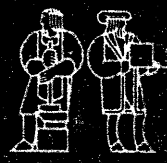
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-311

NIL REFERENCE MANUAL

Glenn S. Burke
George J. Carrette
Christopher R. Eliot

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

NIL Reference Manual

corresponding to

Release 0.286

January 1984

Glenn S. Burke

George J. Carrette

Christopher R. Eliot

This report describes ongoing research at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for this research was provided by the Defense Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract N00015-75-C-066 1 and N00014-83-K-0125, and by Digital Equipment Corporation of Maynard Massachusetts, with grants of equipment.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

Abstract

This document describes NIL, a New Implementation of Lisp. NIL is currently under development on the DEC VAX under the VAX/VMS operating system.

Acknowledgments

The chapter on **defstruct** is a workover of the chapter appearing in [3], by Alan Bawden; added inaccuracies are solely the fault of GSB, however. The chapter on the **loop** macro is a revision of an earlier published memo [5] which also has appeared in Lisp Machine manuals [12].

The chapter on *flavors* was written in part by Patrick Sobalvarro. The editor and its documentation are the work of Christopher Eliot.

The interfaces to many functions and facilities, and some of the terminology used in this document, are taken or derived from those used in the COMMON LISP manual [1]; in particular, the terminology used for describing *scope* and *extent* in chapter 3.

The section on **format** (section 19.6, page 187) is a reworking of a chapter appearing in [3], parts of which had earlier appeared in [12] and are the work of Guy Steele.

Dedication

This publication is dedicated to Randy Davis, may his 750 never crash.

Note

Any comments, suggestions, or criticisms will be welcomed. Please send Arpa or Chaos network mail to BUG-NIL@MIT-MC.

Those not on the Arpanet may send U.S. mail to

Glenn S. Burke
Laboratory for Computer Science
545 Technology Square
Cambridge, Mass. 02139

The Arpa network mail distribution list for announcements pertaining to NIL is normally used for announcements about the facilities described here. Contact the author to be placed on it.

Document History

Earlier versions of this document were distributed as *NIL Notes for Release 0* (in two revisions), and *NIL Notes for Release 0.259*. This update corresponds to NIL release 0.286, Lisp system version 286. It will be revised and reissued with each new NIL release; because of the constant changes, it is not the picture-perfect copy we would like.

Additional Publications

This manual, and other publications, are available from the Lab for Computer Science Publications office

```
MIT LCS Publications
545 Technology Square
Cambridge, MA 02129
```

who should be contacted directly for bibliography, price, and ordering information. Publications of the MIT Artificial Intelligence Laboratory may also be of interest to some; they should be contacted separately, as `MIT AI Lab Publications`, at the same building address.

Summary Table of Contents

Table of Contents

1. Introduction

NIL, which stands for New Implementation of Lisp, is a dialect of LISP which runs on the DEC VAX. NIL currently runs under the VMS operating system. It will likely be converted to run under UNIX (TM) at some point, but there is no effort underway to do so right now.

NIL is a dialect of COMMON LISP. COMMON LISP is essentially a formal specification of the LISP language such that programs which conform to that specification may be transported without modification from one COMMON LISP implementation to another, and be expected to run compatibly. As of this writing, the COMMON LISP manual has just gone to press; this manual has therefore developed into documentation of a subset of COMMON LISP facilities which are currently supported by NIL, and a large number of NIL-specific things.

This document schizophrenically attempts to cover three areas. One is "primer" documentation; those things which must be known for any programming to get done. In this case, attempts are made to point out what of these things are COMMON LISP compatible. Another is the set of things which might be expected to change incompatibly, *due* to COMMON LISP conversion. The third is those which are part of the NIL core *Virtual Machine*, as it is being developed more formally. These include, for example, functions like %string-replace, which are suitable low-level primitives for a VAX (or other byte-machine, like perhaps an IBM-370) to provide. Lastly, there are certain parts of NIL which have undergone large amounts of recent development, and are fairly stable, and which may provide functionality for users in various domains; the I/O system, for example. Much of the provided documentation will be of things which are obscenely low-level; sometimes, this is to point out places where the implementation falls short of the design; often too, to document these for those who may find it useful debugging, or in performing implementation-dependent activities; and occasionally, to explicitly note how the implementation differs from the general and portable semantics (as in the case of numbers and eq).

