may not continue to be shared with other arrays.

But wh
does this
mean?
does it ha
anything to d
with displac

`adjust-array-size` returns *array* as its value.

> Compatibility note: In Lisp Machine LISP, the argument *new-element* is not provided; it would seem useful, however. Also, in Lisp Machine LISP it is possible for the returned array not to be `eq` to the argument array. Should this be reflected in the above definition?
>
> Also the Lisp Machine LISP manual is unclear on the precise method of extension for multidimensional arrays. The above definition ties this down.

If not the implementation would be forced
a particular implementation of arrays (separat
header

Note that in
case in the L
machine, the
array will ca
to work,
it will b
forwarde
to the ne
array if N
is one. M
this expli
als

`array-grow` *array* `&rest` *dimensions* [*Function*]

`array-grow` returns an array of the same type as *array*, with the specified *dimensions*. The number of *dimensions* given must equal the rank of *array*. (Actually, if an extra argument is provided at the end, it is construed to be an optional argument *new-element*. Unfortunately, one cannot write

(*array* `&rest` *dimensions* `&optional` *new-element*)

as a parameter list.)

This is a godawful crock. I would rather you changed it incompatibly to add an extra (required) argument)

Those elements of *array* that are still in bounds appear in the new array. The elements of the new array that are not in the bounds of *array* are initialized to *new-element*; if this argument is not provided, then the initial contents of any new elements are undefined.

`array-grow` may, depending on the implementation and the arguments, simply alter the given array or create and return a new one. If a new array is created, it will get a leader which is a copy of the old array's leader, and moreover its contents will *not* be shared with any other arrays. Therefore, `array-grow` should not be applied to a shared array, in general.

? What's a sh
array.

If the *array* used to share with other arrays, then after the `array-grow` operation it may or may not continue to be shared with other arrays.

`array-grow` differs from `adjust-array-size` in that it keeps the elements of a multidimensional array in the same logical positions while allowing extension of any or all dimensions, not just the first.

SINCE THE KEYWORD IS
"DISPLACED-TO", WHY NOT
JUST CALL THEM DISPLACED
ARRAYS RATHER THAN
USING "DISPLACED",
"INDIRECT", AND "[illegible]
"SHARED" INTERCHANGABL

Chapter 17

Structures

COMMON LISP provides a facility for creating named record structures with named components. In effect, the user can declare a new data type; every data structure of that type has components with specified names. Constructor, access, and assignment constructs are automatically defined when the data type is declared.

This chapter is divided into two parts. The first part discusses the basics of the structure facility, which is very simple and allows the user to take advantage of the type-checking, modularity, and convenience of user-defined record data types. The second part discusses a number of specialized features of the facility which have advanced applications. These features are completely optional, and you needn't even know they exist in order to take advantage of the basics.

> **Rationale:** It is important not to scare the novice away from `defstruct` with a multiplicity of features. The basic idea is very simple, and we should encourage its use by providing a very simple description. The hairy stuff, including all options, is shoved to the end of the chapter. SURE

17.1. Introduction to Structures

The structure facility is embodied in the `defstruct` macro, which allows the user to create and use aggregate datatypes with named elements. These are like "structures" in PL/I, or "records" in PASCAL.

As an example, assume you are writing a LISP program that deals with space ships in a two-dimensional plane. In your program, you need to represent a space ship by a LISP object of some kind. The interesting things about a space ship, as far as your program is concerned, are its position (represented as *x* and *y* coordinates), velocity (represented as components along the *x* and *y* axes), and mass.

A ship might therefore be represented as a record structure with five components: *x*-position, *y*-position, *x*-velocity, *y*-velocity, and mass. This structure could in turn be implemented as a LISP object in a number of ways. It could be a list of five elements; the *x*-position could be the *car*, the *y*-position the *cadr*, and so on. Equally well it could be a vector of five elements: the *x*-position could be element 0, the *y*-position element 1, and so on. The problem with either of these representations is that the components occupy places in the object which are quite arbitrary and hard to remember. Someone looking at `(cadddr ship1)` or `(vref ship1 3)` in a piece of code might find it difficult to determine that this is accessing the *y*-velocity component of `ship1`. Moreover, if the representation of a ship should have to be changed, it would be very difficult top find all the places in the code to be changed to match (not all occurrences of `cadddr` are intended to extract the *y*-velocity from a ship).

Ideally components of record structures should have names. One would like to write something like `(ship-y-velocity ship1)` instead of `(cadddr ship1)`. One would also like a more mnemonic way to create a ship than this:

```
    (list 0 0 0 0 0)
```

Indeed, one would like `ship` to be a new data type, just like other LISP data types, that one could test with `typep` (page 26), for example. The `defstruct` facility provides all of this.

`defstruct` itself is a macro which defines a structure. For the space ship example one we might define the structure by saying:

```
    (defstruct ship
      ship-x-position
      ship-y-position
      ship-x-velocity
      ship-y-velocity
      ship-mass)
```

This declares that every `ship` is an object with five named components. The evaluation of this form does several things:

- It defines `ship-x-position` to be a function of one argument, a ship, which returns its *x*-position; `ship-y-position` and the other components are given similar function definitions. These functions are called the *access functions*, as they are used to access elements of the structure.

- The symbol `ship` becomes the name of a data type, of which instances of ships are elements. This name becomes acceptable to `typep` (page 26), for example; `(typep x 'ship)` is true iff `x` is a ship. Moreover, all ships are instances of the type `structure`, because `ship` is a subtype of `structure`.

  Define what 1-argument typep does.

- A function named `ship-p` of one argument is defined; it is a predicate which returns `t` if its argument is a ship, and `()` otherwise.

- A macro called `make-ship` is defined which, when invoked, will create a data structure with five components, suitable for use with the access functions. Thus executing

  ```
      (setq ship2 (make-ship))
  ```

  sets `ship2` to a newly-created `ship` object. One can specify the initial values of any desired component in the call to `make-ship` in this way:

  ```
      (setq ship2 (make-ship ship-mass *default-ship-mass*
                             ship-x-position 0
                             ship-y-position 0))
  ```

  This constructs a new ship and initializes three of its components. This macro is called the *constructor macro*, because it constructs a new structure.

- Two ways are provided to alter components of a `ship`. One way is to use the macro `setf` (page SETF-FUN) in conjunction with an access function (because `defstruct` performs an appropriate `defsetf` (page DEFSETF-FUN)):

```
(setf (ship-x-position ship2) 100)
```

This alters the *x*-position of *ship2* to be `100`. This works because `defstruct` generates an appropriate `defsetf` (page DEFSETF-FUN) form for each access function.

I assum
it does
actually
this is ju
simplifica
So do it o
it fits

The other way is to use the special *alterant macro*, which allows alteration of several components at once in parallel:

```
(alter-ship enterprise            ;Counter-clockwise inter-quadrant warp!
            ship-x-position (- (ship-y-position enterprise))
            ship-y-position (ship-x-position enterprise))
```

Besides allowing parallel updating of several components, use of the alterant macro may be more efficient in certain cases.

This simple example illustrates the power of `defstruct` to provide abstract record structures in a convenient manner. `defstruct` has many other features as well for specialized purposes.

17.2. How to Use Defstruct

`defstruct` *name-and-options* `&rest` *slot-descriptions* [*Macro*]

Defines a record-structure data type. A general call to `defstruct` looks like this:

```
(defstruct (name option-1 option-2 ...)
           slot-description-1
           slot-description-2
           ...)
```

name must be a symbol; it becomes the name of a new data type consisting of all instances of the structure. The function `typep` (page 26) will accept and use this name as appropriate.

Usually no options are needed at all. If no options are specified, then one may write simply *name* instead of (*name*) after the word `defstruct`. The syntax of options and the options provided are discussed in section ???.

WOULDN'T ALTERNATING KEYWORDS
AND VALUES BE MORE OBVIOUS,
IE ANALOGOUS TO OTHER THINGS?

Each *slot-description-j* is of the form

```
(slot-name default-init slot-option-1 slot-option-2 ...)
```

Each *slot-name* must be a symbol; an access function is defined for each slot. If no options and no *default-init* are specified, then one may write simply *slot-name* instead of (*slot-name*) as the slot description. The *default-init* is a form which is evaluated *each time* a structure is to be constructed; the value is used as the initial value of the slot. If no *default-init* is specified, then the initial contents of the slot are undefined and implementation-dependent. The available slot-options are described in section ???.

> Compatibility note: Slot-options are not currently provided in Lisp Machine LISP, but this is an upward-compatible extension.

Besides defining an access function for each slot, `defstruct` arranges for `setf` to work properly on such access functions, defines a predicate named *name*-`p`, and defines constructor and alterant macros named `make-`*name* and `alter-`*name*, respectively.

Because evaluation of a `defstruct` form causes many functions and macros to be defined, one must take care that two `defstruct` forms do not define the same name (just as one must take care not to use `defun` to define two distinct functions of the same name). For this reason, as well as for clarity in the code, it is conventional to prefix the names of all of the slots with some text which identifies the structure. In the example above, all the slot names start with "`ship-`". The `:conc-name` (page 184)option can be used to provide such prefixes automatically.

space

No! The slot names are different from the access function names. Don't sow unnecessary confusion.

17.3. Using the Automatically Defined Macros

After you have defined a new structure with `defstruct`, you can create instances of this structure by using the constructor macro, and alter the values of its slots by using the alterant macro. By default, `defstruct` defines these macros automatically, forming their names by adding prefixes to the name of the structure; for a structure named `foo`, the respective macro names would be `make-foo` and `alter-foo`. You can specify the names yourself by giving the name you want to use as the argument to the `:constructor` (page 186) and `:alterant` (page 186) options, or specify that you don't want a macro created at all by using `()` as the argument.

17.3.1. Constructor Macros

A call to a constructor macro, in general, has the form

```
(name-of-constructor-macro
        slot-name-1  form-1
        slot-name-2  form-2
        ...)
```

Each *slot-name* should be the name of a slot of the structure. All the *forms* are evaluated.

If *slot-name-j* is the name of a slot, then that element of the created structure will be initialized to the value of *form-j*. If no *slot-name-j*/*form-j* pair is present for a given slot, then the slot will be initialized by evaluating the *default-init* form specified for that slot in the call to `defstruct`. (In other words, the initialization specified in the `defstruct` defers to any specified in a call to the constructor macro.) If the default initialization form is used, it is evaluated at construction time, but in the lexical environment of the `defstruct` form in which it appeared. If the `defstruct` itself also did not specify any initialization, the element's initial value is undefined. You should always specify the initialization, either in the `defstruct` or in the constructor macro, if you care about the initial value of the slot.

Compatibility note: The Lisp Machine LISP documentation is slightly unclear about when the initialization specified in the `defstruct` form gets evaluated: at `defstruct` evaluation time, or at constructor time? The code reveals that it is at constructor time, which causes problems with referential transparency with respect to lexical variables (which currently don't exist officially in Lisp Machine LISP anyway). The above remark concerning the lexical environment in effect requires that the initialization form is treated as a thunk; it is evaluated at constructor time, but in the environment where it was written (the `defstruct` environment). Most of the time this makes no difference anyway, as the initialization form is typically a quoted constant or refers only to special variables. The requirement is imposed here for uniformity, and to ensure that what look like special variable references in the initialization form are in fact always treated as such.

The order of evaluation of the initialization forms is *not* necessarily the same as the order in which they appear in the constructor call or in the `defstruct` form; code should not depend on the order of evaluation. The initialization forms *are* re-evaluated on every constructor-macro call, so that if, for example, the form `(gensym)` were used as an initialization form, either in the constructor-macro call or as the default form in the `defstruct` declaration, then every call to the constructor macro would call `gensym` once to generate a new symbol.

17.3.2. Alterant Macros

A call to the alterant macro, in general, has the form

(*name-of-alterant-macro instance-form*
    slot-name-1 form-1
    slot-name-2 form-2
    ...)

instance-form is evaluated, and should return an instance of the structure. Each *form-j* is evaluated, and the corresponding slot named by *slot-name-j* is changed to have the result as its new value. The assignments are parallel; that is, the slots are altered *after* all the *forms* have been evaluated, so you can exchange the values of two slots, as follows:

```
(alter-ship enterprise
    ship-x-position (ship-y-position enterprise)
    ship-y-position (ship-x-position enterprise))
```

As with the constructor macro, the order of evaluation of the *forms* is undefined.

Single slots can also be altered by using `setf` (page SETF-FUN). Using the alterant macro may produce more efficient code than using consecutive `setf` forms.

17.4. `defstruct` Slot-Options

Each *slot-description* in a `defstruct` form may specify one or more slot-options. A slot-option may be a keyword, or a list of a keyword and arguments for that keyword.

For example:

```
(defstruct ship
  (ship-x-position 0.0 (:type :short-float))
  (ship-y-position 0.0 (:type :short-float))
  (ship-x-velocity 0.0 (:type :short-float) :invisible)
  (ship-y-velocity 0.0 (:type :short-float) :invisible)
  (ship-mass *default-ship-mass* :invisible :read-only))
```

specifies that the first four slots will always contain short-format floating-point numbers, that the last three slots are "invisible" (will not ordinarily be shown when a ship is printed), and that the last slot may not be altered once a ship is constructed.

The available slot-options are:

`:type` The option (`:type` *type*) specifies that the contents of the slot will always be of the specified data type. This is entirely analogous to the declaration of a variable or function; indeed, it effectively declares the result type of the access function. An implementation may or may not choose to check the type of the new object when initializing or assigning to a slot.

`:invisible` The option `:invisible` specifies that the contents of this slot should not be printed when an instance of the structure is printed.

`:read-only` The option `:read-only` specifies that this slot may not be altered; it will always contain the value specified at construction time. The alterant macro will not accept the name of this slot, and `setf` (page SETF-FUN) will not accept the access function for this slot.

17.5. Options to `defstruct`

The preceding description of `defstruct` is all that the average user will need (or want) to know in order to use structures. The remainder of this chapter discusses more complex features of the `defstruct` facility.

This section explains each of the options that can be given to `defstruct`. As with slot-options, a `defstruct` option may be either a keyword or a list of a keyword and arguments for that keyword.

`:conc-name` This provides for automatic prefixing of names of access functions. It is conventional to begin the names of all the access functions of a structure with a specific prefix, usually the name of the structure followed by a hyphen. If you do not use the `:conc-name` option, then the names of the access functions are the same as the slot names, and it is up to you to name the slots reasonably.

Specifying the `:conc-name` option causes each access functions to have a name consisting of a standard prefix followed by the name of the accessed slot. If the `:conc-name` option has an argument, it should be a string specifying the prefix, or a symbol whose print-name is the prefix. With no argument, the prefix is the name of the structure and a hyphen.

Note that in the constructor and alterant macros, you still use the slot names rather than the access function names. On the other hand, one uses the access-function name when using `setf`. Here is an example:

```
(defstruct (door :conc-name) knob-color width material)
(setq my-door (make-door knob-color 'red width 5.0))
(door-knob-color my-door) ==> red
(alter-door my-door  knob-color 'green  material 'wood)
(door-material my-door) => wood
(setf (door-width my-door) 43.7)
(door-width my-door) => 43.7
```

EXTRA SPACING HERE
LOOKS WEIRD.

There are defstruct
structure types
are not data
tree is the
esoteric

`:type` The `:type` option specifies what kind of LISP object will be used to implement the structure. It takes one argument, which must be one of the types enumerated below. If the `:type` option is not provided, the type defaults to `:vector`, and the `:named` option is assumed unless `:unnamed` is explicitly specified.

an implementation-chosen default type, such as vecter

Rationale: Making a structure be `:unnamed` mostly just saves space. It is probably better to protect align

the novice by providing by default a named vector, since that provides maximal features, nice printing, reasonable use of space (better than lists or arrays in most implementations), etc.

Possibly it would be better for the default to be unnamed if a type option was specified. I'm unsure.

`:vector` Use a general vector, storing components as vector elements. This is normally `:named`.

`(vector` *type*`)` A specialized vector may be used, in which case every component must be of a type which can be stored in such a vector. A structure of this type must be `:unnamed`.

Compatibility note: This is a suggested feature not yet in Lisp Machine LISP.

`:array` Use a one-dimensional array, storing components in the body of the array. By default this is `:named`.

`(array` *type*`)` A specialized array may be used, in which case every component must be of a type which can be stored in such an array. The array must be one-dimensional, and a structure of this type must be `:unnamed`.

This restric
is bogu
What a
array lea
for

Compatibility note: This is a suggested feature not yet in Lisp Machine LISP.

WE HAVE LOTS OF SPECIALIZED ARRAYS THAT ARE NAMED!

`:array-leader` Use an array, storing components in the *leader* of the array. By default this is `:named`. (See the option `:make-array` (page 188), described below.)

`:list` Use a list. A structure of this type cannot be distinguished by `typep`, even if the `:named` option is used. By default this is `:unnamed`.

`:integer` This unusual type implements the structure as a single integer. The structure may only have one slot. This is only useful with the byte field feature (see page DEFSTRUCT-BYTE-FIELD); it lets you store several small numbers within fields of an integer, giving the fields names. This cannot be `:named`.

which ha
to gotten

where did the byte fields go?

Compatibility note: The `:integer` option is a suggested feature not yet in Lisp Machine LISP. It is similar to the `fixnum` option.

flush.

`:fixnum` The `:fixnum` option is similar to the `:integer` option, but further declares that in fact a fixnum may be used.

??? **Query:** Is this really necessary? Or can this be determined at `defstruct` expansion time from the byte field information? THE LATTER.

Compatibility note: All the "`named-`" types such as `:named-array` from Lisp Machine LISP have been omitted here, as they tend to multiply. An implementation may provide them but they are not required here. The `:named` and `:unnamed` options may be used separately to get the same effect.

`:named` The `:named` option specifies that the structure is "named"; this option takes no argument. A named structure has an associated predicate for determining whether a given LISP object is a structure of that name. Some named structures in addition can be distinguished by the predicate `typep` (page 26). If neither `:named` nor `:unnamed` is specified, then the default depends on the `:type` option.

`:unnamed` The `:unnamed` option specifies that the structure is not named; this option takes no argument.

`:constructor`

This option takes one argument, a symbol, which specifies the name of the constructor macro. If the argument is not provided or if the option itself is not provided, the name of the constructor is produced by concatenating the string `"make-"` and the name of the structure. If the argument is provided and is `()`, no constructor macro is defined.

This option actually has a more general syntax which is explained in ???.

`:alterant` This option takes one argument, which specifies the name of the alterant macro. If the argument is not provided or if the option itself is not provided, the name of the alterant macro is made by concatenating the string `"alter-"` to the name of the structure. If the argument is provided and is `()`, no alterant macro is defined. Use of the alterant macro is explained on ???.

`:predicate` This option takes one argument, which specifies the name of the type predicate. If the argument is not provided or if the option itself is not provided, the name of the predicate is made by concatenating the name of the structure to the string `"-p"`. If the argument is provided and is `()`, no predicate is defined. A predicate can be defined only if the structure is `:named` (page 189).

`:include` This option is used for building a new structure definition as an extension of an old structure definition. As an example, suppose you have a structure called `person` that looks like this:

```
(defstruct (person :conc-name)
   name
   age
   sex)
```

Now suppose you want to make a new structure to represent an astronaut. Since astronauts are people too, you would like them to also have the attributes of name, age, and sex, and you would like LISP functions that operate on `person` structures to operate just as well on `astronaut` structures. You can do this by defining `astronaut` with the `:include` option, as follows:

```
(defstruct (astronaut (:include person))
   helmet-size
   (favorite-beverage 'tang))
```

The `:include` option causes the structure being defined to have the same slots as the included structure, in such a way that the access functions and alterant macro for the included structure will also work on the structure being defined. In this example, an `astronaut` will therefore have five slots: the three defined in `person`, and the two defined in `astronaut` itself. The access functions defined by the `person` structure can be applied to instances of the `astronaut` structure, and they will work correctly. The following examples illustrate how you can use `astronaut` structures:

```
(setq x (make-astronaut name 'buzz
                        age 45.
                        sex t
                        helmet-size 17.5))

(person-name x) => buzz
(favorite-beverage x) => tang
```

Note that the `:conc-name` (page 188) option was *not* inherited from the included structure; it only applies to the names of the access functions of `person` and not to those of `astronaut`.

EXPLAIN WHAT THIS DOES W.R.T. 2-ARG TYPEP. THAT IS, ALL ASTRONAUTS ARE ALSO PERSONS.

The argument to the `:include` option is required, and must be the name of some previously defined structure. The included structure must be of the same `:type` as this structure. The structure name of the including structure definition becomes the name of a data type, of course; moreover, it becomes a subtype of the included structure. In the above example, `astronaut` is a subtype of `person`; hence

or a sub typ

```
(typep (make-astronaut) 'person)
```

is true, indicating that all operations on persons will work on astronauts.

The following is an advanced feature of the `:include` option. Sometimes, when one structure includes another, the default values or slot-options for the slots that came from the included structure are not what you want. The new structure can specify default values or slot-options for the included slots different from those the included structure specifies, by giving the `:include` option as:

`(:include` *name slot-description-1 slot-description-2* `...)`

Each *slot-description-j* must have a *slot-name* which is the same as that of some slot in the included structure. If *slot-description-j* has no *default-init*, then in the new structure the slot will have no initial value. Otherwise its initial value form will be replaced by the *default-init* in *slot-description-j*. A normally writable slot may be made read-only, and a normally visible slot may be made invisible in the defined structure. If a slot is invisible or read-only in the included structure, then it must also be so in the including structure. If a type is specified for a slot, it must be a the same as or a subtype of the type specified in the included structure. If it is a strict subtype, the implementation may or may not choose to error-check assignments.

This seems like a silly restriction

? what does this mean?

For example, if we had wanted to define `astronaut` so that the default age for an astronaut is 45, then we could have said:

```
(defstruct (astronaut (:include person (age 45)))
   helmet-size
   (favorite-beverage 'tang))
```

`:make-array` If an array is used to represent the structure being defined (the `:type` (page 188) option is `:array` or `:array-leader`), this option allows you to control those aspects of the array used to implement the structure that are not otherwise constrained by `defstruct`. For example, if you are creating a structure of type `:array-leader`, you almost certainly want to specify the dimensions of the array to be created, and you may want to specify the type of the array.

or (array ...)

The argument to the `:make-array` option should be a list of alternating keyword symbols to the function `make-array` (page 175) and forms whose values are the arguments to those keywords.

`defstruct` may need to specify some arguments to `make-array` for its own purposes. If these conflict with the specifications given to the `:make-array` keyword, an error is signalled.

> **Compatibility note:** This is more robust than the current Lisp Machine LISP specification that `defstruct` quietly overrides what you specify.

Accidentally left over from before this option was keyword-oriented.

Constructor macros for structures implemented as arrays all allow the keyword `:make-array` to be supplied. Attributes supplied therein override any `:make-array` option attributes supplied in the original `defstruct` form. If some attribute appears in neither the invocation of the constructor nor in the `:make-array` option to `defstruct`, then the constructor will chose appropriate defaults.

If a structure is of type `:array-leader`, you probably want to specify the dimensions of the array. The dimensions of an array are given to `:make-array` as a position argument rather than a keyword argument, so there is no way to specify them in the above syntax. To solve this problem, you may use the special keyword `:dimensions` or `:length` (they mean the same thing), with a value that is anything acceptable as `make-array`'s first argument.