1.1 Conventions

All otherwise unqualified rational numbers in this manual are in *decimal*, not *octal* (as has been the practice in certain other manuals, notably [9] and [12]). Special qualifying syntax for forcing the interpretation of rational numbers in other radices is described in section 2.1, page 3.

There are a couple conventions for the forming names of functions and variables coming into use from within the COMMON LISP community.

Generally, variables whose values are special and which may be modified during the course of program operation have names which begin and end with an asterisk; for instance, *package*. Lexical variables, and constants defined with defconstant (page 24), are normally named without these asterisks. For instance, NIL defines the constants pi and most-positive-double-float. This is a convention only and does not affect the operation of the NIL interpreter or compiler; however, the possibility exists that (in the future) the compiler will use the presence or absence of such asterisks to choose a course of action if it encounters a free reference to an undeclared variable. (Special and lexical variable reference is discussed in chapter 3, page 11.)

The names of predicate functions in NIL and COMMON LISP are typically formed by suffixing the character "p" to the end of a descriptive name. For instance, **consp** is a predicate which is true if its argument is a cons, and **lessp** (which actually is a MACLISP, not COMMON LISP, function) compares numbers and (when given two arguments) returns t if the first is less than the second. If the name itself is hyphenated, then "-p" is suffixed: **upper-case-p** is a predicate which tells if a character is upper-case. If, however, the predicate name is formed by prefixing a specializing name to an existing predicate name, then the final "p" would not have hyphenation added to it: **string-lessp** is a predicate can be used to compare strings using a standard collating sequence, the name being formed by prefixing "string-" to "lessp". There are, of course, many exceptions to this, and this convention does not eliminate all ambiguity, but it helps. A good number of NIL functions did not follow this convention in earlier versions of NIL; many of these have been fixed, and the old names made synonymous with the new names for the time being.

2. Data Types

This chapter provides an overview of some of the data types used in NIL, their uses, and their syntax (their *printed representation*). Those not strongly familiar with LISP should go over this lightly to get some idea of the sorts of objects which NIL offers, and proceed to the next chapter; others might want to read it anyway to see what NIL provides which may differ from other LISP dialects. Most of the sections here have later chapters devoted to them, which give much more complete information on the types, how they may be used, and the functions which can manipulate them. Also, more complete information on the syntax used for these types is presented in chapter 20, page 213.

2.1 Numbers

The NIL (and COMMON LISP) hierarchy of numerical types looks like this:

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

Collectively, the non-complex numbers are referred to in NIL as the type non-complex-number; the term real is not used because of potential confusion with floating-point. Note that there is no guarantee that the above types might not be further subdivided or grouped for the convenience of the implementation.

2.1.1 Rationals

The *integer* data type is intended to represent mathematical integers. There is no magnitude limit on them other than that imposed by memory or addressing limitations.

In NIL, those integers which can be fit in a 30 bit field in twos-complement are *fixnum*s, which are represented in such a way that no storage is consumed. For integers not in this range, *bignum*s are used. Generic arithmetic functions automatically choose the appropriate representation.

The printed representation of integers ordinarily uses decimal notation, optionally preceeded by a sign character and optionally followed by a decimal point. The sharpsign reader macro (section 20.1, page 214) may be used to input rational numbers in other radices; for instance,

```
259             ;The integer 259
259.            ;The integer 259
#o403           ; entered in octal
#b100000011     ; entered in binary
```

A *ratio* is the type used to represent non-integer rational numbers. It consists logically of integer components which are its *numerator* and *denominator* (which are accessible by functions of the same names). The external interface is defined such that a ratio will always appear to be in reduced form (whether or not it is), and the denominator will always be positive. (COMMON LISP sez it can't be zero, infinity freaks.) The arithmetic routines which deal with rational numbers tranparently convert between ratios, bignums, and fixnums as appropriate. Ratios are denoted separating the numerator and the denominatory by a /: thus, the ratio three-halves is 3/2.

2.1.2 Floating-point Numbers

COMMON LISP allows for four kinds of floating-point representations, which must meet the following criteria:

Format	Minimum Precision	Minimum Exponent Size
Short	13 bits	7 bits
Single	24 bits	8 bits
Double	50 bits	8 bits
Long	50 bits	8 bits

NIL provides all of these formats, with the following specs:

Type	Precision	Exponent
short-float	19 bits	8 bits
single-float	24 bits	8 bits
double-float	56 bits	8 bits
long-float	113 bits	15 bits

The long-float type requires microcode support to avoid software emulation.