`:size-variable`

The `:size-variable` option allows a user to specify a global (special) variable whose value will be the "size" of the structure; `defconst` (page 22) is used to declare this variable. The exact meaning of this size varies, but in general this number is the one you would need to know if you were going to allocate one of these structures yourself. The variable will have this value both at compile time and at run time. If this option is present without an argument, then the name of the structure is concatenated with `"-size"` to produce the name.

> ??? **Query:** The name of this option in Lisp Machine LISP, `size-symbol`, indicates a confusion between "symbol" and "variable" (which tends to pervade Lisp Machine LISP). A symbol can represent or implement a variable, but also a function, a macro, etc. OK

`:size-macro` This is similar to the `:size-symbol` option. A macro of no arguments is defined that expands into the size of the structure. The name of this macro is the argument to the option; this argument defaults as with `:size-symbol`. It is permissible to use the `:size-symbol` and `:size-macro` options in the same `defstruct` form.

YOU MIGHT EXPLAIN THAT THIS REQUIRES PARENS (BAD) BUT ELIMINATES RUNTIME SYMEVAL (GOOD).

`:print-function`

The argument to this option should be a function of four arguments which is to be used to print structures of this type. When a structure of this type is to be printed, the function is called on the structure to be printed, a stream to print to, an integer indicating the current depth (to be compared against `prinlevel` (page PRINLEVEL-VAR)), and a flag which is `t` for `prin1`-style printout and `()` for `princ`-style printout. This option can be used only with `:named` structures.

> **Compatibility note:** This is suggested merely to provide a simple way to set up the printing function in a central place and in an implementation-independent manner. In Lisp Machine LISP this would

I think DESCRIBE should be part of the core system. It is especially useful with defstructs. Alan has a DESCRIBE and an INSPECT for Maclisp which he finds extremely convenient. They're pretty simple, too.

(ME TOO, WHEN I USE MACLISP!)

presumably set up an invoke handler for the type. There needs to be a good way to interface to the grinder, too.

If the default way of printing a defstruct is going to be to dump its guts, there should be an option which automatically writes a print function which prints somethi conc ~~[illegible]~~ (e.g. shows options content a slot probably same name, redefine such a addler in nea >)

`:initial-offset`
This allows you to tell `defstruct` to skip over a certain number of slots before it starts allocating the slots described in the body. This option requires an argument (which must be a non-negative integer) which is the number of slots you want `defstruct` to skip. To make use of this option requires that you have some familiarity with how `defstruct` is implementing your structure; otherwise, you will be unable to make use of the slots that `defstruct` has left unused.

`:callable-accessors`
This option controls whether access functions are really functions, and therefore "callable", or whether thay are really macros. With an argument of `t`, or with no argument, or if the option is not provided, then the accessors are really functions. If the option is not provided, they are additionally declared `inline`, so that the compiler can integrate them into calling code for faster execution; explicitly providing the option suppresses this, so that they may be traced, for example. If the argument is `()` then the accessors will really be macros, defined by `defmacro` (page DEFMACRO-FUN), just like the constructor and alterant macros.

Compatibility note: So what about the above, which is not really compatible with Lisp Machine LISP? ? how

I SEE. DIFFERENCE IS MINOR AND OK.

`:eval-when`
Normally the macros defined by `defstruct` are defined at eval time, compile time, and load time. This option allows the user to control this behavior. The argument to the `:eval-when` option is just like the list that is the first subform of an `eval-when` (page EVAL-WHEN-FUN) special form. For example,

```
(:eval-when (:eval :compile))
```

will cause the macros to be defined only when the code is running interpreted or inside the compiler.

17.6. By-position Constructor Macros

If the `:constructor` (page 190) option is given as `(:constructor` *name arglist*`)`, then instead of making a keyword driven constructor, `defstruct` defines a "function style" constructor, taking arguments whose meaning is determined by the argument's position rather than by a keyword. The *arglist* is used to describe what the arguments to the constructor will be. In the simplest case something like `(:constructor make-foo (a b c))` defines `make-foo` to be a three-argument constructor macro whose arguments are used to initialize the slots named `a`, `b`, and `c`.

In addition, the keywords `&optional`, `&rest`, and `&aux` are recognized in the argument list. They work in the way you might expect, but there are a few fine points worthy of explanation.

For example:

```
(:constructor create-foo
        (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines `create-foo` to be a constructor of one or more arguments. The first argument is used to initialize the a slot. The second argument is used to initialize the b slot. If there isn't any second argument, then the default value given in the body of the `defstruct` (if given) is used instead. The third argument is used to initialize the c slot. If there isn't any third argument, then the symbol `sea` is used instead. Any arguments following the third argument are collected into a list and used to initialize the d slot. If there are three or fewer arguments, then `()` is placed in the d slot. The e slot *is not initialized*; its initial value is undefined. Finally, the f slot is initialized to contain the symbol `eff`.

The actions taken in the b and e cases were carefully chosen to allow the user to specify all possible behaviors. Note that the `&aux` "variables" can be used to completely override the default initializations given in the body.

With this definition, one can write

```
(create-foo 1 2)
```

instead of

```
(make-foo a 1 b 2)
```

and of course `create-foo` provides defaulting different from that of `make-foo`.

It is permissible to use the `:constructor` option more than once, so that you can define several different constructors, each with a different syntax.

Because this kind of constructor is a function, the arguments in a call to one will be evaluated in order. This is unlike a constructor macro, which may evaluate initialization forms in any order.

Compatibility note: In Lisp Machine LISP the evaluation can be in any order. This is a bad idea. It's not so hard to make it behave as a real function. Also, if you don't guarantee order, it's hard to let `&optional` and `&aux` initialization forms refer to earlier variables properly; this is essential if we are not to confuse the user by using lambda-list syntax.

If you write the keyword `:make-array` in place of a variable name, then the corresponding argument will specify the `:make-array` option at construction time, just as for a constructor macro.

Compatibility note: Lisp Machine LISP doesn't allow this, but it's consistent and convenient.

17.7. The `si:defstruct-description` Structure

*** This section not fully worked over yet. ***

This section discusses the internal structures used by `defstruct` that might be useful to programs that want to interface to `defstruct` nicely. The information in this section is also necessary for anyone who is thinking of defining his own structure types.

Whenever `defstruct` defines a new structure, it creates an instance of the `si:defstruct-description` structure. This structure can be found as the `si:defstruct-description` property of the name of the structure; it contains such useful information as the name of the structure, the number of slots in the structure, and so on. The `si:defstruct-description` structure is defined as follows, in the `system-internals` package (also called the `si` package): (This is a simplified version of the real definition. There are other slots in the structure which we aren't telling you about.)

```
(defstruct (defstruct-description
               (:default-pointer description)
               (:conc-name defstruct-description-))
           name
           size
           property-alist
           slot-alist)
```

The `name` slot contains the symbol supplied by the user to be the name of his structure, such as `spaceship` or `phone-book-entry`. The `size` slot contains the total number of locations in an instance of this kind of structure. This is *not* the same number as that obtained from the `:size-symbol` or `:size-macro` options to `defstruct`. A named structure, for example, usually uses up an extra location to store the name of the structure, so the `:size-macro` option will get a number one larger than that stored in the `defstruct` description. The `property-alist` slot contains an alist with pairs of the form `(`*property-name* `.` *property*`)` containing properties placed there by the `:property` option to `defstruct` or by property names used as options to `defstruct` (see the `:property` option, page DEFSTRUCT-PROPERTY-OPTION). The `slot-alist` slot contains an alist of pairs of the form `(`*slot-name* `.` *slot-description*`)`. A *slot-description* is an instance of the `defstruct-slot-description` structure. The `defstruct-slot-description` structure is defined something like this, also in the `si` package: (This is a simplified version of the real definition. There are other slots in the structure which we aren't telling you about.)

```
(defstruct (defstruct-slot-description
               (:default-pointer slot-description)
               (:conc-name defstruct-slot-description-))
           number
           ppss
           init-code
           ref-macro-name)
```

??? Query: Ought to flush the `:default-pointer` option? Also, it would be nicer to rename `ppss` to `byte-specifier`; `ppss` has representation-dependent connotations. Finally, may need a `data-type` slot.

The `number` slot contains the number of the location of this slot in an instance of the structure. Locations are numbered starting with `0`, and continuing up to one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of the structure.

The `ppss` slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, then the `ppss` slot contains `()`.

The `init-code` slot contains the initialization code supplied for this slot by the user in his `defstruct` form. If there is no initialization code for this slot then the init-code slot contains the symbol `si:%%defstruct-empty%%`.

The `ref-macro-name` slot contains the symbol that is defined as a macro that expands into a reference to this slot (that is, the name of the accessor macro).

Chapter 18

EVAL

Chapter 19

Input/Output

19.1. Printed Representation of LISP Objects

LISP objects are not normally thought of as being text strings; they have very different properties from text strings as a consequence of their internal representation. However, to make it possible to get at and talk about LISP objects, LISP provides a representation of objects in the form of printed text; this is called the *printed representation*, which is used for input/output purposes and in the examples throughout this manual. Functions such as `print` (page 216) take a LISP object and send the characters of its printed representation to a stream. The collection of routines which does this is known as the (LISP) *printer*. The `read` function takes characters from a stream, interprets them as a printed representation of a LISP object, builds a corresponding object, and returns it; the collection of routines that does this is called the (LISP) *reader*.

Ideally, one could print a LISP object and then read the printed representation back in, and so obtain the same identical object. In practice this is difficult, and for some purposes not even desirable. Instead, reading a printed representation produces an object which is (with obscure technical exceptions) `equal` (page 31) to the originally printed object.

Most LISP objects have more than one possible printed representation. For example, the integer twenty-seven can be written in any of these ways:

```
27    27.    #o33    #x1B    #b11011    #.(* 3 3 3)
```

A list of two symbols `A` and `B` can be printed in many, many ways:

```
(A B)    (a b)    (  a  b )    (\A |B|)
    (|\A|
  B
)
```

The last example, which is spread over three lines, may be ugly, but it is legitimate. In general, wherever whitespace is permissible in a printed representation, any number of spaces, tab characters, and newlines may appear. and <FORM>

When `print` produces a printed representation, it must choose arbitrarily from among many possible printed representations. It attempts to choose one which is readable. There are a number of global variables which can be used to control the actions of `print`, and a number of different printing functions.

This section describes in detail what is the standard printed representation for any Lisp object, and also describes how `read` operates.

19.1.1. What the `read` Function Accepts

The purpose of the reader LISP is to accept characters, interpret them as the printed representation of a LISP object, and construct and return such an object. The reader cannot accept everything that the printer produces; for example, the printed representations of compiled code objects and closures cannot be read in. However, the reader has many features which are not used by the output of the printer at all, such as comments, alternative representations, and convenient abbreviations for frequently-used unwieldy constructs. The reader is also parameterized in such a way that it can be used as a lexical analyzer for a more general user-written parser.

When the reader is invoked, it reads a character from the input stream and dispatches according to the attributes of that character. Every character which can appear in the input stream can have one of the following attributes:

- *Whitespace.*
- *Constituent.*
- *Macro-character.*
- *Escape-character.*
- *Ignored.*

Supposing that the one character has been read; call it *x*. The reader then performs the following actions:

- If *x* is a *whitespace* or *ignored* character, then discard it and start over, reading another character.

- If *x* is a *macro character*, then execute the function associated with that character. The function may return a LISP object. If so, that object is returned by the reader; if not, the reader starts anew, reading a character from the input stream and dispatching. The function may of course read characters from the input stream; if it does, it will see those characters following the macro character.

DOES THIS MEAN IT RETURNS ZERO OR WHAT?

PUT IN ONE OF THESE ITEMS FOR "IGNORED", JUST FOR COMPLETENESS

- If *x* is an *escape character*, then read the next character and call it *x* instead, but pretend it is a *constituent*, and drop into the next case.

- If *x* is a *constituent*, then it begins an extended token, representing a symbol or a number. The reader reads more characters, accumulating them until a *whitespace* character or *macro character* is found. However, *ignored* characters are simply discarded, and whenever an *escape character* is found during the accumulation, the character after that is treated as a pure *constituent* and also accumulated, no matter what its usual syntax is. Call the eventually found *whitespace* character or *macro character y*. All characters beginning with *x* up to but not including *y* form a single extended token, which is then interpreted as a number if possible, and otherwise as a symbol. The number or symbol is then returned by the reader.

This does not allow for characters which are macro characters but do not break symbols. (being treated as constituents inside symbols.) If this is an intentional omission, put a compatibility note.

Compatibility note: Note that characters of type *single* are not provided for. They can be viewed as simply a kind of macro character. That is,

```
(setsyntax '$ 'single ())
  <=> (setsyntax '$ 'macro #'(lambda (ignore ignore) '$))
```

which is easy enough to do oneself. After all, one might prefer to see a character rather than a symbol.

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

IS THIS → A BACKQUOTE? LOOKS LIKE THE SINGLE QUOTE.

Not if there are packages

for ASR33 compatibility?

Table 19-1: Standard Character Syntax Attributes

The characters of the standard character set initially have the attributes shown in Table 19-1.

If the reader encounters a macro character, then the function associated with that macro character is called, and may produce an object to be returned. This function may read following characters in the stream in whatever syntax it likes (it may even call `read` recursively) and returns the object represented by that syntax. Macro characters may not be recognized, of course, when read as part of other special syntaxes (such as for strings).

The reader is therefore organized into two parts: the basic dispatch loop, which also distinguishes symbols and numbers, and the collection of macro characters. Any character can be reprogrammed as a macro character; this is a means by which the reader can be extended.

The general abilities of macro characters are discussed below in ???. First, however, some standard macro characters are described here:

(The left parenthesis character initiates reading of a pair or list. `Read` is called recursively to read successive objects, until a right parenthesis is found to be next in the input stream. A list of the objects read is returned. Thus

```
(a b c)
```

is read as a list of three objects (the symbols `a`, `b`, abd `c`). The right parenthesis need not follow the printed representation of the last object immediately; whitespace and ignored characters may precede it. This can be useful for putting one object on each line and making it easy to add new objects:

```
(defun traffic-light (color)
  (caseq color
    (green)
    (red (stop))
    (amber (accelerate))          ; Insert more colors after this line.
    ))
```

It may be that *no* objects precede the right parenthesis, as in "`()`" or "`( )`"; this reads as a list of zero objects (the empty list).

If a token is read between objects which is just a dot "`.`", not preceded by an escape character, then exactly one more object must follow, and then the right parenthesis:

```
(a b c . d)
```

This means that the *cdr* of the last pair in the list is not `()`, but rather the object whose representation followed the dot. The above example might have been the result of evaluating

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

Similarly, we have

```
(cons 'znets 'wolq-zorbitan) => (znets . wolq-zorbitan)
```

It is permissible for the object following the dot to be a list:

`(a b c d . (e f . (g)))` is the same as `(a b c d e f g)`

but this is a non-standard form that `print` will never produce.

) The right-parenthesis character is part of various constructs (such as the syntax for lists) using the left-parenthesis character, and is invalid except when used in such a construct.

' The single-quote (accent acute) character provides an abbreviation to make it easier to put constants in programs. `'`*foo* reads the same as `(quote` *foo*`)`: a list of the symbol `quote` and *foo*.

; Semicolon is used to write comments. The semicolon and everything up through the next newline are ignored. Thus a comment can be put at the end of any line without affecting the reader (except that semicolon, being a macro character and therefore a delimiter, will terminate a token, and so cannot be put in the middle of a number or symbol).

For example:

```
;;;; COMMENT-EXAMPLE and related nonsense.
;;; This function is useless except to demonstrate comments.
;;; Notice that there are several kinds of comments.

(defun comment-example (x y)    ;X is anything; Y is an a-list.
  (cond ((listp x) x)           ;If X is a list, use that.
        ;; X is now not a list.  There are two other cases.
        ((symbolp x)
         ;; Look up a symbol in the a-list.
         (cdr (assq x y)))      ;Remember, (cdr '()) is ().
        ;; Do this when all else fails:
        (t (cons x              ;Add x to a default list.
                 '((lisp t)     ;LISP is okay.
                   (fortran ()) ;FORTRAN is not.
                   (pl/i -500)  ;Note that you can put comments in
                   (ada .001)   ; "data" as well as in "programs".
                   ;; COBOL??
                   (teco -1.0e9))))))
```

This example illustrates a few conventions for comments in common use. Comments may begin with one to four semicolons.

- Single-semicolon comments are all aligned to the same column at the right; usually each comments about only the line it is on. Occasionally two or three contain a single sentence together; this is indicated by indenting all but the first by a space.
- Double-semicolon comments are aligned to the level of indentation of the code. A space follows the two semicolons. Usually each describes the state of the program at that point, or describes the section that follows.
- Triple-semicolon comments are aligned to the left margin. Usually they are not used within S-expressions, but precede them in large blocks.
- Quadruple-semicolon comments are interpreted as subheadings by some software such as the ATSIGN listing program.

" The double-quote character begins the printed representation of a string. Characters are read from the input stream and accumulated until another double-quote is encountered, except that if an escape character is seen, it is discarded, the next character is accumulated, and accumulation continues. When a matching double-quote is seen, all the accumulated characters up to but not including the matching double-quote are made into a string and returned.

| The vertical-bar character begins one ? printed representation of a symbol. Characters are read from the input stream and accumulated until another vertical-bar is encountered, except that if an escape

character is seen, it is discarded, the next character is accumulated, and accumulation continues. When a matching vertical-bar is seen, all the accumulated characters up to but not including the matching vertical-bar are made into a symbol and returned. In this syntax, no characters are ever converted to upper case; the name of the symbol is precisely those characters between the vertical bars (allowing for any escape characters).

' The backquote (accent grave) character makes it easier to write programs to construct complex data structures by using a template. As an example, writing

```
'(cond ((numberp ,x) ,@y) (t (print ,x) ,@y))
```

is roughly equivalent to writing

```
(list 'cond
      (cons (list 'numberp x) y)
      (list* 't (list 'print x) y))
```

See ??? for details.

, The comma character is part of the backquote syntax and is invalid if used other than inside the body of a backquote construction. See ???.

The sharp-sign character is a *dispatching* macro character. It reads an optional digit string and then one more character, and uses that character to select a function to run as a macro-character function. See the next section for predefined sharp-sign macro characters.

MAYBE THIS IS A BAD EXAMPLE SINCE "O" LOOKS LIKE "0" (ZERO). NOT IMPORTANT.

19.1.2. Sharp-Sign Abbreviations

The standard syntax includes forms introduced by a sharp sign ("#"). These take the general form of a sharp sign, a second character which identifies the syntax, and following arguments in some form. If the second character is a letter, then case is not important; `#O` and `#o` are considered to be equivalent, for example. (To be precise, # *does* distinguish case, but for all standard syntaxes which use letters the same macro-character function is initially associated with the upper-case and lower-case versions of each letter.)

Delete T
note-do
encourag
bad sty

Certain sharp-sign forms allow an unsigned decimal number to appear between the sharp sign and the second character; some other forms even require it.

The currently-defined sharp-sign constructs are described below and summarized in Table 19-2; more are likely to be added in the future.

GIVE AT LEAST ONE EXAMPLE AT BOTH POINTS.

#\ #*x* reads in as a character object which represents the character *x*. Also, #*name* reads in as the character object whose name is *name*. This is the recommended way to include character constants in your code. Note that the backslash "\" allows this construct to be parsed easily by EMACS-like editors.

Upper-case and lower-case letters are distinguished after #\; "`#\A`" and "`#\a`" denote different character objects. Any character works after #\, even those that are normally special to `read`, such as parentheses. Non-printing characters may be used after #\, although for them names are generally preferred.

#<tab>	signals error	#<form>	signals error	#<return>	signals error
#<space>	signals error	#@	undefined	#'	undefined
#!	undefined	#A	array	#a	array
#"	bit-vector	#B	undefined	#b	undefined
##	reference to label	#C	complex number	#c	complex number
#$	undefined	#D	undefined	#d	undefined
#%	undefined	#E	undefined	#e	undefined
#&	undefined	#F	undefined	#f	undefined
#'	`function` abbreviation	#G	undefined	#g	undefined
#(	vector	#H	undefined	#h	undefined
#)	signals error	#I	undefined	#i	undefined
#*	undefined	#J	undefined	#j	undefined
#+	read-time conditional	#K	undefined	#k	undefined
#,	load-time evaluation	#L	undefined	#l	undefined
#-	read-time conditional	#M	undefined	#m	undefined
#.	read-time evaluation	#N	undefined	#n	undefined
#/	undefined	#O	octal number	#o	octal number
#0	(infix argument)	#P	undefined	#p	undefined
#1	(infix argument)	#Q	undefined	#q	undefined
#2	(infix argument)	#R	radix-*n* number	#r	radix-*n* number
#3	(infix argument)	#S	structure	#s	structure
#4	(infix argument)	#T	undefined	#t	undefined
#5	(infix argument)	#U	undefined	#u	undefined
#6	(infix argument)	#V	undefined	#v	undefined
#7	(infix argument)	#W	undefined	#w	undefined
#8	(infix argument)	#X	hexadecimal number	#x	hexadecimal number
#9	(infix argument)	#Y	undefined	#y	undefined
#:	labels LISP object	#Z	undefined	#z	undefined
#;	undefined	#[	undefined	#{	undefined
#<	signals error	#\	named character	#|	undefined
#=	character integer	#]	undefined	#}	undefined
#>	undefined	#↑	undefined	#~	undefined
#?	undefined	#_	`undefined`	#<rubout>	undefined

NOT DOCUMENTED BELOW

NOT DOCUMENTED BELOW

font

Table 19-2: Standard Sharp-Sign Macro Character Syntax

#*name* reads in as a character object whose name is *name*. The following names are standard across all implementations:

`form`	The formfeed or page-separator character.
`return`	The carriage return or newline character.
`rubout`	The rubout or delete character.
`space`	The space or blank character.
`tab`	The tabulate character.

Explain how #\char and #\name are distinguished; I assume it's by whether the seco following character is a constituent (skippi ignored characters) -- why have ignored characters at all?

The *name* should have the syntax of a symbol.

When the LISP printer types out the name of a special character, it uses the same table as the `#\` reader; therefore any character name you see typed out is acceptable as input (in that implementation). Standard names are always preferred over non-standard names for printing.

The following convention is used in implementations which support non-zero bits attributes for character objects. If a name after `#\` is longer than one character and has a hyphen in it, then it may be split into the two parts preceding and following the first hyphen; the first part may then be interpreted as the name or initial of a bit, and the second part as the name of the character (which may in turn contain a hyphen and be subject to splitting).

For example:

```
#\Control-Space                 #\Control-Meta-Tab
#\C-M-Return                    #\H-S-M-C-Rubout
```

If the character name consists of a single character, then that character is used. Another "\" may be necessary to quote the character.

?? →