Floating-point numbers in NIL are syntactically distinguished from integers in that they *must* have a decimal point followed by at least one digit. (This is more rigid than the COMMON LISP specification.) So, for instance, 10 and 10. are both the integer ten, but 10.0 is floating-point 10. The various formats of floating-point number are syntactically distinguished by the use of the character used in exponential notation. For example, 10.0d0 is double-float ten; 10.0s0 is short-float ten, 10.0l0 is long-float ten, and 10.0e0 is single-float ten. When exponential notation is not used, the type of float is determined by the user-modifiable variable *read-default-float-format* (page 72). COMMON LISP specifies that the default type of float is single-float, both for readin and the values of various irrational or trancendental mathematical functions when they are given rational arguments (e.g., sqrt). However, in NIL, the default is currently double-float, because this NIL release is the first to supply any format *other* than double-float. The default *will* be changed in a future release.

2.1.3 Complex Numbers

Complex numbers in NIL represent a point in the complex plane in cartesian coordinates. Their printed representation shows these coordinates:

#C(realpart imagparg)

The real and imaginary parts may be extracted with the realpart and imagpart functions.

The real and imaginary components of complex numbers must both be either rational numbers, or floating point numbers of the same floating-point format.

Rational numbers and complex numbers with rational components collectively constitute the gaussian-rational NIL data type. Many, but not all, rational number functions have been extended to operate on gaussian rationals in NIL, for instance numerator, denominator, and mod. In order to provide what we call a *seamless extension* of the rational numbers to the complex plane, a gaussian rational with a zero imaginary part is not of type complex, but just the rational number. This interconversion may be compared with that which interconverts between fixnums and bignums as necessary. Gaussian rationals which have integer components (i.e., integers, and complex numbers with integer components) are the NIL type gaussian-integer, which is of course a subtype of gaussian-rational. Certain integer functions in NIL have been extended to operate on gaussian integers; these include gcd, oddp, and evenp. The gaussian rational and gaussian integer extensions in NIL are not defined by COMMON LISP, and should therefore be considered experimental and potentially subject to change. For this reason, feedback on their usage and utility is desired in order that these extensions can be evaluated.

Complex numbers with floating point components always have components of the same floating-point format. Such a number is always of type complex, even if the imaginary part is zero.

2.2 Characters

NIL provides a data type for representing characters. Characters are the things one manipulates when doing "character I/O" on streams. They are the things one gets out of, and puts into, strings. Having a separate data type allows them to maintain their identity within the lisp (as opposed to being an interpretation placed on fixnums, for instance). Chapter 11 is devoted to this.

Characters in NIL use #\ syntax for input and output, as shown below. Note that if the character after the #\ stands alone, it is taken literally. If it occurs after a prefix such as "control-", then it will be treated like an ordinary token, so may need to have a preceding backslash to inhibit case translation or just to allow proper token parsing.

```
#\a                    ;Lowercase "a".
#\A                    ;Uppercase "a".
#\Control-a            ;Uppercase "a", with the control bit.
#\Meta-\a              ;Lowercase "a", with the meta bit.
```

Some characters have names, which may be used in place of the character itself:

```
#\Rubout               ;The "rubout" or "delete" character
#\Hyper-Space          ;The "space" character with the hyper bit.
```

Only a subset of all possible characters are allowed to be contained in strings. These comprise the string-char data type. It happens that in NIL these are those characters which have no font or bits attributes (both are 0).

2.3 Symbols

Symbols are what are used as names in lisp. They can name functions, and variables (the two uses of which are syntactically distinguishable by the LISP evaluator). Symbols have a *name*, also called the *print name* or *pname*, which is a string containing the characters used in the printed representation of the symbol. A symbol also has a *property list* or *plist* associated with it. This is a list of alternating "indicators" and "values", allowing one to store unidirectional associations on the symbol. A symbol also has a *package*, which points to the "name space" it is associated with (chapter 15, page 121).

The symbol nil is special. It is used both to represent *boolean false*, and the *empty list*. Its alternate printed representation is (). the empty list. It has the data type null, which is both a subtype of symbol and a subtype of list, and is the only object of that type. Its value is not allowed to be changed. Otherwise, it is treated the same as other symbols (it has a property list etc.).