```
#\Control-                      #\Control-Meta-\"
#\Control-\a                    #\Meta->
```

If an unsigned decimal integer appears between the "#" and "\", it is interpreted as a font number, to become the `char-font` (page 101) of the character object.

> Compatibility note: Formerly, Lisp Machine LISP and MACLISP used `#\` to mean only the `#\`*name* version of this syntax, using `#/` for the `#\`*x* version. Lisp Machine LISP has recently changed to allow `#/` to handle both syntaxes. The incompatibility is a result of the general exchange of the / and \ characters.
>
> Also, MACLISP and Lisp Machine LISP define `#\` and `#/` to be a syntax for *numbers*, integers which represent characters. Here they are a syntax for character objects. Code conforming to the "Character Standard for LISP" will not depend on this distinction; but non-conforming code (such as that which does arithmetic on bare character values) may not be compatible.

#' `#'`*foo* is an abbreviation for (`function` *foo*). *foo* may be the printed representation of any LISP object. This abbreviation can be remembered by analogy with the ' macro-character, since the `function` and `quote` special forms are similar in form. ?

#(A series of representations of objects enclosed by "#(" and ")" is read as a general vector of those objects. This is analogous to the notation for lists.

If an unsigned decimal integer appears between the "#" and "(", it specifies explicitly the length of the vector. In that case, it is an error if too many objects are specified before the closing ")", and if too few are specified the last one is used to fill all remaining elements of the vector.

For example:

```
#(a b c c c c)
#6(a b c c c c)
#6(a b c)
#6(a b c c)
```

all mean the same thing: a vector of length 6 with elements a, b, and four instances of c.

If the character "@" immediately follows the left parenthesis ("#(@"), then after the "@" and before the representation of the first vector element should appear the representation *type* of a data type. All the elements of the vector must be of this type, and the vector created will be of type `(vector` *type*`)`.

What an awful crock! Isn't there a better way?

For example:

```
#(@short-float 0.0s0 1.0s4)      ; Vector of two short-floats.
#(@(mod 4) 1 3 2 0 1 1 3 2)      ; Vector of eight (mod 4) integers.
#100(@(mod 3) 0)                 ; 100-long vector of ternary digits, all 0.
```

#" A series of binary digits (0 and 1) enclosed by "#"" and """ is read as a bit vector of those objects. This is analogous to the notation for strings.

the first

If an unsigned decimal integer appears between the "#" and """, it specifies explicitly the length of the bit vector. In that case, it is an error if too many bits are specified before the closing """, and if too few are specified the last one is used to fill all remaining elements of the bit vector.

For example:

```
#"101000"
#6"101000"
#6"1010"
#6"10100"
#(@(mod 2) 1 0 1 0 0 0)
```

all mean the same thing.

#A This syntax denotes an array. A general array may be notated as "#*n*a" followed by the notation for a LISP object. The infix argument *n* indicates the rank (number of dimensions) of the array. The notated LISP object following "#*n*a" indicates the contents of the array in the following recursive manner. If the rank *n* is zero, then the object is the single element of the array. Otherwise, the object must be a sequence (a list or vector). The length of that sequence is the first dimension of the array, and each element of the sequence must be an object describing the contents of an array of rank *n*-1 whose dimensions are the remaining dimensions of the rank-*n* array.

For example:

```
#0A foo                          ; A rank-0 array whose element is a symbol.
#2A((8 1 6) (3 5 7) (4 9 2))     ; A 3-by-3 array containing a magic square.
#1A#(2 3 5 7 11 13 17 19)        ; A one-dimensional array with eight primes.
#2A("ROT" "HEH" "ORE")           ; A 3-by-3 matrix of characters.
                                 ;    (The columns spell words also:
                                 ;    RHO, OER, and THE.)
#1A("ROT" "HEH" "ORE")           ; A 1-dimensional, length-3 array of strings.
#3A((() ()) (() ()) (() ()))     ; A 3-by-2-by-0 array; it has no elements.
#2A((() ()) (() ()) (() ()))     ; A 3-by-2 array, all of whose elements are ().
```

As another example, this notation is *not* legal:

```
#2a((1 2 3) (a b) (w x y z))
```

because the sublists are not all the same length, so it is unclear what the second dimension should be.

For convenience, if all of the sequences for rank-1 arrays in this notation are of the same specialized type, then the array will have that same underlying specialization if possible.

For example:

```
#2A("ROT" "HEH" "ORE")          ;This is of type (array string-char)
                                ;    if the implementation supports that type.
#1A#"00110101000101000101"      ;This is of type (array (mod 2))
                                ;    if the implementation supports that type.
```

Alternatively, one may specify the specialization explicitly. If the character immediately following the "A" is "@", then between the "@" and the representation of the array contents should appear the representation of the element type.

For example:

```
#2A@(mod 4) ((3 2) (0 1) (2 1)) ;A 3-by-2 array of type (array (mod 4)).
```

There is a problem with the #*n*A notation: there is no way to write a 0-by-3 array, for example. One might try writing `#2A()`, but this fails to specify that the second dimension should be 3. Another, less serious, problem with this notation is that it is annoying to notate a large array all of whose elements are the same.

There is a second notation for arrays that solves these problems. If no integer specifying the rank is written between the "#" and the "A", then there should follow the notation for a sequence of integers and then the notation for a sequence of objects. The length of the first sequence is the rank, and its elements are the dimensions; the second sequence specifies the values of the array elements in row-major order, with the understanding that if too few are given then the last element of the sequence is replicated, and if the sequence is empty the array contents are not initialized (it is as if no `:initial` option were specified to `make-array` (page 175)).

No notation for leaders (and otherwise non-extensible)

#A is awful!

For example:

```
#A()(foo)                   ;Another notation for a zero-rank array containing foo.
#A(100 100 100)(0.0)        ;A cube, 100 on an edge, filled with 0.0.
#A(3 3)"ROTHEHORE"          ;The same matrix of characters as above.
#A(100)(2 3 5 0)            ;A 100-long array, filled with 2, 3, 5, and 97 zeros.
#A@(mod 7)(100)(2 3 5 0)    ;The same thing, but of specialized type.
#A(0 3)()                   ;A 0-by-3 array.
#A(0 4 0 5)()               ;A 0-by-4-by-0-by-5 array.
```

`#.` `#.`*foo* is read as the object resulting from the evaluation of the LISP object represented by *foo*, which may be the printed representation of any LISP object. The evaluation is done during the `read` process, when the `#.` construct is encountered. This, therefore, performs a "read-time" evaluation of *foo*. By contrast, `#,` (see below) performs a "load-time" evaluation.

This allows you, for example, to include in your code complex list-structure constants which cannot be written with `quote`. Note that the reader does not put `quote` around the result of the evaluation. You must do this yourself if you want it, typically by using the `'` macro-character. An example of a case where you do not want `quote` around it is when this object is an element of a constant list.

`#,` `#,`*foo* is read as the object resulting from the evaluation of the LISP object represented by *foo*, which may be the printed representation of any LISP object. The evaluation is done during the `read` process, *unless* ~~unless~~ the compiler is doing the reading, in which case it is arranged that *foo* will be evaluated when the file of compiled code is loaded. This, therefore, performs a "load-time" evaluation of *foo*. By contrast, `#.` (see above) performs a "read-time" evaluation. In a sense, `#,` is like specifying `(eval load)` to `eval-when` (page EVAL-WHEN-FUN), while `#.` is more like specifying `(eval compile)`. It makes no difference when loading interpreted code, but when code is to be compiled, `#.` specifies compile-time evaluation and `#,` specifies load-time evaluation.

and cannot be saved in files of compiled code

`#,` allows you, for example, to include in your code complex list-structure constants which cannot be written with `quote`. Note that the reader does not put `quote` around the result of the evaluation. You must do this yourself if you want it, typically by using the `'` macro-character. An example of a case where you do not want `quote` around it is when this object is an element of a constant list.

#O `#o`*number* reads *number* in octal (radix 8).

#X `#x`*number* reads *number* in hexadecimal (radix 16). The digits above `9` are the letters `A` through `F` (the lower-case letters `a` through `f` are also acceptable).

CASE INCONSISTENCY IS CONFUSING.

#R `#`*radix*`r`*number* reads *number* in radix *radix*. *radix* must consist of only digits, and it is read in decimal.

For example, `#3r102` is another way of writing `11`, and `#11R32` is another way of writing `35`. For radices larger than 10, letters of the alphabet are used in order for the digits after `9`.

#s The syntax `#s(`*name slot1 value1 slot2 value2* `...)` denotes a structure. This is legal only if *name* is the name of a structure already defined by `defstruct` (page 185), and if the structure has a standard constructor macro, which it normally will. If it is assumed that the name of the constructor macro is `make-`*name* (which it normally is), then this syntax is equivalent to

`(make-`*name slot1* `'`*value1 slot2* `'`*value2* `...)`

That is, the constructor macro is called, with the specified slots having the specified values (note that one does not write quote-marks in the `#s` syntax). Whatever object the constructor macro returns is returned by the `#s` syntax.

It sounds like this works regardless of the name of the constructor macro. If so, say so more explicitly. If not also say

#: The syntax `#`*n*`:`*object* reads as whatever LISP object has *object* as its printed representation. However, that object is labelled by *n*, a required unsigned decimal integer, for possible reference by the syntax `#`*n*`#` (below).

??? Query: Resolve when the labels get reset.

Compatibility note: MACLISP is currently using `#:` for some bootstrapping purpose related to package syntax. The purpose described here does not conflict; they can be distinguished by the presense or absence of an infix number *n*. Presumably the package usage will eventually be phased out.

Lisp machine will be using name #: name as a package syntax. Doesn't conflict since syntax inside toke and othe delim

The syntax `#`*n*`#`, where *n* is a required unsigned decimal integer, serves as a reference to some object labelled by `#`*n*`:`; that is, `#`*n*`#` represents a pointer to the same identical (`eq`) object labelled by `#`*n*`:`. This permits notation of structures with shared or circular substructure. For example, a structure created in the variable `y` by this code:

```
(setq x (list 'p 'q))
(setq y (list (list 'a 'b) x 'foo x))
(rplacd (last x) (cdr x))
```

could be represented in this way:

equals or colon?

```
((a b) . #1=(#2=(p q) foo #2# . #1#))
```

Without this notation, but with `prinlength` (page PRINLENGTH-VAR) set to `10`, the structure would print in this way:

```
((a b) (p q) foo (p q) (p q) foo (p q) (p q) foo (p q) ...)
```

Compatibility note: In Lisp Machine LISP, the `##` syntax is used for an obsolete version of character syntax which has been flushed long ago.

is superseded in Lisp Machine LISP by `#/` and here by `#\`.

#+ The `#+` syntax provides a read-time conditionalization facility. The general syntax is "`#+`*feature form*". If *feature* is "true", then this syntax represents a LISP object whose printed representation is *form*. If *feature* is "false", then this syntax is effectively whitespace; it is as if it did not appear.

The *feature* should be the printed representation of a symbol or list. If *feature* is a symbol, then it is true iff it is a member of the list which is the value of the global variable `features` (page FEATURES-VAR). --- continuing the campaign to use up all english words for system use. Call it something else.

> Compatibility note: MACLISP uses the `status` special form for this purpose, and Lisp Machine LISP duplicates `status` essentially only for the sake of `(status features)`. The use of a variable allows one to bind the features list, for example when compiling. Not only for that reason

Otherwise, *feature* should be a boolean expression composed of `and`, `or`, and `not` operators on (recursive) *feature* expressions.

For example, suppose that in implementation A the features `spice` and `perq` are true, and in implementation B the feature `lispm` is true. Then the expressions on the left below are read the same as those on the right in implementation A:

```
(cons #+spice "Spice" #+lispm "Lispm" x)   (cons "Spice" x)
(setq a '(1 2 #+perq 43 #+(not perq) 27))  (setq a '(1 2 43))
(let ((a 3) #+(or spice lispm) (b 3))      (let ((a 3) (b 3))
   (foo a))                                   (foo a))
```

In implementation B, however, they are read in this way:

```
(cons #+spice "Spice" #+lispm "Lispm" x)   (cons "Lispm" x)
(setq a '(1 2 #+perq 43 #+(not perq) 27))  (setq a '(1 2 27))
(let ((a 3) #+(or spice lispm) (b 3))      (let ((a 3) (b 3))
   (foo a))                                   (foo a))
```

The `#+` construction must be used judiciously if unreadable code is not to result. The user should make a careful choice between read-time conditionalization and run-time conditionalization. See the macros named `if-for-spice` (page IF-FOR-SPICE-FUN), `if-in-spice` (page IF-IN-SPICE-FUN), and so on. ——— ??

#- `#-`*feature form* is equivalent to `#+(not` *feature*`)` *form*.

#< This is not legal reader syntax. It is used in the printed representation of objects which cannot be read back in. Attempting to read a `#<` will cause an error. (More precisely, it *is* legal syntax, but the macro-character function for it signals an error.)

#<space>, #<tab>, #<return>, #<form>

A `#` followed by a standard whitespace character is not legal reader syntax. This is so that abbreviated forms produced via `prinlevel` (page PRINLEVEL-VAR) cutoff will not read in again; this serves as a safeguard against losing information. (More precisely, it *is* legal syntax, but the macro-character function for it signals an error.)

#) This is not legal reader syntax. This is so that abbreviated forms produced via `prinlevel` (page PRINLEVEL-VAR) cutoff will not read in again; this serves as a safeguard against losing information. (More precisely, it *is* legal syntax, but the macro-character function for it signals an error.)

19.1.3. The Readtable

Previous sections have described the standard syntax accepted by the `read` function. This section discusses the advanced topic of altering the standard syntax, either to provide extended syntax for LISP objects or to aid the writing of other parsers.

There is a data structure called the *readtable* which is used to control the reader. It contains information about the syntax of each character. Initially it is set up to give the standard LISP meanings to all the characters, but the user can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the symbol `readtable`.

`readtable` [*Variable*]

The value of `readtable` is the current readtable. The initial value of this is a readtable set up for standard LISP syntax. You can bind this variable to temporarily change the readtable being used.

To program the reader for a different syntax, a set of functions are provided for manipulating readtables. Normally, you should begin with a copy of the standard COMMON LISP readtable and then customize the individual characters within that copy.

`copy-readtable &optional` *from-readtable to-readtable* [*Function*]

A copy is made of *from-readtable*, which defaults to the current readtable (the value of the global variable `readtable`). If *from-readtable* is `()`, then a copy of a standard COMMON LISP readtable is made; for example,

```
(setq readtable (copy-readtable ()))
```

will restore the input syntax to standard COMMON LISP syntax, even if the original readtable has been clobbered (assuming it is not so badly clobbered that you cannot type in the above expression!).

If *to-readtable* is unsupplied or `()`, a fresh copy is made. Otherwise *to-readtable* must be a readtable, which is clobbered with the copy.

`set-syntax-from-char` *to-char from-char* `&optional` *to-readtable from-readtable* [*Function*]

Makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*. The *to-readtable* defaults to the current readtable (the value of the global variable `readtable` (page 207)), and *from-readtable* defaults to `()`, meaning to use the syntaxes from the standard LISP readtable.

If the definition of an ordinary character is copied, any special attributes it might have within a symbol or number are copied with it. The attributes in the standard readtable are shown in Table

<tab>	whitespace
<form>	whitespace
<return>	whitespace
<space>	whitespace
!	alphabetic
"	macro character
#	macro character
$	alphabetic
%	alphabetic
&	alphabetic
'	macro character
(	macro character
)	macro character
*	alphabetic
+	alphabetic, plus sign
,	macro character
-	alphabetic, minus sign
.	alphabetic
/	alphabetic, ratio marker
0	digit
1	digit
2	digit
3	digit
4	digit
5	digit
6	digit
7	digit
8	digit
9	digit
:	package marker
;	macro character
<	alphabetic
=	alphabetic
>	alphabetic
|	macro character
<rubout>	ignored
{	alphabetic
}	alphabetic
`	macro character
@	alphabetic
A, a	alphabetic, digit>10
B, b	alphabetic, digit>10, bigfloat exponent
C, c	alphabetic, digit>10
D, d	alphabetic, digit>10, double-float exponent
E, e	alphabetic, digit>10, float exponent
F, f	alphabetic, digit>10, single-float exponent
G, g	alphabetic, digit>10
H, h	alphabetic, digit>10
I, i	alphabetic, digit>10
J, j	alphabetic, digit>10
K, k	alphabetic, digit>10
L, l	alphabetic, digit>10, long-float exponent
M, m	alphabetic, digit>10
N, n	alphabetic, digit>10
O, o	alphabetic, digit>10
P, p	alphabetic, digit>10
Q, q	alphabetic, digit>10
R, r	alphabetic, digit>10
S, s	alphabetic, digit>10, short-float exponent
T, t	alphabetic, digit>10
U, u	alphabetic, digit>10
V, v	alphabetic, digit>10
W, w	alphabetic, digit>10
X, x	alphabetic, digit>10
Y, y	alphabetic, digit>10
Z, z	alphabetic, digit>10
[	alphabetic
\	escape character
]	alphabetic
↑	alphabetic
~	alphabetic

Table 19-3: Standard Readtable Character Attributes

19-3. For example, if the definition of "S" is copied to "*", then "*" will be useable not only as an alphabetic character but as an exponent indicator in short-format floating-point number syntax.

> **Compatibility note:** No provision is made here for specifying the syntax attributes directly, as by keywords. It is more intuitive for the user simply to copy some standard character, and I believe that all the useful syntaxes are already provided in the standard readtable shown in Table 19-3. I AGREE.

BUT NON-EXTENSIBLE ARE M
IF NEW SYNTAXES — THERE ARE APPLICA
PROGRAM WHICH

It "works" to copy a macro definition from a character such as "|" to another character; the standard definition for "`|`" looks for another character which is the same as the character which invoked it. It doesn't "work" to copy the definition of "`(`" to "`{`", for example; it does work, but lets one write lists in the form "`{a b c)`", not "`{a b c}`", because the definition always looks for a closing "`)`". See the function `read-delimited-list` (page 212), which is useful in this connection.

`set-macro-character` *char function* `&optional` *readtable* [*Function*]
`get-macro-character` *char* `&optional` *readtable* [*Function*]

`set-macro-character` causes *char* to be a macro character which when seen by `read` causes *function* to be called. `get-macro-character` returns the function associated with *char*, or `()` if *char* does not have macro-character syntax. In each case, *readtable* defaults to the current readtable.

Different args. Splicing flushed.

function is called with two arguments, *stream* and *char*. The *stream* is the input stream, and *char* is the macro-character itself. In the simplest case, *function* may return a LISP object. This object is taken to be that whose printed representation was the macro character and any following characters read by the *function*. As an example, a plausible definition of the standard single-quote character is:

```
(defun single-quote-reader (stream ignore)
  (list 'quote (read stream)))
(set-macro-character #\' #'single-quote-reader)
```

No # here, don't want to fsymeval, rather store the symbol (so can be redefined) Rig

The function reads an object following the single-quote and returns a list of the symbol `quote` and that object. The *char* argument is ignored.

The function may choose instead to return *zero* values (for example, by using `(values)` as the return expression). In this case the macro character and whatever it may have read contribute nothing to the object being read. As an example, here is a plausible definition for the standard semicolon (comment) character:

```
(defun semicolon-reader (stream ignore)
  (do () ((char= (inch stream) #\Return)))
  (values))
(set-macro-character #\; #'semicolon-reader)
```

As another example, here is a simplified definition of the #+ syntax, which omits handling of `and`, `or` and `not`:

```
(defun sharp-plus-reader (stream ignore)
  (let ((feature (read stream))
        (object (read stream)))
    (if (memq feature features) object (values))))
(set-dispatch-macro-character #\# #\+ #'sharp-plus-reader)
```

If the feature is present, then `object` is returned, and otherwise nothing.

The *function* should not have any side-effects other than on the *stream* and *list-so-far*. Front ends (such as editors and rubout handlers) to the reader may cause *function* to be called repeatedly during the reading of a single expression in which the macro character only appears once, because of backtracking and restarting of the `read` operation.

`make-dispatch-macro-character` *char* `&optional` *readtable* [*Function*]

This causes the character *char* to be a dispatching macro character in *readtable* (which defaults to the current readtable). Initially every character in the dispatch table has a character-macro function which signals an error. Use `set-dispatch-macro-character` to define entries in the dispatch table.

`set-dispatch-macro-character` *disp-char sub-char function* `&optional` *readtable* [*Function*]
`get-dispatch-macro-character` *disp-char sub-char* `&optional` *readtable* [*Function*]

`set-dispatch-macro-character` causes *function* to be called when the *disp-char* followed by *sub-char* is read. The *readtable* defaults to the current readtable. The arguments and return values for *function* are the same as for normal macro characters, documented above under `set-macro-character` (page 213), except that *function* gets *sub-char* as its second argument, and also receives a third argument which is the non-negative integer whose decimal representation appeared between *disp-char* and *sub-char*, or `()` if there was none. The *sub-char* may not be one of the ten decimal digits: they are always reserved for specifying an infix integer argument.

`get-dispatch-macro-character` returns the macro-character function for *sub-char* under *disp-char*.

As an example, suppose one would like #$*foo* to be read as if it were `(dollars` *foo*`)`. One might say:

```
(defun sharp-dollar-reader (stream ignore ignore)
  (list 'dollars (read stream)))
(set-dispatch-macro-character #\# #\$ #'sharp-dollar-reader)
```

Compatibility note: This macro-character mechanism is different from those in MACLISP, INTERLISP, and Lisp Machine LISP. Recently LISP systems have implemented very general readers, even readers so programmable that they can parse arbitrary compiled BNF grammars. Unfortunately, these readers can be complicated to use, and have suffered from performance problems. This design is an attempt to make the reader as *simple* as possible to understand, use, and implement. Splicing macros have been eliminated; a recent informal poll indicates that no one uses them to produce other than zero or one value. The ability to access parts of the object preceding the macro character have been eliminated. The single-character-object feature has been eliminated, because it is seldom used and trivially obtainable by defining a macro.

Sounds like misinformation

The user is encouraged to turn off most macro characters, turn others into single-character-object macros, and then use `read` purely as a lexical analyzer on top of which to build a parser. It is unnecessary, however, to cater to more complex lexical analysis or parsing than that needed for COMMON LISP.

19.1.4. What the `print` Function Produces

(missing)

19.2. Input Functions

19.2.1. Input from ASCII Streams

Many input functions take optional arguments called *input-stream* and *eof-option*. The *input-stream* argument is the stream from which to obtain input; if unsupplied or `()` it defaults to the value of the special variable `standard-input` (page 231). One may also specify `t` as a stream, meaning the value of the special variable `terminal-io` (page 232).

> **Rationale:** Allowing the use of `t` provides some semblance of MACLISP compatibility.

Hardly anything else in here is compatible with Maclisp; why bother? (Admittedly the Lisp machine does it

The *eof-option* argument controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If no *eof-option* argument is supplied, an error will be signalled at end of file. If there is an *eof-option*, it is the value to be returned. Note that an *eof-option* of `()` means to return `()` if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*. The *eof-option* argument is always evaluated; the resulting value is used, however, only when end of file is encountered.

Functions such as `read` (page 211) which read an "object" rather than a single character will always signal an error, regardless of *eof-option*, if the file ends in the middle of an object. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, `read` will complain. If a file ends in a symbol or a number immediately followed by end-of-file, `read` will read the symbol or number successfully and when called again will see the end-of-file and obey *eof-option*. Similarly, the function `readline` (page 213) will successfully read the last line of a file even if that line is terminated by end-of-file rather than the newline character. If a file contains ignorable text at the end, such as blank lines and comments, `read` will not consider it to end in the middle of an object and will obey *eof-option*. GOOD!

> **Compatibility note:**
>
> These end-of-file conventions are compatible with Lisp Machine LISP, but not completely compatible with Maclisp. Maclisp's deviations from this are generally considered to be bugs rather than features.
>
> The MACLISP "feature" of letting *input-stream* and *eof-option* appear in either order is not supported. Good. HURRAY!

Note that all of these functions will echo their input if used on an interactive stream. The functions that input more than one character at a time allow the input to be edited. The function `inchpeek` (page 214) echoes all of the characters that are skipped over (if any) if `inch` would have echoed them; the character not removed from the stream is not echoed either.

`read &optional` *input-stream eof-option* [*Function*]

`read` reads in the printed representation of a LISP object from *input-stream*, builds a corresponding LISP object, and returns the object. The details are explained above.

`read-preserve-delimiters` [*Variable*]

Certain printed representations given to `read`, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the close parenthesis marks the end of the list.) Normally `read` will throw away the delimiting character if it is a white-space character, but will preserve it (using `untyi` (page 213)) if the character is syntactically meaningful, since it may be the start of the next expression.

The variable `read-preserve-delimiters` controls this throwing away of white-space characters. Its normal value is `()`, but if its value is not `()` then no delimiting characters will be thrown away by `read`, even if they are whitespace. This may be useful for certain macro characters or special input syntaxes.

As an example, consider this macro-character definition:

```
(defun slash-reader (stream ignore)
  (let ((read-preserve-delimiters t))
    (do ((path (list (read stream))
               (cons (progn (inch stream) (read stream))
                     path)))
        ((not (char= (inchpeek stream) #\/))
         (cons 'pathname (nreverse path))))))
(set-macro-character #\/ #'slash-reader)
```

Consider now calling `read` on this expression:

```
(zyedh /usr/games/zork /usr/games/boggle)
```

The "`/`" macro reads objects separated by more "`/`" characters; thus `/usr/games/zork` is intended to read as `(pathname usr games zork)`. The entire exmaple expression should therefore be read as

```
(zyedh (pathname usr games zork) (pathname usr games boggle))
```

However, if `read` were not instructed by the binding of the variable `read-preserve-delimiters` to preserve whitespace, then on reading the symbol `zork`, the following space would be discarded, and then the next call to `inchpeek` would see the following "`/`", and the loop would continue, producing this interpretation:

```
(zyedh (pathname usr games zork usr games boggle))
```

On the other hand, there are times when whitespace *should* be discarded. If one has a command interpreter which takes single-character commands, but occasionally reads a LISP object, then if the whitespace after a symbol were not discarded it might be interpreted as a command some time later after the symbol had been read.

I don't ~~thi~~ this argument makes sense since eof will always be in the middle of an object.

`read-delimited-list` *char* `&optional` *input-stream eof-option* [*Function*]

This reads objects from *stream* until the next character after an object's representation (ignoring whitespace and ignored characters) is *char*. (The *char* should not have whitespace or ignored syntax in the current readtable.) A list of the objects read is returned.

This function is particularly useful for defining new macro-characters. Suppose one were to want "`#{`*a b c* `...` *z*`}`" to read as a list of all pairs of the elements *a, b, c, . . . , z*; for example:

`#{p q z a}` read as `((p q) (p z) (p a) (q z) (q a) (z a))`

This can be done by specifying a macro-character definition for "`#{`" which does two things: read in all the items up to the "`}`", and construct the pairs. `read-delimited-list` performs the first task.

```
(defun sharp-leftbrace-reader (stream ignore ignore)
  (mapcon #'(lambda (x)
              (mapcar #'(lambda (y) (list x y)) (cdr x)))
          (read-delimited-list #\} stream)))
(set-dispatch-macro-character #\# #\{ #'sharp-leftbrace-reader)
```

`readline &optional` *input-stream eof-option* [*Function*]

`readline` reads in a line of text, terminated by a carriage return. It returns the line as a character string, *without* the return character. This function is usually used to get a line of input from the user. A second returned value is a flag which is `()` if a carriage return terminated the line, or `t` if end-of-file terminated the (non-empty) line. BUT THE RETURN IS ECHOED.

`inch &optional` *input-stream eof-option* [*Function*]
`tyi &optional` *input-stream eof-option* [*Function*]

`inch` inputs one character from *input-stream* and returns it as a character object. The character is echoed if *input-stream* is interactive.

`tyi` is similar to `inch`, but returns the character as an integer; it is as if `inch` were used, and `char-int` (page 103) applied to the result.

It is almost always preferable to use `inch` rather than `tyi`, if only for reasons of portability.

`uninch` *character* `&optional` *input-stream* [*Function*]
`untyi` *integer* `&optional` *input-stream* [*Function*]

`uninch` puts the *character* onto the front of *input-stream*. When a character is next read from *input-stream*, it will be the specified character, followed by the previous contents of *input-stream*. `uninch` returns `()`.

`untyi` is similar to `uninch`, but takes an integer rather than a character object. it is as if `uninch` were used after applying `int-char` (page 103) to the first argument.

It is almost always preferable to use `uninch` rather than `untyi`, if only for reasons of portability.

> ??? Query: This from the Lisp Machine LISP Manual: "Note that you are only allowed to `untyi` one character before doing a `tyi`, and you aren't allowed to `untyi` a different character than [*sic*] the last character you read from the stream. Some streams implement `untyi` by saving the character, while others implement it by backing up the pointer to a buffer." Opinions? Current trend is to opt for generality rather than hacks.

? This is to allow freedom of implementation. Untyi is not a mechanism to put input into a stream through the side door.

`inchpeek &optional` *peek-type input-stream eof-option* [*Function*]
`tyipeek &optional` *peek-type input-stream eof-option* [*Function*]

What `inchpeek` does depends on the *peek-type*, which defaults to `()`. With a *peek-type* of `()`, `inchpeek` returns the next character to be read from *input-stream*, without actually removing it from the input stream. The next time input is done from *input-stream* the character will still be there. It is as if one had called `inch` and then `uninch` in succession.

If *peek-type* is `t`, then `inchpeek` skips over whitespace and ignored characters, and then performs the peeking operation on the next character. This is useful for finding the (possible) beginning of the next printed representation of a Lisp object. As above, the last character (the one that starts an object) is not removed from the input stream.

If *peek-type* is a character object, then `inchpeek` skips over input characters until a character which is `char=` (page 100) to that object is found; that character is left in the input stream.

Characters passed over by `inchpeek` are echoed if *input-stream* is interactive.

`tyipeek` is similar to `inchpeek`, but returns an integer rather than a character object; it is as if `inchpeek` were used, and `char-int` (page 103) applied to the result. (If, however, an *eof-option* is provided and returned, `char-int` is not applied!) `tyipeek` also requires an integer instead of a character as the *peek-type*.

It is almost always preferable to use `inchpeek` rather than `tyipeek`, if only for reasons of portability. WHAT DOES THIS DO W.R.T. ECHOING

`listen &optional` *input-stream* [*Function*]

The function `listen` returns `t` if there is a character immediately available from *input-stream*, and `()` if not. This is particularly useful when the stream obtains characters from an interactive device such as a keyboard; a call to `inch` (page 217) would simply wait until a character was available, but `listen` can sense whether or not input is available and allow the program to decide whether or not to attempt input. On a non-interactive stream, the general rule is that `listen` returns `t` except when at end-of-file.

`inch-no-hang &optional` *input-stream eof-option* [*Function*]
`tyi-no-hang &optional` *input-stream eof-option* [*Function*]

These functions are exactly like `inch` (page 217) and `tyi` (page 217), except that if it would be necessary to wait in order to get a character (as from a keyboard), `()` is immediately returned without waiting. This allows one efficiently to check for input being available and get the input if it is. This is different from the `listen` (page 214) operation in two ways. First, these functions potentially actually read a character, while `listen` never inputs a character. Second, `listen` does not distinguish between end-of-file and no input being available, while these functions do make that distinction, returning *eof-option* at end-of-file, but always returning `()` if no input is available.

Erring if eof-option not supplied?

`clear-input &optional` *input-stream* [*Function*]

This clears any buffered input associated with *input-stream*. It is primarily useful for clearing type-ahead from keyboards when some kind of asynchronous error has occurred. If this operation doesn't make sense for the stream involved, when `clear-input` does nothing. `clear-input` returns `()`.

SHOULDN'T THIS BE START, END LIKE EVERYTHING ELSE?

THIS IS MY FAULT, BUT THAT'S NO EXCU -MOO

`read-from-string` *string* `&optional` *eof-option* `(`*index* `0)` [*Function*]

The characters of *string* are given successively to the LISP reader, and the LISP object built by the reader is returned. Macro characters and so on will all take effect.

The *eof-option* is what to return if the end of the string is reached, as with other reading functions.

??? Query: In Lisp Machine LISP, what happens if end-of-string occurs in the middle of an object? An error, of cours

The argument *index* is the index in the string of the first character to be read; by default the entire

There should also be
parse-number string &optional (from 0) to (radix 10.)
returns number or NIL if no number there.
and 2nd value is next character position
(index of delimiter)

Should there be a flaw
of this which errors
the specified range
exactly a number
than return
or sto

string is used.

`read-from-string` returns two values; the first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this will be either the length of the string or one greater than the length of the string. The variable `read-preserve-delimiters` (page 215) affects this second value. Use ≥ to test it.

For example:

```
(read-from-string "(a b c)") => (a b c) and 7
```

19.2.2. Input from Binary Streams

(missing)

19.2.3. Input Editing

(missing)

19.3. Output Functions

19.3.1. Output to ASCII Streams

These functions all take an optional argument called *output-stream*, which is where to send the output. If unsupplied or `()`, *output-stream* defaults to the value of the variable `standard-output` (page 231). If it is `t`, the value of the variable `terminal-io` (page 232) is used.

`prin1` *object* `&optional` *output-stream* [*Function*]
`print` *object* `&optional` *output-stream* [*Function*]
`princ` *object* `&optional` *output-stream* [*Function*]

`prin1` outputs the printed representation of *object* to *output-stream*, using escape characters. As a rule, the output from `prin1` is suitable for input to the function `read` (page 215); see ???. `prin1` returns `t`.

`print` is just like `prin1` except that the printed representation of *object* is preceded by a carriage return and followed by a space. `print` returns `t`.

`princ` is just like `prin1` except that the output has no escape characters. A symbol is printed as simply the characters of its print-name; a string is printed without surrounding double-quotes; and there may be differences for other data types as well. The general rule is that output from `princ` is intended to look good to people, while output from `prin1` is intended to be acceptable to the function `read` (page 215). `princ` returns `t`.

The output from these functions is affected by the values of the variables `base` (page BASE-VAR), `prinlevel` (page PRINLEVEL-VAR), and `prinlength` (page PRINLENGTH-VAR).

I want these to return what they print. What is the rationale for the change?

*no point ?
Or is it gone?
(Seems desirable to h
when back-porting

`ouch` *character* `&optional` *output-stream* [*Function*]
`tyo` *integer* `&optional` *output-stream* [*Function*]

`ouch` outputs the *character* to *output-stream*.

`tyo` is similar, but takes an integer instead of a character; it is as if `int-char` were applied to the first argument and then `ouch` were called.

It is almost always preferable to use `ouch` rather than `tyo`, if only for reasons of portability.

Both functions return `t`.

`terpri` `&optional` *output-stream* [*Function*]
`fresh-line` `&optional` *output-stream* [*Function*]

`terpri` outputs a newline to *output-stream*; this may be simply a carriage-return character, a return-linefeed sequence, or whatever else is appropriate for the stream. `terpri` returns `t`.

`fresh-line` is similar to `terpri`, but outputs a newline only if the stream is not already at the start of a line. (If for some reason this cannot be determined, then a newline is output anyway.) This guarantees that the stream will be on a "fresh line" while consuming as little vertical distance as possible. `fresh-line` returns `t` if it output a newline, and otherwise returns `()`.

`string-out` *string* `&optional` *output-stream* *start* *end* [*Function*]
`line-out` *string* `&optional` *output-stream* *start* *end* [*Function*]

← These 2 seem totally unnecessary.

The characters in the argument *string*, which must be a string (not a symbol), are output to the *output-stream*. The optional arguments *start* and *end* specify a substring of *string* to be output; *start* is the index of the first character to output, and *end* is an index one greater than the last character to be output. These default to the beginning and end of the string.

`string-out` simply puts out the specified characters; `line-out` additionally outputs a newline afterwards. Each function returns `t`.

`force-output` `&optional` *output-stream* [*Function*]
`clear-output` `&optional` *output-stream* [*Function*]

Some streams may be implemented in an asynchronous or buffered manner. The function `force-output` attempts to ensure that all output sent to *output-stream* has reached its destination, and only then returns `t`.

The function `clear-output`, on the other hand, attempts to abort any outstanding output operation in progress, to allow as little output as possible to continue to the destination. This is useful, for example, to abort a lengthy output to the terminal when an asynchronous error occurs. `clear-output` returns `t`.

The function `format` (page 217) is very useful for producing nicely formatted text. It can do anything any of the above functions can do, and it makes it easy to produce good looking messages and such. `format` can generate a string or output to a stream.

The function `pprint` (page PPRINT-FUN) is useful for printing LISP objects "prettily" in an indented format. Also, `grindef` (page GRINDEF-FUN) is useful for formatting LISP programs.

19.3.2. Output to Binary Streams

(missing)

19.4. Formatted Output

args (or change text)

`format` *destination control-string* `&rest` *arguments* [*Function*]

`format` is used to produce formatted output. `format` outputs the characters of *control-string*, except that a tilde ("~") introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of *args* to create their output; the typical directive puts the next element of *args* into the output, formatted in some special way.

The output is sent to *destination*. If *destination* is `()`, a string is created which contains the output; this string is returned as the value of the call to `format`. In all other cases `format` returns `t`, performing output to *destination* as a side effect. If *destination* is a stream, the output is sent to it. If *destination* is `t`, the output is sent to the stream which is the value of the variable `standard-output` (page 231).

A `format` directive consists of a tilde ("~"), optional prefix parameters separated by commas, optional colon (":") and atsign ("@") modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the directive character is ignored. The prefix parameters are generally decimal numbers. Examples of control strings:

```
"~S"          ;This is an S directive with no parameters or modifiers.
"~3,4:@s"     ;This is an S directive with two parameters, 3 and 4,
              ;    and both the colon and atsign flags.
"~,4S"        ;Here the first prefix parameter is omitted and takes
              ;    on its default value, while the second parameter is 4.
```

The `format` function includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use `format` effectively. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (" ' ") followed by the desired character may be used as a prefix parameter, so that you don't have to know the decimal numeric values of characters in the character set. For example, you can use `"~5,'0d` to print a decimal number in five columns with leading zeros, or `"~5,'*d"` to get leading asterisks.

In place of a prefix parameter to a directive, you can put the letter "V", which takes an argument from *arguments* as a parameter to the directive. Normally this should be an integer (but in general it doesn't really have to be). This feature allows variable column-widths and the like. Also, you can use the character "#" in place of a parameter; it represents the number of arguments remaining to be processed.

Here are some relatively simple examples to give you the general flavor of how `format` is used.

```
(format () "foo") => "foo"
(setq x 5)
(format () "The answer is ~D." x) => "The answer is 5."
(format () "The answer is ~3D." x) => "The answer is   5."
(format () "The answer is ~3,'0D." x) => "The answer is 005."
(format () "The answer is ~:D." (expt 47 x))
                                => "The answer is 229,345,007."
(setq y "elephant")
(format () "Look at the ~A!" y) => "Look at the elephant!"
(format () "Type ~:C to ~A." (control #\D) "delete all your files")
      => "Type Control-D to delete all your files."
(setq n 3)
(format () "~D item~:P found." n) => "3 items found."
(format () "~R dog~:[s are~; is~] here." n (= n 1))
      => "three dogs are here."
(format () "~R dog~:*~[~1; is~:;s are~] here." n)
      => "three dogs are here."
(format () "Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n)
      => "Here are three puppies."
```

The directives will now be described. The term *arg* in general refers to the next item of the set of *arguments* to be processed. The word or phrase at the beginning of each description is a mnemonic word for the directive.

~A	*Ascii.* An *arg*, any LISP object, is printed without escape characters (as by `princ` (page 219)). In particular, if *arg* is a string, its characters will be output verbatim. **Compatibility note:** In COMMON LISP, the empty list always prints as `()`, so the colon modifier is useless here. In Lisp Machine LISP it specifies whether to print it as `()` or as `NIL`. ~*mincol*A inserts spaces on the right, if necessary, to make the width at least *mincol* columns. The @ modifier causes the spaces to be inserted on the left rather than the right. ~*mincol*, *colinc*, *minpad*, *padchar*A is the full form of ~A, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and the space character for *padchar*.
~S	*S-expression.* This is just like ~A, but *arg* is printed *with* escape characters (as by `prin1` (page 219) rather than `princ`). The output is therefore suitable for input to `read` (page 215).

~D *Decimal.* An *arg*, a which should be an integer, is printed in decimal radix. ~D will never put a decimal point after the number. but don't allow reader to read this as "must"

~*mincol*D uses a column width of *mincol*; spaces are inserted on the left if the number requires fewer than *mincol* columns for its digits and sign. If the number doesn't fit in *mincol* columns, additional columns are used as needed.

~*mincol*,*padchar*D uses *padchar* as the pad character instead of space.

If *arg* is not an integer, it is printed in ~A format and decimal base.

The @ modifier causes the number's sign to be printed always; the default is only to print it if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter may be used to change the character used as the comma. Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*D.

~O *Octal.* This is just like ~D but prints in octal radix instead of decimal. The full form is therefore ~*mincol*,*padchar*,*commachar*O.

~R *Radix.* ~*n*R prints *arg* in radix *n*. The modifier flags and any remaining parameters are used as for the ~D directive. Indeed, ~D is the same as ~10R. The full form here is therefore ~*radix*,*mincol*,*padchar*,*commachar*R.

If no arguments are given to ~R, then an entirely different interpretation is given. The argument should be an integer; suppose it is 4.

- ~R prints *arg* as a cardinal English number: "four".
- ~:R prints *arg* as an ordinal English number: "fourth".
- ~@R prints *arg* as a Roman numeral: "IV".
- ~:@R prints *arg* as an old Roman numeral: "IIII".

~P *Plural.* If *arg* is not 1, a lower-case "s" is printed; if *arg* is 1, nothing is printed.

??? Query: Should this work for floating-point numbers also?

i.e. should it use EQUAL, or EQUA
Usually one says "1.0 dogs" ra
Than "1
But I
care

~:P does the same thing, after doing a ~:* to back up one argument; that is, it prints a lower-case "s" if the *last* argument was not 1. This is useful after printing a number using ~D.

~@P prints "y" if the argument is 1, or "ies" if it is not. ~:@P does the same thing, but backs up first.

```
(format () "~D tr~:@P/~D win~:P" 7 1) => "7 tries/1 win"
(format () "~D tr~:@P/~D win~:P" 1 0) => "1 try/0 wins"
(format () "~D tr~:@P/~D win~:P" 1 3) => "1 try/3 wins"
```

~F *Floating-point.*

??? **Query:** Is this really what we want? N$

arg is printed in floating point. `~nF` rounds *arg* to a precision of *n* digits. The minimum value of *n* is 2, since a decimal point is always printed. If the magnitude of *arg* is too large or too small, it is printed in exponential notation. If *arg* is not a number, it is printed in `~A` format. Note that the prefix parameter *n* is not *mincol*; it is the number of digits of precision desired. Examples:

```
(format () "~2F" 5)  =>  "5.0"
(format () "~4F" 5)  =>  "5.0"
(format () "~4F" 1.5) => "1.5"
(format () "~4F" 3.14159265)  =>  "3.142"
(format () "~3F" 1e10)  =>  "1.0e10"
```

~E *Exponential.* This one is admittedly random.

??? **Query:** Is this the right thing Study PL/I, FORTRAN.

arg is printed in exponential notation. This is identical to `~F`, including the use of a prefix parameter to specify the number of digits, except that the number is always printed with a trailing exponent, even if it is within a reasonable range.

~C *Character.* The next *arg* should be a character; it is printed according to the modifier flags.

`~C` prints the character in an implementation-dependent abbreviated format. This format should be culturally compatible with the host environment.

Implementation note: In Lisp Machine LISP, the following format is used. If the character has any control bits set, and the output stream can represent the necessary Greek characters, then the control bits are output as alpha (α) for Control, beta (β) for Meta, lambda (λ) for Hyper, and pi (π) for Super. If the character itself is alpha, beta, lambda, pi, or equivalence-sign (≡), then it is preceded by an equivalence-sign to quote it. After all this, the base character itself is output.

In any case, with no shift bits, this simply outputs the character a la tyo.

Implementations which do not have Greek characters may well choose to represent control characters by initials and hyphens thus:

```
C-A            C-M-$            H-S-C-#
```

This has the advantage of staying within the standard character set.

ACTUALLY, I STRONGLY HOPE T
FLUSH THE STUPID GREEK LET
AND DO EXACTLY WHAT
YOU SUGGEST HERE!

`~:C` spells out the names of the control bits, and represents non-printing characters by their names: `"Control-Meta-F"`, `"Control-Return"`, `"Space"`. This is a "pretty" format for printing characters.

`~:@C` prints what `~:C` would, and then if the character requires unusual shift keys on the keyboard to type it, this fact is mentioned: `"Control-δ (Top-F)"`. This is the format used for telling the user about a key he is expected to type, for instance in prompt messages. The precise output may depend not only on the implementation, but on the particular I/O devices in use.

`~@C` prints the character in a way that the LISP reader can understand, using "#\" syntax.

Rationale: In some implementations the `~S` directive would accomplish this also, but the `~C` directive is compatible with LISP dialects which do not have a character data type.

~% Outputs a newline (see `terpri` (page 220)). `~n%` outputs *n* newlines. No *arg* is used. Simply putting a newline in the control string would work, but `~%` is often used because it makes the control string look nicer in the middle of a LISP program.

~& Unless the stream knows that it is already at the beginning of a line, this outputs a newline (see `fresh-line` (page 220)). *~n*& does a `:fresh-line` operation and then outputs *n − 1* newlines.

~| Outputs a page separator character, if possible. *~n*| does this *n* times. With a : modifier, if the output stream supports the `clear-screen` (page CLEAR-SCREEN-FUN) operation this directive clears the screen; otherwise it outputs page separator character(s) as if no : modifier were present. | is vertical bar, not capital I.

~~ *Tilde.* Outputs a tilde. *~n~* outputs *n* tildes.

~<newline> Tilde immediately followed by a newline ignores the newline and any following non-newline whitespace. With a :, the newline is ignored but the whitespace is left in place. With an @, the newline is left in place but the whitespace is ignored. This directive is typically used when a format control string is too long to fit nicely into one line of the program:

```
(defun pet-rock-warning (rock friend amount)
  (unless (equalp rock friend)
          (format t "~&Warning!  Your pet rock ~A just ~
                     bit your friend ~A,~%  and ~
                     ~:[he~;she~] is suing you for $~D!"
                  rock friend (femalep friend) amount)))
(pet-rock-warning "Fred" "Susan" 500)  prints:
Warning:  Your pet rock Fred just bit your friend Susan,
  and she is suing you for $500!
```

~X FORTRAN x *format.* Outputs a space. *~n*X outputs *n* spaces.

~T *Tabulate.* Spaces over to a given column. *~colnum*, *colinc*T will output sufficient spaces to move the cursor to column *colnum*. If the cursor is already past column *colnum*, it will output spaces to move it to column *colnum*+ *k***colinc*, for the smallest non-negative integer *k* possible. *colnum* and *colinc* default to 1.

~:T is like ~T, but *colnum* and *colinc* are in units of pixels, not characters; this makes sense only for streams which can set the cursor position in pixel units.

If for some reason the current column position cannot be determined or set, any ~T operation will simply output two spaces. When `format` is creating a string, ~T will work, assuming that the first character in the string is at the left margin (column 0).

~* The next *arg* is ignored. *~n** ignores the next *n* arguments.

~:* "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. *~n*:* backs up *n* arguments.

When within a ~{ construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

This is a "relative goto"; for an "absolute goto", see `~G`.

`~nG` *Goto.* Goes to the *n*th *arg*, where 0 means the first one. Directives after a `~nG` will take arguments in sequence beginning with the one gone to.

When within a `~{` construct, the "goto" is relative to the list of arguments being processed by the iteration.

This is an "absolute goto"; for a "relative goto", see `~*`.

The format directives after this point are much more complicated than the foregoing; they constitute "control structures" which can perform conditional selection, iteration, justification, and non-local exits. Used with restraint, they can perform powerful tasks. Used with wild abandon, they can produce unreadable and unmaintainable spaghetti with goulash on top.

`~[`*str0*`~;`*str1*`~;`...`~;`*strn*`~]`

Conditional expression. This is a set of control strings, called *clauses*, one of which is chosen and used. The clauses are separated by `~;` and the construct is terminated by `~]`. For example,

```
"~[Siamese~;Manx~;Persian~;Tortoise-Shell~] Cat"
```

The *arg*th clause is selected, where the first clause is number 0. If a prefix parameter is given (as `~n[`), then the parameter is used instead of an argument (this is useful only if the parameter is specified by "`#`"). If *arg* is out of range then no clause is selected. After the selected alternative has been processed, the control string continues after the `~]`.

`~[`*str0*`~;`*str1*`~;`...`~;`*strn*`~:;`*default*`~]` has a default case. If the *last* "`~;`" used to separate clauses is instead "`~:;`", then the last clause is an "else" clause, which is performed if no other clause is selected. For example:

```
"~[Siamese~;Manx~;Persian~;Tortoise-Shell~:;Alley~] Cat"
```

`~[~`*tag00*`,`*tag01*`, . . . ;`*str0*`~`*tag10*`,`*tag11*`, . . . ;`*str1*` . . . ~]` allows the clauses to have explicit tags. The parameters to each `~;` are numeric tags for the clause which follows it. That clause is processed which has a tag matching the argument. If `~`*a1*`,`*a2*`,`*b1.*`,`*b2*`, . . . :;` (note the colon) is used, then the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, etc. `~:;` with no parameters may be used at the end to denote a default clause. For example:

```
"~[~'+,'-,'*,'/;operator ~'A,'Z,'a,'z:;letter ~
   ~'0,'9:;digit ~:;other ~]"
```

`~:[`*false*`~;`*true*`~]` selects the *false* control string if *arg* is `()`, and selects the *true* control string otherwise.

`~@[`*true*`~]` tests the argument. If it is not `()`, then the argument is not used up, but is the next one to be processed, and the one clause *true* is processed. If the *arg* is `()`, then the argument is used up, and the clause is not processed. The clause therefore should normally use exactly one (non-( )) argument. For example:

```
(setq prinlevel () prinlength 5)
(format () "~@[ PRINLEVEL=~D~]~@[ PRINLENGTH=~D~]"
           prinlevel prinlength)
   =>  " PRINLENGTH=5"
```

The combination of `~[` and `#` is useful, for example, for dealing with English conventions for printing lists:

```
(setq foo "Items:~#[ none~; ~S~; ~S and ~
          ~S~:;~@{~#[~1; and~] ~S~↑,~}~].")
(format () foo)
          =>  "Items: none."
(format () foo 'foo)
          =>  "Items: FOO."
(format () foo 'foo 'bar)
          =>  "Items: FOO and BAR."
(format () foo 'foo 'bar 'baz)
          =>  "Items: FOO, BAR, and BAZ."
(format () foo 'foo 'bar 'baz 'quux)
          =>  "Items: FOO, BAR, BAZ, and QUUX."
```

`~;` Separates clauses in `~[` and `~<` constructions. It is undefined elsewhere.

`~]` Terminates a `~[`. It is undefined elsewhere.

`~{`*str*`~}` *Iteration.* This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to `format`. The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes as arguments; if *str* uses up two arguments by itself, then two elements of the list will get used up each time around the loop. If before any iteration step the list is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there will be at most *n* repetitions of processing of *str*. Finally, the `~↑` directive can be used to terminate the iteration prematurely.

Here are some simple examples:

```
(format () "The winners are:~{ ~S~}." '(fred harry jill))
     => "The winners are: FRED HARRY JILL."
(format () "Pairs:~{ <~S,~S>~}." '(a 1 b 2 c 3))
     => "Pairs: <A,1> <B,2> <C,3>."
```

`~:{`*str*`~}` is similar, but the argument should be a list of sublists. At each repetition step one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
(format () "Pairs:~:{ <~S,~S>~}."
              '((a 1) (b 2) (c 3)))
     => "Pairs: <A,1> <B,2> <C,3>."
```

`~@{`*str*`~}` is similar to `~{`*str*`~}`, but instead of using one argument which is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
(format () "Pairs:~@{ <~S,~S>~}."
                'a 1 'b 2 'c 3)
      => "Pairs: <A,1> <B,2> <C,3>."
```

`~:@{`*str*`~}` combines the features of `~:{`*str*`~}` and `~@{`*str*`~}`. All the remaining arguments are used, and each one must be a list. On each iteration the next argument is used as a list of arguments to *str*. Example:

```
(format () "Pairs:~:@{ <~S,~S>~}."
                '(a 1) '(b 2) '(c 3))
      => "Pairs: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with `~:}` instead of `~}` forces *str* to be processed at least once even if the initial list of arguments is null (however, it will not override an explicit prefix parameter of zero).

If *str* is empty, then an argument is used as *str*. It must be a string, and precedes any arguments processed by the iteration. As an example, the following are equivalent:

```
(funcall* #'format stream string args)
(format stream "~1{~:}" string args)
```

This will use `string` as a formatting string. The `~1{` says it will be processed at most once, and the `~:}` says it will be processed at least once. Therefore it is processed exactly once, using `args` as the arguments.

As another (rather sophisticated) example, the `format` function itself uses `format-error` (a routine internal to the `format` package) to signal error messages, which in turn uses `ferror`, which uses `format` recursively. Now `format-error` takes a string and arguments, just like `format`, but also prints the control string to `format` (which at this point is available in the variable `ctl-string`) and a little arrow showing where in the processing of the control string the error occurred. The variable `ctl-index` points one character after the place of the error.

```
(defun format-error (string &rest args)
  (ferror () "~1{~:}~%~VT↓~%~3X/"~A/"~%"
             string args (+ ctl-index 3) ctl-string))
```

This first processes the given string and arguments using `~1{~:}`, then goes to a new line, tabs a variable amount for printing the down-arrow, and prints the control string between double-quotes. The effect is something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
>>ERROR: The argument to the FORMAT "~[" command
         must be a number.
                   ↓
   "The item is a ~[Foo~;Bar~;Loser~]."
...
```

`~}` Terminates a `~{`. It is undefined elsewhere.

`~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`<`*str*`~>`

Justification. This justifies the text produced by processing *str* within a field at least *mincol* columns wide. *str* may be divided up into segments with `~;`, in which case the spacing is evenly divided between the text segments.

With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified; if there is only one, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the @ modifier causes spacing to be added after the last. The *minpad* parameter (default 0) is the minimum number of padding characters to be output between each segment. The padding character is specified by *padchar*, which defaults to the space character. If the total width needed to satisfy these constraints is greater than *mincol*, then the width used is *mincol*+ *k***colinc* for the smallest possible non-negative integer value *k*; *colinc* defaults to 1, and *mincol* defaults to 0.

Examples:

```
(format () "~10<foo~;bar~>")              =>  "foo    bar"
(format () "~10:<foo~;bar~>")             =>  "  foo  bar"
(format () "~10:@<foo~;bar~>")            =>  "  foo bar "
(format () "~10<foobar~>")                =>  "    foobar"
(format () "~10:<foobar~>")               =>  "    foobar"
(format () "~10@<foobar~>")               =>  "foobar    "
(format () "~10:@<foobar~>")              =>  "  foobar  "
(format () "$~10,,,'*<~3F~>" 2.59023) =>  "$******2.59"
```

Note that *str* may include `format` directives. All the clauses in *str* are processed in order; it is the resulting pieces of text that are justified. The last example illustrates how the ~< directive can be combined with the ~F directive to provide more advanced control over the formatting of numbers.

??? Query: Unfortunatly, the ~F command as defined above isn't really flexible enough?

The ~↑ directive may be used to terminate processing of the clauses prematurely, in which case only the completely processed clauses are justified.

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~↑, of course), but the first one is not used in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a newline (such as a ~% directive). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the ~:; has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~↑,~}.~%"
```

can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ";; ". The prefix parameter 1 in ~1:; accounts for the width of the comma which will follow the justified item if it is not the last element in the list, or the period if it is. If ~:; has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream.

To make the preceding example use a line width of 50, one would write

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~↑,~}.~%"
```

If the second argument is not specified, then `format` uses the line width of the output stream. If this cannot be determined (for example, when producing a string result), then `format` uses 72 as the line length.

~> Terminates a ~<. It is undefined elsewhere.

~↑ *Up and out.* This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the ~< case, the formatting *is* performed, but no more segments are processed before doing the justification. The ~↑ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause it appears in (as well as all following clauses). ~↑ may appear anywhere in a ~{ construct.

```
(setq donestr "Done.~↑  ~D warning~:P.~↑  ~D error~:P.")
(format () donestr) => "Done."
(format () donestr 3) => "Done.  3 warnings."
(format () donestr 1 5) => "Done.  1 warning.  5 errors."
```

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~↑ is equivalent to ~#↑.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~↑ is used within a ~:{ construct, then it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use ~:↑.

Here are some examples of the use of ~↑ within a ~< construct.

```
(format () "~15<~S~;~↑~S~;~↑~S~>" 'foo)
        =>  "               FOO"
(format () "~15<~S~;~↑~S~;~↑~S~>" 'foo 'bar)
        =>  "FOO            BAR"
(format () "~15<~S~;~↑~S~;~↑~S~>" 'foo 'bar 'baz)
        =>  "FOO    BAR    BAZ"
```

Compatibility note: The ~Q directive and user-defined directives have been omitted here, as well as control lists (as opposed to strings), which are rumored to be changing in meaning.

I believe control lists have been flushed,
but a list means simply output the
string in its car without looking for
"~" -- can be passed to programs
which expect a format string and args, to
output
string ~
going t
format.
(An alterna
might be "~.
means output t
rest of the st
without looking

19.5. Querying the User

The following functions provide a convenient and consistent interface for asking questions of the user. Questions are printed and the answers are read using the stream `query-io`, which normally is synonymous with `terminal-io` but can be rebound to another stream for special applications.

We describe first two simple functions for asking yes-or-no questions, and then the general function `fquery` on which all querying is built.

`y-or-n-p &optional` *message stream* [*Function*]

This is for asking the user a question whose answer is either "yes" or "no". It types out *message* (if supplied and not `()`), reads a one-character answer, echoes it as "`Yes.`" or "`No.`", and returns `t` if the answer was "yes" or `()` if the answer was "no". The characters which mean "yes" are `Y`, `y`, `T`, `t`, and space. The characters which mean "no" are `N`, `n`, and rubout. If any other character (`?` in particular) is typed, the function will demand a "Y or N" answer.

> Implementation note: Some implementations may choose to allow other characters to be valid answers, such as "hand-up" and "hand-down" in Lisp Machine Lisp.

If the *message* argument is supplied, it will be printed on a fresh line (see `fresh-line` (page 220)). Otherwise it is assumed that a message has already been printed. If you want a question mark and/or a space at the end of the message, you must put it there yourself; `y-or-n-p` will not add it. *stream* defaults to the value of the global variable `query-io` (page 232).

As an example, consider this call:

```
(y-or-n-p "Cannot establish connection.  Retry?  ")
Cannot establish connection.  Retry?
                                     ↑ Input cursor is now here.
```

The second line above was printed by `y-or-n-p`. The third line does not show, but is shown here to indicate where the cursor is when input is expected. If the user type `Y`, then the interaction looks like this:

```
(y-or-n-p "Cannot establish connection.  Retry?  ")
Cannot establish connection.  Retry?  Yes.
```

Now the cursor is just after "`Yes.`".

`y-or-n-p` should only be used for questions which the user knows are coming. If the user is unlikely to anticipate the question, or if the consequences of the answer might be grave and irreparable, then `y-or-n-p` should not be used, because the user might type ahead a `T`, `Y`, `N`, space, or rubout, and thereby accidentally answer the question. For such questions as "Shall I delete all of your files?", it is better to use `yes-or-no-p` (below).

`yes-or-no-p &optional` *message stream* [*Function*]

This, like `y-or-n-p`, is for asking the user a question whose answer is either "Yes" or "No". It types out *message* (if supplied and not `()`), beeps, and reads in a line from the keyboard. If the line is the string "Yes", it returns `t`. If the line is "No", it returns `()`. (Case is ignored, as are leading and trailing spaces and tabs.) If the input line is anything else, `yes-or-no-p` beeps and demands a "yes" or "no" answer. If a `?` is typed at any point, a message will be printed demanding a "yes" or "no" answer.

If the *message* argument is supplied, it will be printed on a fresh line (see `fresh-line` (page 220)). Otherwise the caller is assumed to have printed the message already. If you want a question mark and/or a space at the end of the message, you must put it there yourself; `yes-or-no-p` will not add it. *stream* defaults to the value of the global variable `query-io` (page 232).

To allow the user to answer a yes-or-no question with a single character, use `y-or-n-p`. `yes-or-no-p` should be used for unanticipated or momentous questions; this is why it beeps and why it requires several keystrokes to answer it.

The preceding two functions allow the asking of simple yes-or-no questions. More complicated questions can be asked using `fquery`, described below. `fquery` is quite general and complicated. It is best to write some interface function for each particular kind of question, using `fquery` in the definition. In this way the complicated arguments to `fquery` need be written in only a few places.

`fquery` *options format-string* `&rest` *format-args* [*Function*]

This asks a question, printed by executing

`(format query-io` *format-string format-args*`...)`

and returns the answer. `fquery` takes care of checking for valid answers, reprinting the question when the user clears the screen, giving help, and so forth.

options is a list of alternating keywords and values, used to select among a variety of features. Most callers will have a constant list to pass as *options* (rather than consing up a different list each time).

`:type` The expected form of the answer. The types currently defined are:

- `:inch` A single character, as read by `inch` (page 217). This is the default.
- `:tyi` This is similar to `inch`; the answer is a single character, but the result is an integer, as if read by `tyi` (page 217).
- `:readline` A string, typed as a line terminated by a carriage return, as read by `readline` (page 217).

`:choices` Defines the allowed answers. The allowed forms of choices are complicated and explained below. The default is the same set of choices as for `y-or-n-p` (page 231), if `:type` is `:inch` or `:tyi`, or the same as for `yes-or-no-p`, if `:type` is `:readline`. Note that the `:type` and `:choices` options should be consistent with each other.

> Compatibility note: In Lisp Machine LISP, `:choices` always defaults to `y-or-n-p` choices, even if `:type` is `:readline`. This is clearly bogus.

OK. Extra fiel
in type d

`:list-choices`

If `t`, the allowed choices are listed (in parentheses) after the question. The default is `t`; supplying `()` causes the choices not to be listed unless the user tries to give an answer which is not one of the allowed choices.

`:help-function`

Note - possible implementation-defined HELP key, in addition to "?".

Specifies a function to be called if the user types `"?"`. The default help-function simply lists the available choices. Specifying `()` disables special treatment of `"?"`. Specifying a function of three arguments (the stream, the value of the `:choices` option, and the type-function) allows smarter help processing. The type-function is a function selected by the `:type` option; it does `inch`, `tyi`, or `readline`, but with additional processing. Often it can be ignored by the help-function.

`:fresh-line` If `t`, `query-io` is advanced to a fresh line before asking the question. If `()`, the question is printed wherever the cursor was left by previous typeout. The default is `t`.

`:beep` If `t`, `fquery` beeps to attract the user's attention to the question. The default is `()`, which means not to beep unless the user tries to give an answer which is not one of the allowed choices.

`:clear-input`
If `t`, `fquery` throws away type-ahead before reading the user's response to the question. Use this for unexpected questions. The default is `()`, which means not to throw away typeahead unless the user tries to give an answer which is not one of the allowed choices. In that case, type-ahead is discarded since the user probably wasn't expecting the question.

The argument to the `:choices` option is a list each of whose elements is a *choice*. The cdr of a choice is a list of the user inputs which correspond to that choice. These should be characters if the `:type` is `:inch`, integers corresponding to characters for `:tyi`, or strings for `:readline`. The car of a choice is either an atom which `fquery` should return if the user answers with that choice (in which case nothing is echoed), or a list whose first element is such an atom and whose second element is the string to be echoed when the user selects the choice.

> Compatibility note: In Lisp Machine Lisp the choice-value is specified to be a symbol. To allow `()` to be returned, or even integers, atoms (non-lists) are specified here.

OK.
Probably a typo anyw

In most cases a `:type` of `:readline` would use the first format, since the user's input has already been echoed, and `:inch` or `:tyi` would use the second format, since the input has not been echoed and furthermore is a single character, which would not be mnemonic to see on the display.

As an example, here is a definition of the function `y-or-n-p` in terms of `fquery`:

```
(defun y-or-n-p (&optional message (stream query-io))
  (let ((query-io stream))
    (fquery '(:fresh-line ()
              :list-choices ()
              :choices
                  (((t "Yes.") #\y #\t #\Space)
                   ((() "No.") #\n #\Rubout)))
            (if message "~&~a" "~*")
            message)))
```

As another example, here is a definition of `yes-or-no-p`:

```
(defun y-or-n-p (&optional message (stream query-io))
  (let ((query-io stream))
    (fquery '(:fresh-line ()
              :list-choices ()
              :beep t
              :type :readline
              :choices ((t "Yes") (() "No")))
            (if message "~&~a" "~*")
            message)))
```

As a third example, this function allows more complex choices. One may type `P`, `Q`, `R`, or `D`, in which respective cases the symbol `proceed`, `quit`, `retry`, or `debug` is returned. Space or rubout may be typed instead of `P` or `Q`, respectively.

← paren missing

```
(defun request-i/o-error-action &optional message (stream query
  (let ((query-io stream))
    (fquery '(:fresh-line ()
              :list-choices t
              :choices
                  (((proceed "Proceed") #\p #\Space)
                   ((quit "Quit") #\q #\Rubout)
                   ((retry "Retry") #\r)
                   ((debug "Debug") #\d)))
            (if message "~&~a" "~*")
            message)))
```

↑ truncation

19.6. Streams

19.6.1. Standard Streams

There are several global variables whose values are streams used by many functions in the LISP system. These variables and their uses are listed here. By convention, variables which are expected to hold a stream capable of input have names ending with "`-input`", and similarly "`-output`" for output streams. Those expected to hold a bidirectional stream have names ending with "`-io`".

`standard-input` [*Variable*]

In the normal LISP top-level loop, input is read from `standard-input` (that is, whatever stream is the value of the global variable `standard-input`). Many input functions, including `read` (page 215) and `inch` (page 217), take a stream argument which defaults to `standard-input`.

`standard-output` [*Variable*]

In the normal LISP top-level loop, output is sent to `standard-output` (that is, whatever stream is the value of the global variable `standard-output`). Many output functions, including `print` (page 219) and `ouch` (page 220), take a stream argument which defaults to `standard-output`.

`error-output` [*Variable*]

The value of `error-output` is a stream to which error messages should be sent. Normally this is the same as `standard-output`, but `standard-output` might be bound to a file and `error-output` left going to the terminal or a separate file of error messages.

`query-io` [*Variable*]

The value of `query-io` is a stream which should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. When the normal input to a program may be coming from a file, questions such as "Do you really want to delete all of the files in your directory??" should be sent directly to the user, and the answer should come from the user, not from the data file. `query-io` is used by such functions as `yes-or-no-p` (page 231).

`terminal-io` [*Variable*]

The value of `terminal-io` is ordinarily the stream which connects to the user's console.

`trace-output` [*Variable*]

The value of `trace-output` is the stream on which the `trace` function prints its output.

`standard-input`, `standard-output`, `error-output`, `trace-output`, and `query-io` are initially bound to synonym streams which pass all operations on to the stream which is the value of `terminal-io`. (See `make-synonym-stream` (page 232).) Thus any operations performed on those streams will go to the terminal.

No user program should ever change the value of `terminal-io`. A program which wants (for example) to divert output to a file should do so by binding the value of `standard-output`; that way error messages sent to `error-output` can still get to the user by going through `terminal-io`, which is usually what is desired.

19.6.2. Creating New Streams

Perhaps the most important constructs for creating new streams are those which open files; see `with-open-file` (page 237) and `open` (page 239).

`make-synonym-stream` *symbol* [*Function*]

`make-synonym-stream` creates and returns a "synonym stream". Any operations on the new stream will be performed on the stream which is then the value of the dynamic variable named by the *symbol*. If the value of the variable should change or be bound, then the synonym stream will operate on the new stream.

??? Query: In Lisp Machine LISP this is called `make-syn-stream`. The documentor found it necessary to explain that "syn" meant "synonym"; it certainly isn't obvious. The abbreviation "syn" could be mistaken for any number of other things, such as "synchronous" or "syntactic" or "synthetic" ... Here this confusion is eliminated.

ok. Brain damage accidentally inherited from Multics `make-broadcast-stream &rest` *streams* [*Function*]

Returns a stream which only works in the output direction. Any output sent to this stream will be sent to all of the streams given. The set of operations which may be performed on the new stream is the intersection of those for the given streams. The results returned by a stream operation are the values returned by the last stream in *streams*; the results of performing the operation on all preceding streams are discarded.

|| `make-concatenated-stream &rest` *streams* [*Function*]

Returns a stream which only works in the input direction. Input is taken from the first of the *streams* until it reaches end-of-file; then that stream is discarded, and input is taken from the next of the *streams*, and so on. If no arguments are given, the result is a stream with no content; any input attempt will result in end-of-file.

|| `make-io-stream` *input-stream output-stream* [*Function*]

Returns a bidirectional stream which gets its input from *input-stream* and sends its output to *output-stream*.

|| `make-echo-stream` *input-stream output-stream* [*Function*]

Returns a bidirectional stream which gets its input from *input-stream* and sends its output to *output-stream*. In addition, all input taken from *input-stream* is echoed to *output-stream*.

|| `make-string-input-stream` *string* &optional start end [*Function*]

Returns an input stream which will supply the characters of *string* in order and then signal end-of-file.

|| `make-string-output-stream &optional` *line-length* [*Function*]

Returns an output stream which will accumulate all output given it for the benefit of the function `get-output-stream-string`.

|| `get-output-stream-string` *string-output-stream* [*Function*]

Given a stream produced by `make-string-output-stream`, this returns a string containing all the characters output to the stream so far. The stream is then reset; thus each call to `get-output-stream-string` gets only the characters since the last such call (or the creation of the stream, if no such previous call has been made).

what
happen
to th
structu
interfac
to the
(with

19.6.3. Operations on Streams

`streamp object` [*Function*]

`streamp` returns `t` if its argument is a stream, and otherwise returns `()`.

```
(streamp x) <=> (typep x 'stream)
```

`input-stream-p` [*Function*]

This predicate returns `t` if its argument (a stream) can handle input operations, and otherwise returns `()`.

`output-stream-p` [*Function*]

This predicate returns `t` if its argument (a stream) can handle output operations, and otherwise returns `()`.

`close` *stream* `&optional` *abort-flag* [*Function*]

The stream is closed. No further input/output operations may be performed on it. However, certain inquiry operations may still be performed, and it is permissible to close an already-closed stream.

If *abort-flag* is not `()` (it defaults to `()`), it indicates an abnormal termination of the use of the stream. An attempt is made to clean up any side effects of having created the stream in the first place. For example, if the stream performs output to a file, the file is deleted and any previously existing file is not superseded.

`charpos`, `linenum`, and so on?

19.7. File System Interface

This is somewhat outside
the domain of a language core.

19.7.1. File Names

There are two representations of the name of a file as LISP objects: namestrings and namelists. A *namestring* is a string (or symbol, whose print-name is used) which is the name of the file in an implementation-dependent and file-system-dependent syntax. This representation is intended for use by people. A *namelist* is a list in a special format which is less readable, but more portable and suitable for manipulation by programs.

The model of the file system embodied by namelists is a three-level hierarchy of *host*, *directory*, and *file*. This model is crude, but adapts itself reasonably well to most contemporary operating systems. Generally speaking, a host is thought of as some single computer, a directory as a single lowest-level group of related files within that host, and a file as the smallest named unit of data within the file system. Each of the three levels may be specified by a compound name consisting of a non-empty list of components, each of which may be a string, symbol, or integer: a string is used as a component name; a symbol is used as a special keyword; and an integer is used for its decimal representation, with optional minus sign and no leading zeros.

Symbols which have special meaning are:

*	This indicates an *unspecified* component. or ( ) ?
`:newest`	As a version number, this indicates the most recent version (for input), or one greater than the most recent version (for output creation).
`:oldest`	As a version number, this indicates the least recent version.

The general and canonical form of a namelist is:

((*host-name* . *directory-name*) . *file-name*)

The *host-name* may be unambiguously elided, in which case all its components are taken to be * (unspecified); similarly, the list (*host-name* . *directory-name*) may be unambiguously elided, in which case all components of both *host-name* and *directory-name* are taken to be *.

EXPLAIN THAT IT IS UNAMBIGUOUS BECAUSE THESE GUYS ARE ALWAYS LISTS WHEREAS COMPONENTS ARE NEVER LISTS. IT TOOK ME LITTLE WHILE TO REALIZE TH

As an example, suppose that host CMUC is a TOPS-20 system. Then a file name on that system might be:

```
PS:<SLISP.MANUAL>LISTS.MSS.43
```

and the corresponding namelist might be:

```
(((CMUC) PS SLISP MANUAL) LISTS MSS 43)
```

← I thought you just said ~~these~~ had to be strings.

meaning that PS must always be specified if a directory is?

Similarly, this partially specified file name:

```
<SLISP.MANUAL>LISTS.*
```

might be rendered as a namelist in this way:

```
(((*) * SLISP MANUAL) LISTS * *)
```

Most functions which accept namelists also accept namestrings, and will, in effect, first parse the namestring to produce a namelist. If the namestring is malformed and therefore cannot be parsed, an error will be signaled.

`namelist` *filespec* [*Function*]

The *filespec* argument may be a namelist, a namestring, or a stream which is or was open to a file. The name represented by *filespec* is returned as a namelist in canonical form.

If *filespec* is a stream, the name returned represents the name used to *open* the file, which may not be the *actual* name of the file (see `truename` (page 236)).

`namestring` *filespec* [*Function*]
`file-namestring` *filespec* [*Function*]
`directory-namestring` *filespec* [*Function*]
`host-namestring` *filespec* [*Function*]
`enough-namestring` *filespec defaults* [*Function*]

The *filespec* argument may be a namelist, a namestring, or a stream which is or was open to a file. The name represented by *filespec* is returned as a namelist in canonical form.

??

If *filespec* is a stream, the name returned represents the name used to *open* the file, which may not be the *actual* name of the file (see `truename` (page 236)).

`namestring` returns the full form of the *filespec* as a string. `file-namestring` returns a string representing just the *file-name* portion of the *filespec*; the result of `directory-namestring` represents just the *directory-name* portion; and `host-namestring` returns a string for just the *host-name* portion. Note that the full namestring cannot necessarily be constructed simply by concatenating the three shorter strings in some order.

`enough-namestring` takes another argument, *defaults*, which also should be a namelist, namestring, or file-stream. It returns an abbreviated namestring which is just sufficient to identify the file named by *filespec* when considered relative to the *defaults*. That is,

```
(mergef (enough-namestring filespec defaults) defaults)
  <=> (namelist filespec)
```

`parse-namestring` *string* `&optional` *break-characters start end* [*Function*]

This is the primitive namestring parser. It takes a string argument, and parses a file name within it in the range delimited by *start* and *end* (which are integer indices into *string*, defaulting to the beginning and end of the string). Parsing is terminated upon reaching the end of the specified substring or upon reaching a character in *break-characters*, which may be a string or a list of characters; this defaults to an empty set of characters.

Why not leave this to the normal string-processing functions?

Two values are returned by `parse-namestring`. If the parsing is successful, then the first value is a namelist for the parsed file name, and otherwise the first value is `()`. The second value is an integer, the index into *string* one beyond the last character processed. This will be equal to *end* if processing was terminated by hitting the end of the substring; it will be the index of a break character if such was the reason for termination; it will be the index of an illegal character if that was what caused processing to (unsuccessfully) terminate.

Parsing an empty string always succeeds, producing a namelist with all components unspecified.

`truename` *file-stream* [*Function*]

`truename` returns a namelist for the *actual* name of the file which is or was associated with the stream *file-stream*. This may differ from the name used to open the file because of such file-system features as links, searching for oldest or newest versions, etc.

`mergef` *filespec1 filespec2* [*Function*]

Each argument must be a namelist, namestring, or stream which is or was open to a file. A namelist is constructed and returned whose components are taken from *filespec1*, except that components unspecified ("*") in *filespec1* are taken from *filespec2*. Components not specified by either argument remain unspecified in the result.

`file-name-type` *filespec* [*Function*]
`file-name-version` *filespec* [*Function*]

The argument must be a namelist, namestring, or stream which is or was open to a file. `file-name-type` returns the type part of the *filespec*; `file-name-version` returns the version part of the *filespec*.

`merge-file-name-type` *type filespec* [*Function*]
`merge-file-name-version` *version filespec* [*Function*]

A namelist is returned equivalent to *filespec* except that the first argument specifies the type or version part; a first argument of * forces the appropriate part to be unspecified. The first argument must be a symbol, string, or integer; the *filespec* argument must be a namelist, namestring, or stream which is or was open to a file.

19.7.2. Opening and Closing Files

When a file is *opened*, a stream object is constructed which is the file system's ambassador to the LISP environment; operations on the stream are reflected by operations on the file in the file system. The act of *closing* the file (actually, the stream) ends the association; the transaction with the file system is terminated, and input/output may no longer be performed on the stream. The stream function `close` (page 237) may be used to close a file; the functions described below may be used to open them.

`with-open-file` *bindspec* `&rest` *body* [*Special form*]

(`with-open-file` (*stream filename options*) . *body*) evaluates the forms of *body* (an implicit `progn`) with the variable *stream* bound to a stream which reads or writes the file named by the value of *filename*. The *options* should evaluate to a keyword or list of keywords.

— SAY EXA
THAT IT OP
THE FILE

When control leaves the body, either normally or abnormally (such as by use of `throw` (page 64)), the file is closed. If a new output file is being written, and control leaves abnormally, the file is aborted and it is, so far as possible, as if it had never been opened. Because `with-open-file` always closes the file, even when an error exit is taken, it is preferred over `open` for most applications.

filename is the name of the file to be opened; it can be a namelist or namestring. If an error occurs (such as "File Not Found"), the user is asked to supply an alternate pathname, unless this is overridden by *options*. At that point the user can quit or enter the error handler if the error was not due to a misspelled pathname after all.

options is either a single keyword or a (possibly empty) list of options, where an option is either a keyword or a list of a keyword and arguments to that keyword. (If a keyword with an argument is to be used, then *options* must be a list of options and not a single option.)

Compatibility note: Lisp Machine LISP uses a format where the argument to a keyword simply follows the keyword in the list. This is not compatible with other keyword formats, for example that of `defstruct`. It only makes a difference here in the case of `:byte-size`. It seems worthwhile to minimize the number of keyword formats in COMMON LISP.

There are many keyword-taking functions that work this way in the Lisp Machine. The reason for this is so that you can simply pass the keywords as arguments, e.g. (OPEN name ':READ ':BYTESIZE 65)

Unfortunately this is damaged in the case of OPEN, since it doesn't take an &REST argument. So I don't care what OPEN does, but don't rule this out in general.

Valid keywords are:

`:in` or `:input` or `:read`
Open file for input. This is the default.

`:out` or `:output` or `:write` or `:print`
Open file for output; a new file is to be created.

`:append` Open an existing file for output, arranging that output to the resulting stream should be appended to the previous contents of the file.

> **Compatibility note:** Not all file systems can support this operation. An implementation may choose to simulate it by copying the old file into a new one and then continuing to write the new one.

> **Compatibility note:** The Lisp Machine LISP implementation appears not to support this, but MACLISP does in the `open` function.

`:read-alter` Open a file in read-alter mode; the result is a stream which can perform both input and output on a random-access file.

> **Compatibility note:** Not all file systems can support this operation.

`:character` or `:ascii`
The unit of transaction is a character; the file is a text file. This is the default.

`:binary` or `:fixnum`
The unit of transaction is a small unsigned integer. The `:byte-size` option may be used to specify the number of binary bits in the transaction unit. This precise way in which this interacts with the file system is implementation-dependent.

`:byte-size` This keyword takes an argument, an integer specifying the number of bits per transaction unit; this is used in conjunction with the `:binary` option. If the `:binary` keyword is specified but the `:byte-size` keyword is not, then an implementation-dependent "natural" byte size is used.

`:echo` This keyword requires an argument, an output stream, and is valid only when opening a stream for input. The result stream will echo everything read from it onto the output stream.

Is it necessary to put echoing in in so many different places? It's an operating system functio anyw

`:probe` This keyword specifies that the file is not being opened to do I/O, but only to find out information about it. A stream is returned, but it cannot handle I/O transactions; it is as if the stream were immediately closed after opening it. `:probe` implies `:noerror` (see below).

> ??? Query: In Lisp Machine LISP, `:probe` also implies `:fixnum`. Why??

It affects what :info message returns. But it's accidental; woul be okay cha

`:noerror` If the file cannot be opened, then instead of returning a stream, a string containing the error message is returned. If `:noerror` is not specified, then an error is signalled using the error message, and the user is asked to supply a different filename.

`open` *filename* `&optional` *options* [*Function*]

Returns a stream which is connected to the file specified by *filename*. The *options* argument is as for `with-open-file` (page 240). If an error occurs, such as "File Not Found", the user is asked to supply an alternate pathname, unless the `:noerror` (page 241) option is used, in which case the error message is returned as a string.

When the caller is finished with the stream, it should close the file by using the `close` (page 237) function. The `with-open-file` (page 240) special form does this automatically, and so is preferred for most purposes. `open` should be used only when the control structure of the program necessitates opening and closing of a file in some way more complex than provided by `with-open-file`. It is suggested that any program which uses `open` directly should use the special form `unwind-protect` (page 63) to close the file if an abnormal exit occurs.

Implementation note: While `with-open-file` tries to automatically close the stream on exit from the construct, for robustness it is helpful if the garbage collector can detect discarded streams and automatically close them.

That implies a lot of assumptions about the implementation.

19.7.3. Renaming, Deleting, and Other Operations

`renamef` *file* *new-name* `&optional` *error-p* [*Function*]

file can be a filename or a stream which is open to a file. The specified file is renamed to *new-name* (a filename). If *error-p* is not `()` (the default is `t`), then if a file-system error occurs it will be signalled as a LISP error. If *error-p* is `()` and an error occurs, the error message will be returned as a string. If no error occurs, `renamef` returns `t`.

`deletef` *file* `&optional` *error-p* [*Function*]

file can be a filename or a stream which is open to a file. The specified file is deleted. If *error-p* is not `()` (the default is `t`), then if a file-system error occurs it will be signalled as a LISP error. If *error-p* is `()` and an error occurs, the error message will be returned as a string. If no error occurs, `deletef` returns `t`.

`probef` *filename* [*Function*] ?

Returns `()` if there is no file named *filename*, and otherwise returns a filename which is the true name of the file (which may be different from *filename* because of file links, version numbers, or other artifacts of the file system).

// `faslp` *file* `&optional` *implementation* [*Function*]

file can be a filename or a stream which is open to a file. This predicate returns `t` if the file is a fasload (compiled) file in *implementation* format, and otherwise returns `()`. The argument *implementation* defaults to the implementation in which the call is executed.

Think of a better name for this.

`file-creation-date` *file* [*Function*]

file can be a filename or a stream which is open to a file. This returns the creation date of the file as an integer in universal time format, or `()` if this cannot be determined.

`file-author` *file* [*Function*]

file can be a filename or a stream which is open to a file. This returns the name of the author of the file as a string, or `()` if this cannot be determined.

What about the other fifty attributes of a file you might want to ask for?

`filepos` *file-stream* `&optional` *position* [*Function*]

`filepos` returns or sets the current position within a random-access file.

(`filepos` *file-stream*) returns a non-negative integer indicating the current position within the *file-stream*, or `()` if this cannot be determined. Normally, the position is zero when the stream is first created. For a `:character` (page 241) stream, the position is in units of characters; for a `:binary` (page 241) file, the position is in bytes.

(`filepos` *file-stream position*) sets the position within *file-stream* to be *position*. The *position* may be an integer, or `()` for the beginning of the stream, or `t` for the end of the stream. If the integer is too large, an error occurs (the `lengthf` (page 240) function returns the length beyond which `filepos` may not access). With two arguments, `filepos` returns `t` if it actually performed the operation, or `()` if it could not.

WHY NOT FILE-LENGTH? YOU DIDN'T USE AUTHORF...

`lengthf` *file-stream* [*Function*]

file-stream must be a stream which is open to a file. The length of the file is returned as a non-negative integer, or `()` if the length cannot be determined. For a `:character` (page 241) stream, the position is in units of characters; for a `:binary` (page 241) file, the position is in bytes.

stream

19.7.4. Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files; the expressions in the file are mostly special forms such as `defun` (page DEFUN-FUN), `defmacro` (page DEFMACRO-FUN), and `defvar` (page 21) which define the functions and variables of the program.

Loading a compiled ("fasload") file is similar, except that the file does not contain text, but rather predigested expressions created by the compiler which can be loaded more quickly.

`load` *filename* `&optional` *package nonexistent-ok do-not-set-default* [*Function*]

This function loads the file named by *filename* into the Lisp environment. If the file is a fasload file, it calls `fasload`; otherwise it calls `readfile`. The argument *package* can be used to specify the package into which to load the file; it can be either a package or the name of a package as a string or a symbol. If *package* is not specified, `load` prints a message saying what package the file is being loaded into. If *nonexistent-ok* is specified and not `()`, `load` just returns `()` if the file

It also chooses the right one

cannot be opened. If the file is successfully loaded, `load` always returns `t`.

IF FILENAME IS SAME, DOESN'T MEAN SAME FILE! (versions)

`load` maintains a default filename, used to default missing components of the *filename* argument; thus `(load)` will load the same file previously loaded. Normally `load` updates the filename defaults from *filename*, but if *dont-set-default* is specified and not `()` this is suppressed.

inconsistent with argument specification on previous page.

If *filename*, after defaulting, is still missing components in such a way that it does not specify either a fasload or LISP source file, then `load` first tries to open a fasload file, and failing that tries to load a LISP source file, in each case trying to load the most recent version.

`readfile &optional` *filename package no-msg-p* [*Function*]

`readfile` is the version of `load` (page 243) for text files. It reads and evaluates each expression in the file, discarding the results. As with `load`, *package* can specify what package to read the file into. Unless *no-msg-p* is specified and not `()`, a message is printed indicating what file is being read into what package. The defaulting of *filename* is the same as with `load`.

`fasload` *filename* `&optional` *package no-msg-p* [*Function*]

`fasload` is the version of `load` for fasload (compiled) files. It defines functions and performs other actions as directed by the specifications inserted in the file by the compiler; the result is roughly the same as performing `readfile` on the original source file, but faster, and with functions being in compiled form. As with `load`, *package* can specify what package to read the file into. Unless *no-msg-p* is specified and not `()`, a message is printed indicating what file is being loaded into what package. The defaulting of *filename* is the same as with `load`.

??? **Query:** There are several problems with the above specifications, which are essentially as in Lisp Machine LISP.

- The arguments for the three functions are not compatible; there is not even a subset relationship.

  Make them take keyword arguments.

- The file name defaulting has been criticized as being "very M.I.T."; not all cultures prefer to maintain default file names in this way.

  Make it a mode.

- There ought to be an option to print the results of each evaluation (and in the case of `fasload`, the name of each function as it is loaded). Another flavor of this option is to print a one-character blip every so often to signal progress. This is useful for debugging, for example to determine where in a load file an error is occurring.

These problems should be fixed.

Make the first flavor a keyword.
The second flavor sounds monumentally useless.

19.7.5. Accessing Directories

(missing)

It wouldn't bother me to change the defaults about messages and so forth when they aren't overridden with keywords. There should be a special variable with a default set of keywords in it.

For some reason you forgot to make files be sequences.

A lot of the stuff in this
chapter reproduces Lisp
machine stuff which we
consider to be brain-
damaged and are redesigni
I would say to leave out
things marked with an X
for now. We do understa
the issues better now.

Chapter 20

Errors

X COMMON LISP handles errors through a system of *conditions*. One may establish *handlers* which gain control which conditions occur, and *signal* a condition when an error actually occurs. When the system or a user function detects an error it signals an appropriately named condition and some handler established for that condition will deal with it.

X The condition mechanism is completely general and could be used for purposes other than "error" handling. There are some functions supplied in COMMON LISP which make use of the condition mechanism to handle errors in a convenient way.

20.1. Signalling Conditions

X Condition handlers are associated with conditions (see next section). Every condition is essentially a name, which is a symbol or `()`. When an unusual situation occurs, such as an error, a condition is signalled. The system (essentially) searches up the stack of nested function invocations looking for a handler establised to handle that condition. The handler is a function which gets called to deal with the condition.

`signal` *condition-name* `&rest` *args* [*Function*]

X This searches through all currently established condition handlers, perhaps twice, starting with the most recent. If it finds one which was established to handle `()` or *condition-name*, then it calls that handler with a first argument of *condition-name* and with *args* as the rest of the arguments. If the first value returned by the handler is `()`, `signal` will continue searching for another handler; otherwise it will return the first two values returned by the handler. If *condition-name* is not `T`, and if no handler was willing to handle the condition, then a second pass of the established condition handlers is made, searching for any handler established to handle `T`. If one is found that is used in the same manner as in the first pass search. If there is still no willing handler found then `signal` returns `()`.

X Thus a handler set up to handle condition `()` will handle *all* conditions which are not handled by a more recently established handler. This is intended to make it easy to set up a debugger which intercepts all errors and handles them itself. Note that such a handler doesn't have to actually handle all conditions; it will be offered the chance to do so but can return `()` to refuse any condition which it doesn't wish to handle.

Conditions established to handle condition `t` will handle *any* condition for which a more specific willing handler can't be found. This makes it easy to set up, at any time, a handler which which will be given a chance to handle all conditions that no one else wants.

20.2. Establishing Handlers

Condition handlers are established through the `condition-bind` or `condition-setq` special forms. These have behaviors somewhat analagous to `let` and `setq`. They make use of the ordinary variable binding mechanism, so that if a `condition-bind` is thrown through the handlers get disestablished. It also means that in multiple stack group implementations of COMMON LISP the handlers are established only in the current stack group.

`condition-bind` *bindings* `&rest` *body* [*Special form*]

This is used to establish handlers for conditions then perform the *body* in that established handler environment.

For example:

For example:

```
(condition-bind ((cond1 hand1)
                 (cond2 hand2)
                 ...
                 (condm handm))
  form1
  form2
  ...
  formn)
```

Each *condj* is either the name of a condition or a list of names of conditions. Each *handj* is a form which is evaluated to produce a handler function. No guarantee is made on the order of evaluation of these forms, but the conditions are established in sequential order, so that *cond1* will be looked at first. The expressions *formj* are then evaluated in order; the values of all but the last are discarded (that is, the body of the `condition-bind` form is an implicit `progn`). The value of the `condition-bind` form is the value of *formn* (if the body is empty, `()` is the value). The established conditions become disestablished when the `condition-bind` form is exited.

As an example consider:

For example:

```
(condition-bind ((:wrong-type-argument 'some-wta-handler)
                 ((silliness-1 silliness-2) silliness-handler))
  (format msgfiles '"Yodle-eh-eh-ho!")
  (+ 23 ()))
```

This sets up the function `some-wta-handler` to handle the `:wrong-type-argument` condition. The value of the symbol `silliness-handler` is set up to handle both the `silliness-1` and `silliness-2` conditions. With these handlers set up, it outputs a message and then causes a `:worng-type-argument` error by trying to add 23 to `()`, which is not a number. The condition handler `some-wta-handler` will be given a chance to handle the error.

`condition-setq &rest` *specs* [*Special form*]

The `condition-setq` form is used to establish condition handlers as a side effect of some operation -- for instance loading a file which contains condition handlers and a `condition-setq` form to establish them.

It takes the form:

For example:

```
(condition-setq cond1  hand1
                cond2  hand2
                ...
                condn  handn)
```

Each *condj* is either the name of a condition or a list of names of conditions. Each *handj* is a form which is evaluated to produce a handler function. No guarantee is made on the order of evaluation of these forms, but the conditions are established in sequential order, so that *cond1* will be looked at first.

The conditions established by `condition-setq` remain established until execution is unwound (either normally or by being thrown) past the most recent `condition-bind`. Multiple uses of `condition-setq` cause the most recently established handler to be tried first when a condition is signalled. For example, consider:

For example:

```
(condition-setq :wrong-type-argument 'default-wta-handler)
(+ 23 ())
(condition-setq :worng-type-argument 'hairy-wta-handler)
(+ 105 ())
```

When the first `:wrong-type-argument` error is signalled (because of the attempt to add 23 to `()`) the function `default-wta-handler` will be given first chance at handling the error. When the second error is signalled (because of the attempt to add 105 to `()`) the function `hairy-wta-handler` will be given first chance. If it declines (by returning `()` as its first result) then `default-wta-handler` will be given a chance.

20.3. Error Handlers

When `signal` (page 245) invokes a condition handler it passes it the *condition-name* along with whatever other arguments were handed to `signal`. Condition handlers set up to handle errors can safely assume certain things about those arguments for all errors signalled by the system or signalled by user code via the functions `ferror` (page 247) and `cerror` (page 247).

An error handler can expect to be invoked as

For example:

`(funcall*` *error-handler*
condition-name
control-string
proceedable-flag
restartable-flag
function
params`)`

where *params* may vary in length. Handlers for particular condition names may expect certain parameters to always be included in the *params* list. The parameters supplied by the system for certain standard conditions are given in ???section-ref "standard condition names"???. The program siganalling the condition is free to pass any extra parameters. All error handlers should therefore be written with `&rest` arguments.

The *condition-name* is the name of the condition signalled.

control-string should be a string which when given to `format` (page 221) as a control string, along with *params* as additional arguments, produces some reasonable explanation of the error. It is up to the handling function whether it makes use of that control string.

The third and fourth arguments are flags. If the *proceedable-flag* is non-`()` then the error is said to be *proceedable.* If the *restartable-flag* is non-`()` then the error is said to be *restartable.* The values of these flags may be used by the signallers and handlers to pass more information than a single bit. It is up to the user how these are used. For instance, a set of signallers and handlers may pass information concerning the values expected from the handler when an error is proceeded.

function is the name of the function which initiated the signalling of the error, or `()` if the signaller can't determine it.

An error handler can do some processing and then make one of four respones to the error (assuming the error was signalled with `ferror` (page 247) or `cerror` (page 247)). It can return `()` to decline handling the error, it can *proceed,* it can *restart* or it can *throw.*

Throwing simply consists of using the function `throw` (page 64) to some tag outside the scope of the error.

Proceeding and *Restarting* are achieved by returning from the error handler with multiple values. The first value should be one of the following:

`:return`	This means to *proceed* the error. If the error was signalled by `cerror` and the error was proceedable then the second value returned with `:return` is returned as the value of `cerror`. If the error was not proceedable (always the case for errors signalled by `ferror`, then the system forces a `break` (page 249).
`:error-restart`	This means to *restart* the error. If the error was signalled by `cerror` and the error was restartable then the second value returned with `:error-restart` is thrown to the catch

tag `error-restart`. If the error was not restartable (always the case for errors signalled by `ferror`, then the system forces a `break` (page 249). An error may also be simply restarted from the handler by throwing directly from there to a catch tag of `error-restart`, but that is not as bullet proof if the error wasn't in fact restartable.

No other values are legal as the first values returned by error handlers. For errors signalled by `ferror` or `cerror` illegal values will force a `break`.

20.4. Signalling Errors

LISP programs can signal errors by using one of the functions `ferror` (for *f*atal *error*) or `cerror` (for *c*ontinuable *error*).

we maybe leave these out, too,
we are planning on flushing
them out in favor of a new
one with different arguments.

`ferror` *condition-name control-string* `&rest` *params* [*Function*]

`ferror` signals the error condition *condtition-name*. The associated error message is obtainable by calling `format` (page 221) on *control-string* and *params*. The error is neither proceedable nor restartable. Function `ferror` never returns. It can be thrown through however. A usual COMMON LISP environment will have some sort of error handler established for condition name `t`. Thus the user can get at least minimal error handling with `ferror` using a null *condition-name* knowing that the error will at least be signalled to the user console.

`cerror` *proceedable-flag restartable-flag condition-name control-string* `&rest` *params* [*Function*]

`cerror` is similar to `ferror` (see above) except for *proceedable-flag* and *restartable-flag*. These are passed through to the eventual error handler. If `cerror` is called with a non-`()` *proceedable-flag* the caller should be prepared to accept the returned value of `cerror` and use it to retry the operation that failed. If `cerror` is passed a non-`()` *restartable-flag* then there should be a catch for taf `error-restart` somewhere above the caller.

`error-restart` `&rest` *body* [*Macro*]

Flush this crock.

`error-restart` can be used to denote a seciotn of a program that can be restarted if certain errors occure during its execution. The form of an `error-restart` is:

For example:

```
(error-restart
  form1
  form2
  ...
  formn)
```

The expressions *formj* are evaluated in order; the values of all but the last are discarded (that is, the body of the `error-restart` form is an implicit `progn`). The value of the `error-restart` form is the value of *formn* (if the body is empty, `()` is the value). If a *restartbale* error occurs during the evaluation of one of the *formj*'s, and the handler responds by forcing a a restart, then the forms, beginning with *form1* will be re-evaluated in order. The only way a restartable error can occur is if `cerror` is called with a *restartable-flag* which is non-`()`.

`error-restart` is implemented as a macro which expands into:

For example:

```
    (prog ()
      loop (*catch 'error-restart
             (return (progn form1
                            form2
                            ...
                            formn)))
           (go loop))
```

← formatting lossage occurred here

`check-arg` *var-name predicate description*
type-name [*Macro*]

The `check-arg` macro is used to check arguments to make sure they are valid, signal a `:wrong-type-argument` condition if they are not, and use the value returned by the handler to replace the invalid value.

var-name is the name of the variable whose value is being checked to be of the correct type. If the error is proceeded this variable will be `setq`'ed to a replacement value. *predicate* is a test for whether the variable is of the correct type. It can either be a symbol whose function definition takes one argument and returns non-`()` if the type is correct, or it can be a non-atomic form which is evaluated to check the type - usually such a form would contain a reference to the variable *var-name*. *description* is a string which expresses *predicate* in English. It is used in error messages. *type-name* gets passed to the `:wrong-type-argument` handler as the first required parameter of that class of error handlers (see section ??? <<<section ref "standard condition names">>>).

Thus `check-arg` has what consistitutes a valid argument specified to it in three ways. *predicate* is executable, *description* is human understandable and *type-name* is program understandable.

`check-arg` uses *predicate* to determine whether the value of the variable is of the correct type. If it is not a `:wrong-type-argument` condition is signalled with four parameters - *type-name*, the bad value, the symbol *var-name* and *description*. If the error is proceeded, the variable is set to the value returned and `check-arg` repeats the process. Only the first of these two parameters are defined for `:wrong-type-argument` handlers, and so theyt should not depend on the meanin of more than these two.

Consider for example:

For example:

```
    (check-arg foo
               (and (fixnump foo) (< foo 0.))
               "a negative fixnum"
               negative-fixnum)
```

If `foo` is not of the right type an error will be signalled and a `:wrong-type-argument` which makes use of the *control-string* and *parameters* passed to it will print the message (at least):

```
Argument foo was 33, which is not a negative fixnum.
```

20.5. Break-points

Often error handlers want to pass control to the user's terminal. The user can then examine variable bindings and respond to the error, or perhaps just start some new computation. Control is passed by using the special form `break`.

`break` *tag* `&optional` *conditional-form* [*Special form*]

This enters a *breakpoint* loop, which is similar to a LISP top level loop. (`break` *tag*) will always enter the loop; (`break` *tag conditional-form*) will evaluate *conditional-form* and only enter the break loop if it returns non-`( )`. If the break loop is entered, `break` prints out

For example:

`;bkpt` *tag*

and then enters a loop reading, evaluating, and printing forms. After reading a form `break` checks for the following special cases. If the symbol `<altmode>p` is typed, `break` return `( )`. If the the symbol `<altmode>r` is typed, `break` throws to a catch tag `error-restart`. If the symbol `<altmode>g` is typed, `break` throws back to the LISP top level. If (`return` *form*) is typed, `break` evaluates *form* and returns the result. In other respects a break loop appears very simialr to a top level loop.

Bart

20.6. Standard Condition Names

Some condition names are used by the COMMON LISP system itself. They are listed below along with the arguments they expect and the conventions followed in use of these conditions. All error condition handlers expect at least four arugments: *condition-name*, *control-string*, *proceedbale-flag*, and *restartable-flag*. In addition some condition names expect particular values for the fifth and subsequent arguments. These are included in the list below. It is always permissible to supply even more arguments than those required.

*** this list is not yet complete ***

`:wrong-type-argument`

Requires *type-name* and *value*, where the first is a symbol indicating what type of value is required, and the second is the bad value supplied to the fnction signalling the error. If the error is proceeded, the value returned by the handler (that is, the second value returned; the first would be `:return`) should be a new value for the argument to be used instead of the one which was of the wrong type.

`:inconsistent-arguments`

Requires *list-of-inconsistent-argument-values*. This condition is signalled when the arguments to a function are inconsistent with each other, but the fault does not lie with any particular one of them. If the error is proceeded, the value returned by the handler should be returned by the function whose arguments were inconsistent.

Where did errset go?
(and catch_error)

Chapter 21

The Compiler

The compiler is a program THAT THAT ~~which~~ makes code run faster, by translating programs into an implementation-dependent form (subrs) ~~which~~ can be executed more efficiently by the computer. Most of the time you can write programs without worrying about the compiler; compiling a file of code should produce an equivalent but more efficient program. When doing more esoteric things, one may need to think carefully about what happens at "compile time" and what happens at "load time". Then the difference between the syntaxes "`#.`" and "`#,`" becomes important, and the `eval-when` (page EVAL-WHEN-FUN) construct becomes particularly useful.

Most declarations (see ???) are not used by the COMMON LISP interpreter; they may be used to give advice to the compiler. The compiler may attempt to check your advice and warn you if it is inconsistent.

Unlike most other LISP dialects, COMMON LISP recognizes `special` declarations in interpreted code as well as compiled code. This potential source of incompatibility between interpreted and compiled code is thereby *eliminated* in COMMON LISP.

I assume the real manual won't wait until chapter 21 to hint at lexical scoping.

The internal workings of a compiler will of course be highly implementation-dependent. The following functions provide a standard interface to the compiler, however.

`compile` *name* `&optional` *definition* [*Function*]

If *definition* is supplied, it should be a lambda-expression, the interpreted function to be compiled. If it is not supplied, then *name* should be a symbol with an interpreted-code definition; that definition is compiled.

The definition is compiled and a subr object produced. If *name* is a symbol, then the subr object is installed as the global function definition of the symbol and the symbol is returned. If *name* is `()`,then the subr object itself is returned.

For example:

```
(defun foo ...) => foo                          ;A function definition.
(compile 'foo) => foo                           ;Compile it.
                ;Now foo runs faster.
(compile () '(lambda (a b c) (- (* b b) (* 4 a c))))
  => a compiled function of three arguments which computes b^2-4
```

`comfile` *input-filespec* `&optional` *output-filespec* [*Function*]

ok

Each argument should be a valid file name specifier for `with-open-file` (page 240). The file should be a LISP source file; its contents are compiled and written as a `fasload` (page 244) file to *output-filespec*. The *output-filespec* defaults in a manner appropriate to the implementation's file system conventions.

`cl` *name-or-definition* [*Function*]

The argument should be a symbol with an interpreted-code function definition or a lambda-expression. The definition is compiled and the resulting code printed in a symbolic format. This is primarily useful for debugging the compiler, but also often of use to the novice who wishes to understand the workings of compiled code.

> **Implementation note:** Implementors are encouraged to make the output readable, preferably with helpful comments.

Flush this.
Compiler
maintai
can defin
in new
file.
Also diss
could com
first if g
a lam

`disassemble` *name-or-subr* [*Function*]

The argument should be a symbol with a compiled-code function definition or a subr object. The compiled code is "reverse-assembled" and printed out in a symbolic format. This is primarily useful for debugging the compiler, but also often of use to the novice who wishes to understand the workings of compiled code.

> **Implementation note:** Implementors are encouraged to make the output readable, preferably with helpful comments.

more to come

Chapter 22

STORAG

Chapter 23

LOWLEV

Index

Index of Concepts

- " macro character 203

- # macro characters 204

- ' macro character 202

- (macro character 202

-) macro character 202

- , macro character 204

- ; macro character 203

- **Implementation note** 9, 11, 22, 30, 46, 73, 86, 91, 95, 97, 150, 153, 167, 224, 231, 242, 254
- **Incompatibility** 9, 17, 26, 27, 29, 31, 36, 37, 39, 40, 45, 48, 51, 53, 55, 61, 62, 64, 76, 78, 79, 82, 83, 84, 85, 87, 89, 108, 109, 112, 125, 126, 131, 146, 147, 151, 156, 157, 176, 177, 178, 179, 180, 185, 186, 189, 192, 193, 194, 200, 206, 209, 210, 212, 214, 215, 222, 230, 232, 233, 240, 241
- **Query** 11, 14, 16, 21, 42, 59, 60, 64, 65, 72, 86, 115, 129, 156, 169, 176, 179, 189, 192, 195, 209, 217, 218, 223, 224, 229, 235, 241, 244
- **Rationale** 24, 71, 81, 108, 124, 183, 188, 215, 224

- A-list 141
- Access functions 184
- Alterant macros 187
- Array 19
  - predicate 29
- Association list 49, 141
  - as a substitution table 132
  - compared to hash table 146
- Atom
  - predicate 27

- Bignum 9
- Bit
  - predicate 29
- Bit vector
  - infinite 91
  - integer represention 91
- Bit-vector
  - predicate 29
- Byte 94
- Byte specifiers 94

- Car 16, 123
- Catch 61
- Cdr 16, 123
- Character
  - predicate 29
- Character syntax 204
- Cleanup handler 63
- Closure 30
- Comments 203
- Conditional
  - `and` 32
  - `or` 33
  - during `read` 210
- Cons 16, 123
  - predicate 27
- Constructor macro 184
- Constructor macros 186
- Control structure 35

- Data type
  - predicates 26
- Declarations 71
- Defstruct 183
- Denominator 12
- Displaced array 176
- Dotted list 123

- Empty list
  - predicate 26
- Environment structure 35

- Fill pointer 179
- Fixnum 9
- Floating-point number 10
  - predicate 28
- Flow of control 35
- Formatted output 221

- General vector 17

- Hash table 146, 150
- Host 237

- Implicit `progn` 35, 41, 42, 43, 44, 45, 48, 59
- Index offset 176
- Indicator 75
- Indirect array 176
- Integer 8
  - predicate 27
- Iteration 47

- Keywords
  - for `defstruct` slot-descriptions 187
  - for `condition` 251
  - for `declare` 72

  for `defstruct` 188
  for `error` 248
  for `fquery` 232
  for `make-array` 175
  for `namelist` 238
  for `Parm(Text)` 189, 232
  for `with-open-file` 241

List 16, 123
  predicate 27
    See also dotted list
List syntax 202
Logical operators
  on `t` and `()` 32

Macro character 201
Mapping 52
Merging
  file names 239
  sorted sequences 120
Multiple values 58
  returned by `read-from-string` 219

Namelist 237
Namestring 237
Naming conventions 133
  predicates 23
Non-local exit 61
Number 81
  floating-point 10
  predicate 27
Numerator 12

Package cell 75
Parsing 201
Plist 75
Position
  of a byte 94
Predicate 23
Print name 75, 78, 151
Print-name
  coercion to string 155
Printed representation 199
Printer 199
Property 75
Property list 75
  compared to association list 75
  compared to hash table 146
Pseudo-predicate 23, 99

Querying the user 230
Quote character 202

Rank 19
Reader 199, 200
Readtable 211
Record structure 183

Set
  list representation 132

Sets
  bit-vector representation 91
  infinite 91
  integer representation 91
Shared array 176
Sharp-sign macro characters 204
Size
  of a byte 94
Sorting 118
String 151
  predicate 29
String syntax 203
Structure 183
Substitution 131
Symbol 7, 75
  coercion to a string 151
  coercion to string 155
  predicate 26
Symbol syntax 203

Throw 61
Tree 17

Unwind protection 63
Unwinding a stack 64

Vector
  infinite 91
  integer represention 91

Yes-or-no functions 230

` macro character 204

| macro character 203

Index of Variables

A

B
base *219*

C
char-bits-limit 97, *101*
char-code-limit 97, *101*
char-control-bit 104
char-font-limit *13*, *14*, 97, *101*
char-hyper-bit 104
char-meta-bit 104
char-super-bit 104

D
double-pi 87

E
error-output 235

F
features *210*

G

H

I

J

K

L
long-pi 87

M

N

O

P
pi 87
prinlength *209*, *219*
prinlevel *192*, *210*, *219*

Q
query-io *231*, *232*, 235

R
read-default-float-format *11*
read-preserve-delimiters 215, *219*
readtable 211, *211*

S
sample-variable 5
short-pi 87
si:*gensym-counter *80*
si:*gensym-prefix *80*
single-pi 87
standard-input *214*, 234
standard-output *219*, *221*, 234

T
terminal-io *214*, *219*, 235
trace-output 235

U

V

W

X

Y

Z

Chuckle

Index of Keywords

A
:alterant
 for defstruct *186*, 190
:append
 for with-open-file 241
:array
 for type option to defstruct 189
:array-leader
 for type option to defstruct 189
:ascii
 for with-open-file 241

B
:beep
 for fquery 233
:binary
 for with-open-file 241, *243*
:byte-size
 for with-open-file 241

C
:callable-accessors
 for defstruct 193
:character
 for with-open-file 241, *243*
:choices
 for fquery 232
:clear-input
 for fquery 233
:conc-name
 for defstruct *186*, 188, *191*
:constructor
 for defstruct *186*, 190, *193*

D
:displaced-index-offset
 for make-array 176
:displaced-to
 for make-array 175

E
:echo
 for with-open-file 241
:error-restart
 for error 248
:eval-when
 for defstruct 193

F
:fixnum
 for type option to defstruct 189
 for with-open-file 241
:fresh-line
 for fquery 233
:ftype
 for declare 72

G

H
:help-function
 for fquery 232

I
:in
 for with-open-file 241
:inch
 for type option to fquery 232
:include
 for defstruct 190
:inconsistent-arguments
 for condition 251
:initial
 for make-array 175
:initial-offset
 for defstruct 193
:inline
 for declare 73
:input
 for with-open-file 241
:integer
 for type option to defstruct 189
:invisible
 for defstruct slot-descriptions 188

J

K

L
:leader-length
 for make-array 175
:leader-list
 for make-array 175
:list
 for type option to defstruct 189
:list-choices
 for fquery 232

M
:make-array
 for defstruct *189*, 191

N
:named
 for defstruct 189, *190*
:newest
 for namelist 238
:noerror
 for with-open-file 241, *242*
:notinline

  for declare 73

O

:oldest
  for namelist 238
:out
  for with-open-file 241
:output
  for with-open-file 241

P

:predicate
  for defstruct 190
:print
  for with-open-file 241
:print-function
  for defstruct 192
:printer
  for defstruct *19*
:probe
  for with-open-file 241

Q

R

:read
  for with-open-file 241
:read-alter
  for with-open-file 241
:read-only
  for defstruct slot-descriptions 188
:readline
  for type option to fquery 232
:return
  for error 248

S

:size-macro
  for defstruct 192
:size-variable
  for defstruct 192
:special
  for declare *41*, 72

T

:tyi
  for type option to fquery 232
:type
  for defstruct slot-descriptions 187
  for declare 72
  for defstruct 188, *191*
  for fquery 232
  for make-array 175

U

:unnamed
  for defstruct 189

V

:vector
  for type option to defstruct 189

W

:write
  for with-open-file 241
:wrong-type-argument
  for condition 251

X

Y

Z

Index of Functions, Macros, and Special Forms

* 84
*catch *60*, 62
*throw *60*, 64
throw *63*
*unwind-stack 64
+ 84
- 84
/ 85
1+ 85
1- 85
< 83, *101*
<= 83
= *30*, *32*, *81*, 82
> 83
>= 83

A
abs 84
acons *76*, 142
add1 85
adj 135
adjoin 135
adjq 135
adjust-array-size *179*, 180
alphanumericp 99
alphap 98
and 32, *44*, *61*
append 126, *128*, *138*
apply 39, *60*
aref *19*, *151*, 177
array-active-length *177*, 177
array-dimension 178
array-dimensions 178
array-grow 180
array-has-leader-p 178
array-in-bounds-p 178
array-leader 178
array-leader-length *178*, 178
array-length *109*, 177, *179*
array-pop 180
array-push 179
array-push-extend 179
array-rank 177
array-reset-fill-pointer 179
array-type 177
arrayp 29
aset *108*, 177
ash 93
ass 143
ass-if 143
ass-if-not 143
assoc 143, *144*, *145*
assq 143
atan 87
atand 87
atom 27

B
bigp *27*, *82*
bit 166
bit-and 169
bit-andc1 169
bit-andc2 169
bit-cnt 168
bit-cnt-if 168
bit-cnt-if-not 168
bit-concat *111*, 166
bit-count *116*, 168
bit-eqv 169
bit-every 166
bit-fill *109*, 166
bit-ior 169
bit-left-reduce 166
bit-length *109*, 166
bit-map *112*, 166
bit-maxpref 168
bit-maxprefix *118*, 168
bit-maxsuff 168
bit-maxsuffix 168
bit-merge *120*, 169
bit-mergecar 169
bit-mergeslot 169
bit-mismat 168
bit-mismat-from-end 168
bit-mismatch *117*, 168
bit-mismatch-from-end 168
bit-nand 169
bit-nmerge *121*, 169
bit-nmergecar 169
bit-nmergeslot 169
bit-nor 169
bit-not 170
bit-notany 166
bit-notevery 166
bit-nreverse *110*, 166
bit-orc1 169
bit-orc2 169
bit-pos 167
bit-pos-from-end 167
bit-pos-from-end-if 167
bit-pos-from-end-if-not 167
bit-pos-if 167
bit-pos-if-not 167
bit-position *114*, 167
bit-position-from-end 167
bit-reduce *111*, 166
bit-rem 167
bit-rem-from-end 167
bit-rem-from-end-if 167
bit-rem-from-end-if-not 167
bit-rem-if 167
bit-rem-if-not 167

bit-remove *113*, 167
bit-remove-from-end 167
bit-replace *110*, 166
bit-reverse *110*, 166
bit-right-reduce 166
bit-scan 168
bit-scan-from-end 168
bit-scan-from-end-if 168
bit-scan-from-end-if-not 168
bit-scan-if 168
bit-scan-if-not 168
bit-scan-over *115*, 168
bit-scan-over-from-end 168
bit-search *118*, 169
bit-search-from-end 169
bit-some *113*, 166
bit-sort *120*, 169
bit-sortcar 169
bit-sortslot 169
bit-srch 169
bit-srch-from-end 169
bit-vectorp 29
bit-xor 169
bitp 29, *82*
bothcasep 98
boundp *36*, *37*, 37
break *248*, *249*, 251
butlast 129
byte 94
byte-position 94
byte-size 94

C
c...r 123
caaaar 123
caaadr 123
caaar 123
caadar 123
caaddr 123
caadr 123
caar 123
cadaar 123
cadadr 123
cadar 123
caddar 123
cadddr 123
caddr 123
cadr 123
car 123
caseq 46, *60*
catch *55*, *60*, 62
catchall *60*, 63
cdaaar 123
cdaadr 123
cdaar 123
cdadar 123
cdaddr 123
cdadr 123
cdar 123
cddaar 123
cddadr 123
cddar 123
cdddar 123
cddddr 123
cdddr 123
cddr 123
cdr 123
ceil 89
cerror *247*, *248*, *249*
char *98*, *108*, *151*, 151
char-bits *97*, 101
char-code *97*, 101
char-downcase *99*, 102, *154*
char-equal *32*, 100, *152*
char-font *97*, 101, *206*
char-greaterp 101
char-int 103, *217*, *218*
char-lessp 101, *153*
char-name 103
char-upcase *99*, 102, *154*
char< 100, *153*
char= *30*, 100, *100*, *217*
char> 100
character 102
characterp 29, *98*
check-arg 250
cl 254
clear-input 218
clear-output 220
clear-screen *225*
close 237, *240*, *242*
closurep 30
clrhash 149
clrhash-equal 149
cnt 115
cnt-if 115
cnt-if-not 115
cntq 115
code-char 102, *102*
comfile 254
compile 253
complex *13*, *25*
complexp *82*
concat 110, *138*, *156*, *163*, *166*, *171*
cond *23*, 43, *45*, *46*, *48*, *60*
condition
  keywords 251
condition-bind 246
condition-setq 247
cons 124, *161*
consp 27
control 104
controlp 104
copy-readtable 211
copyalist 127
copybits *109*, 166
copylist *109*, 126
copyseq 109, *126*, *156*, *163*, *166*, *171*
copystring 155
copysymbol 79

copytree 127, *132*
copyvec 163
copyvec@ *109*, 171
copyvector *109*
cos 86
cosd 86
count 115, *140*, *157*, *164*, *168*, *172*

D
declare *40*, *48*, *50*, *54*, *56*, 72
  keywords 72
defconst 22, *71*, *192*
defmacro *193*, *243*
defprop 77
defsetf *184*, *185*
defstruct *8*, *19*, *23*, *119*, *120*, 185, *209*
  keywords 188
defun *243*
defvar 21, *71*, *243*
del 134
del-if 134
del-if-not 134
delass 145
delass-if 145
delass-if-not 145
delassoc 145
delassq 145
delete *130*, *133*, 134, *142*, *145*
deletef 242
delq 134
delrass 146
delrass-if 146
delrass-if-not 146
delrassoc 146
delrassq 146
deposit-field 95
digit-char 103
digitp 99, *103*
directory-namestring 238
disassemble 254
do *35*, *38*, *47*, 47, *55*, *58*, *61*
do* *47*, 50
dolist *47*, 50, *53*, *54*
dostring 51
dotimes 51
double-float 88
double-floatp 28, *82*
dovector 51
dpb 95

E
elt 108, *138*, *151*, *162*, *166*, *170*
enough-namestring 238
eq 30
  compared to equal 30
eql *25*, 30, *45*, *55*, *81*, *100*
equal *22*, 31, *100*, *124*, *152*, *199*
equalp 32, *82*
error
  keywords 248
error-restart 249
eval *60*
eval-when *193*, *208*, *253*
evenp 82
every 112, *139*, *156*, *163*, *167*, *171*
exp 85
expt 86

F
fasload 244, *254*
faslp 242
fboundp 37
fceil 90
ferror *247*, *248*, 249
ffloor 90
file-author 243
file-creation-date 243
file-name-type 240
file-name-version 240
file-namestring 238
filepos 243
fill 109, *138*, *156*, *163*, *166*, *171*
firstn 129
fixnump 28, *82*
flet *37*, *38*
float 88
floatp 28, *82*
floor *58*, 89, *90*
fmakunbound *37*, 38
force-output 220
forlist 53
forlists *52*, 54
format *155*, *220*, 221, *248*, *249*
forstring 53
forstrings 54
forvector 53
forvectors 54
fquery 232
  keywords 232
fresh-line 220, *225*, *231*
freturn
  implemented by *unwind-stack 64
fround 90
fset *37*, 38
fsymeval 37
ftrunc 90
funcall 39, *60*
funcall* 40, *60*
function 36
functionp 29
funny-charp *100*

G
gcd 85
gensym 80
gentemp *80*
get 76
get-dispatch-macro-character 214
get-macro-character 213
get-output-stream-string 236

get-package 80
get-pname 78
getf *76*
gethash 148
gethash-equal 149
getl 76
global-declare 71
go *47*, *48*, *50*, 57
graphicp 98, *99*, *103*
greaterp *30*, *32*
grindef *221*

H
haipart 94
haulong 93
host-namestring 238
hyper 104
hyperp 104

I
if *23*, *33*, 44, *44*, *45*, *61*
if-for-spice *210*
if-in-spice *210*
inch 217, *218*, *232*, *234*
inch-no-hang 218
inchpeek *215*, 217
input-stream-p 237
int-char 103, *217*
integerp 27, *82*
intern *30*, *79*
intersect 136
intersection 136
intersectq 136
isqrt 86

J

K

L
labels *37*, *38*
lambda *55*, *60*
last 125
lastn 130
ldb 94
ldb-test 95
ldiff 130, *134*
left-reduce 111, *139*, *156*, *163*, *167*, *171*
left-vreduce 163
left-vreduce@ 171
length 109, *124*, *156*, *163*, *166*, *171*
lengthf *243*, 243
lessp *30*, *32*
let 41, *42*, *43*, *55*, *56*, *60*
let* 42, *56*, *60*
line-out 220
list 125
list* 126
list-cnt 140
list-cnt-if 140
list-cnt-if-not 140
list-cntq 140
list-concat *111*, *126*, 138
list-count *116*, 140
list-elt *108*, *125*, 138
list-every 139
list-fill *109*, 138
list-left-reduce 139
list-length *109*, 124
list-map *112*, 139
list-maxpref 140
list-maxprefix *118*, 140
list-maxprefq 140
list-maxsuff 140
list-maxsuffix 140
list-maxsuffq 140
list-merge *120*, 141
list-mergecar 141
list-mergeslot 141
list-mismat 140
list-mismat-from-end 140
list-mismatch *117*, 140
list-mismatch-from-end 140
list-mismatq 140
list-mismatq-from-end 140
list-nmerge *121*, 141
list-nmergecar 141
list-nmergeslot 141
list-notany 139
list-notevery 139
list-nreverse *110*, *127*, 127
list-pos 139
list-pos-from-end 139
list-pos-from-end-if 139
list-pos-from-end-if-not 139
list-pos-if 139
list-pos-if-not 139
list-position *114*, *134*, 139
list-position-from-end 139
list-posq 139
list-posq-from-end 139
list-reduce *111*, 139
list-rem 139
list-rem-from-end 139
list-rem-from-end-if 139
list-rem-from-end-if-not 139
list-rem-if 139
list-rem-if-not 139
list-remove *113*, *133*, *134*, 139
list-remove-from-end 139
list-remq 139
list-remq-from-end 139
list-replace *110*, 138
list-reverse *110*, 127
list-right-reduce 139
list-scan 140
list-scan-from-end 140
list-scan-from-end-if 140
list-scan-from-end-if-not 140
list-scan-if 140

list-scan-if-not 140
list-scan-over *115*, 140
list-scan-over-from-end 140
list-scanq 140
list-scanq-from-end 140
list-search *118*, 141
list-search-from-end 141
list-setelt *108*, 138
list-some *113*, 139
list-sort *120*, 141
list-sortcar 141
list-sortslot 141
list-srch 141
list-srch-from-end 141
list-srchq 141
list-srchq-from-end 141
list-to-string 130, *155*
list-to-vector 130
listen 218, *218*
listp 27, *123*
ln 86
load 243, *244*
log 86
logand 91
logandc1 92
logandc2 92
logbitp 93
logcount 93
logeqv 91
logior 91
lognand 92
lognor 92
lognot 92
logorc1 92
logorc2 92
logtest 92
logxor 91
long-float 88
long-floatp 28, *82*
lowercasep 98, *102*

M
make-array *24*, 175, *192*, *208*
  keywords 175
make-bit-vector 162
make-broadcast-stream 236
make-concatenated-stream 236
make-dispatch-macro-character 214
make-echo-stream 236
make-equal-hash-table 149
make-hash-table 148, *149*
make-io-stream 236
make-list 126
make-string 153, *153*
make-string-input-stream 236
make-string-output-stream 236
make-symbol 79
make-synonym-stream *235*, 235
make-vector *24*, 162
makunbound *36*, *37*, 38

map 112, *139*, *156*, *163*, *167*, *171*
mapatoms *53*
mapatoms-all *53*
mapc 52
mapcan 52
mapcar 52
mapcon 52
maphash 148
maphash-equal 149
mapl 52, *112*
maplist 52
mask-field 95
max 83
maxpref 117
maxprefix 117, *140*, *158*, *165*, *168*, *173*
maxprefq 117
maxsuff 117
maxsuffix 117
maxsuffq 117
mem 133
mem-if 133
mem-if-not 133
memass 144
memass-if 144
memass-if-not 144
memassoc 144
memassq 144
member *23*, 133, *142*, *144*
memq 133
memrass 144
memrass-if 144
memrass-if-not 144
memrassoc 144
memrassq 144
merge 120, *141*, *158*, *165*, *169*, *174*
merge-file-name-type 240
merge-file-name-version 240
mergecar 120
mergef 239
mergeslot 120, *121*
meta 104
metap 104
min 84
minusp 82
mismat 116
mismat-from-end 116
mismatch 116, *140*, *158*, *165*, *168*, *173*
mismatch-from-end 116
mismatq 116
mismatq-from-end 116
mod 90
multiple-value *59*
multiple-value-let *58*, 59, *89*
multiple-value-list *58*, 59
multiple-value-setq *58*, 59
multiple-value-vector *58*, 59

N
name-char 104
namelist 238

keywords 238
namestring 238
nbutlast 129, *130*
nconc *53*, *126*, 128, *130*
nintersect 136
nintersection 136
nintersectq 136
nmerge 121, *141*, *158*, *165*, *169*, *174*
nmergecar 121
nmergeslot 121
not *26*, 32
notany 112, *139*, *156*, *163*, *167*, *171*
notevery 112, *139*, *156*, *163*, *167*, *171*
nreconc *127*, 128, *130*
nreverse *49*, 110, *119*, *127*, *130*, *156*, *163*, *166*, *171*
nset-exclusive-or 138
nsetdiff 137
nsetdifference 137
nsetdiffq 137
nsetxor 138
nsetxorq 138
nsublis 132
nsubst 132
nsubstq 132
nth *108*, 125, *138*, *161*
nthcdr 125
null 26, *32*
numberp 27, *82*
nunion 135
nunionq 135
nunite 135

O
oddp 82
open *235*, 242
or 33, *45*, *61*
ouch 220, *234*
output-stream-p 237

P
pairlis *76*, 142
parse-namestring 239
plist 78
plusp 82
pop 128
pos 114
pos-from-end 114
pos-from-end-if 114
pos-from-end-if-not 114
pos-if 114, *115*
pos-if-not 114, *115*
posass 144
posass-if 144
posass-if-not 144
posassoc 144
posassq 144
position *26*, 114, *134*, *139*, *142*, *144*, *157*, *164*, *167*, *172*
position-from-end 114
posq 114
posq-from-end 114
posrass 145
posrass-if 145
posrass-if-not 145
posrassoc 145
posrassq 145
pprint *221*
prin1 219, *222*
princ 219, *222*
prinstring *155*
print *199*, 219, *234*
probef 242
prog *47*, *50*, 55, *58*, *61*, *62*
prog* *55*, 56, *61*
prog1 *35*, 40, *60*
prog2 *35*, 41, *60*
progn *35*, 40, *48*, *55*, *60*
progv *38*, 43, *60*
psetq 38, *48*, *61*
push 128
puthash 148
puthash-equal 149
putprop *76*, *77*
putpropf *76*

Q
quote 36

R
random 95
rass 143
rass-if 143
rass-if-not 143
rassoc 143
rassq 143
rational 88
rationalize 88
rationalp 28, *82*
ratiop 28, *82*
read *5*, *78*, *79*, *215*, 215, *219*, *222*, *234*
read-delimited-list *213*, 216
read-from-string 218
readfile 244
readline *215*, 217, *232*
reduce 111, *139*, *156*, *163*, *167*, *171*
rem 113
rem-from-end 113
rem-from-end-if 113
rem-from-end-if-not 113
rem-if 113
rem-if-not *53*, 113
remainder 90
remhash 148
remhash-equal 149
remove 113, *133*, *134*, *139*, *156*, *163*, *167*, *171*
remove-from-end 113
remprop 78
rempropf *76*
remq 113

remq-from-end 113
renamef 242
replace 110, *138*, *156*, *163*, *166*, *171*
reset-fill-pointer *177*
return *47*, *48*, *50*, 57, *61*, *62*
return-from *48*, *50*, *56*, 58, *61*
revappend 127, *128*
reverse 110, *127*, *156*, *163*, *166*, *171*
right-reduce 111, *139*, *156*, *163*, *167*, *171*
right-vreduce 163
right-vreduce@ 171
round *85*, 89
rplaca 131
rplacbit 166
rplacd 131
rplachar *98*, *108*, 152

S
samepnamep 78
sample-function 4
sample-macro 5
sample-special-form 5
scalarp *82*
scan 114
scan-from-end 115
scan-from-end-if 115
scan-from-end-if-not 115
scan-if 114
scan-if-not 114
scan-over 114, *140*, *157*, *164*, *168*, *172*
scan-over-from-end 114
scanq 114
scanq-from-end 115
search 118, *141*, *158*, *165*, *169*, *173*
search-from-end 118
selectq 45, *46*, *60*
set *36*, *37*, 38
set-dispatch-macro-character 214
set-exclusive-or 137
set-macro-character 213, *214*
set-syntax-from-char 211
setdiff 137
setdifference 137
setdiffq 137
setelt 108, *138*, *152*, *162*, *166*, *170*
setf *37*, *61*, *128*, *129*, *184*, *187*, *188*
setnth *108*
setplist *131*
setq 37, *38*, *47*, *51*, *59*, *61*
setxor 137
setxorq 137
short-float 88
short-floatp 28, *82*
signal 245, *247*
sin 86
sind 86
single-float 88
single-floatp 28, *82*
some 112, *139*, *156*, *163*, *167*, *171*
sort 118, *141*, *158*, *165*, *169*, *173*
sortcar 118
sortslot 118
sqrt 86
srch 118
srch-from-end 118
srchq 118
srchq-from-end 118
standard-charp 98
store-array-leader 178
streamp 237
string 155
string-capitalize 154
string-charp 98, *130*, *151*, *152*
string-cnt 157
string-cnt-if 157
string-cnt-if-not 157
string-cnt= 157
string-concat *111*, 155
string-count *116*, 157
string-downcase 154
string-equal 152
string-every 156
string-fill *109*, 155
string-greaterp 153
string-left-reduce 156
string-left-trim 153
string-length *109*, 155
string-lessp 153
string-map *112*, 156
string-maxpref 158
string-maxpref= 158
string-maxprefix *118*, 158
string-maxsuff 158
string-maxsuff= 158
string-maxsuffix 158
string-merge *120*, 158
string-mergecar 158
string-mergeslot 158
string-mismat 157
string-mismat-from-end 157
string-mismat= 157
string-mismatch *117*, 157
string-mismatch-from-end 157
string-mismatq-from-end 157
string-nmerge *121*, 158
string-nmergecar 158
string-nmergeslot 158
string-not-equal 153
string-not-greaterp 153
string-not-lessp 153
string-notany 156
string-notevery 156
string-nreverse *110*, 155
string-out 220
string-pos 156
string-pos-from-end 157
string-pos-from-end-if 157
string-pos-from-end-if-not 157
string-pos-if 156
string-pos-if-not 156

`string-pos=` 156
`string-position` *114*, 156
`string-position-from-end` 156
`string-posq-from-end` 156
`string-reduce` *111*, 156
`string-rem` 156
`string-rem-from-end` 156
`string-rem-from-end-if` 156
`string-rem-from-end-if-not` 156
`string-rem-if` 156
`string-rem-if-not` 156
`string-rem=` 156
`string-remove` *114*, 156
`string-remove-from-end` 156
`string-remq-from-end` 156
`string-repeat` 153
`string-replace` *110*, 155
`string-reverse` *110*, 155
`string-right-reduce` 156
`string-right-trim` 153
`string-scan` 157
`string-scan-from-end` 157
`string-scan-from-end-if` 157
`string-scan-from-end-if-not` 157
`string-scan-if` 157
`string-scan-if-not` 157
`string-scan-over` *115*, 157
`string-scan-over-from-end` 157
`string-scan=` 157
`string-scanq-from-end` 157
`string-search` *118*, 158
`string-search-from-end` 158
`string-some` *113*, 156
`string-sort` *120*, 158
`string-sortcar` 158
`string-sortslot` 158
`string-srch` 158
`string-srch-from-end` 158
`string-srch=` 158
`string-srchq-from-end` 158
`string-to-list` *130*, 155
`string-to-vector` 155
`string-trim` 153
`string-upcase` 154
`string<` 152
`string<=` 152
`string<>` 152
`string=` 152
`string>` 152
`string>=` 152
`stringp` 29, *151*
`structurep` 29
`sub-bits` *109*, 166
`sub1` 85
`sublis` 132
`sublist` *109*, *129*, 138
`subrcall` *60*
`subrcall*` *60*
`subrp` 29
`subseq` 108, *138*, *156*, *163*, *166*, *171*
`subst` 131
`substq` 132
`substring` *153*, 155
`subvec` 163
`subvec@` *109*, 171
`subvector` *109*
`super` 104
`superp` 104
`sxhash` 150
`symbolp` 26
`symeval` 36

T
`tailp` 134
`terpri` 220, *224*
`throw` *48*, *55*, *60*, 64, *240*, *248*
`tree-equal` *31*, 124
`truename` *238*, *239*, 239
`trunc` *85*, 89, *90*
`tyi` *103*, 217, *218*, *232*
`tyi-no-hang` 218
`tyipeek` 217
`tyo` 220
`typecase` *26*
`typecaseq` 46
`typep` *7*, 26, *184*, *185*, *189*

U
`uncontrol` 105
`unhyper` 105
`uninch` 217
`union` 135
`unionq` 135
`unite` 135
`unless` *23*, *33*, 45, *60*
`unmeta` 105
`unsuper` 105
`untyi` *215*, 217
`unwind-protect` *60*, 63, *242*
`unwindall` 63
`uppercasep` 98, *102*

V
`values` *58*, 58
`values-list` 59
`values-vector` 59
`vcnt` 164
`vcnt-if` 164
`vcnt-if-not` 164
`vcnt-if-not@` 172
`vcnt-if@` 172
`vcnt@` 172
`vcntq` 164
`vcntq@` 172
`vconcat` *111*, 163
`vconcat@` *111*, 171
`vcount` *116*, 164
`vcount@` *116*, 172
`vector-to-list` *130*
`vector-to-string` *155*

vectorp 29
vevery 163
vevery@ 171
vfill *109*, 163
vfill@ *109*, 171
vlength *109*, 163
vlength@ *109*, 171
vmap *112*, 163
vmap@ *112*, 171
vmaxpref 165
vmaxpref@ 173
vmaxprefix *118*, 165
vmaxprefix@ *118*, 173
vmaxprefq 165
vmaxprefq@ 173
vmaxsuff 165
vmaxsuff@ 173
vmaxsuffix 165
vmaxsuffix@ 173
vmaxsuffq 165
vmaxsuffq@ 173
vmerge *120*, 165
vmerge@ *120*, 173
vmergecar 165
vmergecar@ 173
vmergeslot 165
vmergeslot@ 173
vmismat 164
vmismat-from-end 164
vmismat-from-end@ 172
vmismat@ 172
vmismatch *117*, 164
vmismatch-from-end 164
vmismatch-from-end@ 172
vmismatch@ *117*, 172
vmismatq 164
vmismatq-from-end 164
vmismatq-from-end@ 172
vmismatq@ 172
vnmerge *121*, 165
vnmerge@ *121*, 173
vnmergecar 165
vnmergecar@ 173
vnmergeslot 165
vnmergeslot@ 173
vnotany 163
vnotany@ 171
vnotevery 163
vnotevery@ 171
vnreverse *110*, 163
vnreverse@ *110*, 171
vpos 164
vpos-from-end 164
vpos-from-end-if 164
vpos-from-end-if-not 164
vpos-from-end-if-not@ 172
vpos-from-end-if@ 172
vpos-from-end@ 172
vpos-if 164
vpos-if-not 164
vpos-if-not@ 172
vpos-if@ 172
vpos@ 172
vposition *114*, 164
vposition-from-end 164
vposition-from-end@ 172
vposition@ *114*, 172
vposq 164
vposq-from-end 164
vposq-from-end@ 172
vposq@ 172
vreduce *111*, 163
vreduce@ *111*, 171
vref *108*, 162, *178*
vref@ *108*, 170
vrem 163
vrem-from-end 163
vrem-from-end-if 163
vrem-from-end-if-not 163
vrem-from-end-if-not@ 171
vrem-from-end-if@ 171
vrem-from-end@ 171
vrem-if 163
vrem-if-not 163
vrem-if-not@ 171
vrem-if@ 171
vrem@ 171
vremove *113*, 163
vremove-from-end 163
vremove-from-end@ 171
vremove@ *114*, 171
vremq 163
vremq-from-end 163
vremq-from-end@ 171
vremq@ 171
vreplace *110*, 163
vreplace@ *110*, 171
vreverse *110*, 163
vreverse@ *110*, 171
vscan 164
vscan-from-end 164
vscan-from-end-if 164
vscan-from-end-if-not 164
vscan-from-end-if-not@ 172
vscan-from-end-if@ 172
vscan-from-end@ 172
vscan-if 164
vscan-if-not 164
vscan-if-not@ 172
vscan-if@ 172
vscan-over *115*, 164
vscan-over-from-end 164
vscan-over-from-end@ 172
vscan-over@ *115*, 172
vscan@ 172
vscanq 164
vscanq-from-end 164
vscanq-from-end@ 172
vscanq@ 172
vsearch *118*, 165

`vsearch-from-end` 165
`vsearch-from-end@` 173
`vsearch@` *118*, 173
`vset` *108*, 162, *178*
`vset@` *108*, 170
`vsome` *113*, 163
`vsome@` *113*, 171
`vsort` *120*, 165
`vsort@` *120*, 173
`vsortcar` 165
`vsortcar@` 173
`vsortslot` 165
`vsortslot@` 173
`vsrch` 165
`vsrch-from-end` 165
`vsrch-from-end@` 173
`vsrch@` 173
`vsrchq` 165
`vsrchq-from-end` 165
`vsrchq-from-end@` 173
`vsrchq@` 173

W
`when` *23*, *33*, *44*, 44, *60*
`with-open-file` *235*, 240, *242*, *254*
  keywords 241

X

Y
`y-or-n-p` 231, *232*
`yes-or-no-p` 231, *235*

Z
`zerop` 82