The symbol t is used to represent *boolean truth*. Its value is also not allowed to be changed.

Symbols are often used as keywords. Because of the existence of multiple namespaces (packages), this might present a problem because two symbols read into different namespaces might not be the same. This is solved by having special *keyword symbols*, or just *keywords* for short. A symbol which is typed in preceded by a colon (and nothing else) is read into the namespace (package) for keywords. Thus, all symbols so designated are the same (they are eq). Keywords are self-evaluating, and their values are not allowed to be modified.

Notationally, tokens which cannot be interpreted as anything else are taken to be *symbols*, except that tokens consisting entirely of unslashified dots are supposed to cause a syntax error. Thus, 1.0e+4 will read as a floating-point number, but 1.0e+4a will read as a symbol. More complete details on input syntax are in chapter 20, page 213.

2.4 Lists and Conses

A cons is the type which makes LISP what it is. In simpler lisps, it may be the only data type which can be used to associate more than one object. A cons makes a binary association: it has two components, its car and its cdr, which are accessed by just those functions. It is thus the datastructure of choice for representing binary trees.

A list, when considered to be a data-type, is the union of the types cons and null—that is, it is either a cons, or the symbol nil (the empty list). That is why the alternate printed representation of nil is (). (See section 20.1, page 214 for exposition of the printed representations of conses and lists.)

Often, a list is a conceptual sequence which has a discrete end. In this context, the **cdr** of the last **cons** of the list must be **nil**. In text (and error messages), the phrase "proper list" is used to distinguish between just a **cons** (or **nil**) and an actual list whose final **cdr** is **nil**. For instance, the cons

```
(a . b)
```

is of the type **list**, but if given as a list argument to the **member** function (page 61) would cause an error which would say that it was not a proper list.

2.5 Arrays and Sequences

Arrays in NIL are a very general type. One dimensional arrays are the type **vector**. Arrays can be specialized as to the types of elements they may contain. A one dimensional array (a vector) which can only contain "string characters" (see the **string-char-p** function, page 19) is a *string*. A one dimensional array which is allowed to hold only objects of type **bit** (that is, the integers **0** or **1**) is a *bit vector*. Arrays are discussed fully in chapter 12, page 103.

The type **sequence** is the union of the types **list** and **vector**. There are a large number of *sequence functions* in NIL, which operate on both lists and vectors, viewing either as just a sequence of objects. One therefore has available the same general sequence operations, to be used on whatever particular type of datastructure was chosen for the task at hand; they may be lists, to make adding, deleting, splicing, etc., easy, or vectors of a particular type suitable for the application—strings, bit vectors, etc. The functions for operating on sequences are described in chapter 7, page 48.

2.6 Structures

NIL provides a *structure* or *record* definition facility. This is supplied by the **defstruct** function (page 125), which is essentially the same one used in both MACLISP and LISP MACHINE LISP. In NIL, the normal way **defstruct** is used defines a type, and the structures created can be distinguished with **typep**. Additionally, such types interface to the NIL flavor system, which may be used to give them methods for such things as how they should print and pretty-print.

2.7 Functions

There are several different things which can be "functions" in NIL. Most basically, there is the type **compiled-function** (also known as **subr**). This corresponds to a function created by the NIL compiler, or part of the NIL kernel. It may also be created "on the fly" for the purpose of interfacing compiled code to interpreted code; in NIL, functional evaluation of an interpreted function will result in an object of type **compiled-function**.

Functional objects which implement "environment transfer" (which is discussed in a later chapter) are of the type **closure**. The most commonly seen specialization of this type is that used in the interpreter, the type **interpreter-closure**.

2.8 Random Internal Types

Here are some of the internally used types in NIL. While they should generally not be seen, they may pop up on occasion either themselves or as a result of errors.

2.8.1 Minisubrs

The minisubr type is somewhat gratuitous; it will be flushed as a separate type someday, and its type bits reused by something more useful. A *minisubr* is a special routine within the NIL kernel; such routines are called with the VAX JSB instruction. However, they tend to have various assorted

2.8.2 Modules

A *module*, as a type, is the structure used by NIL to contain a collection of compiled functions and the constants and datastructures they reference. When the compiler compiles a file, it produces a module. When the garbagecollector (haha) relocates things, it relocates the module as a block. The use of the name module for this primitive datastructure is a bit pretentious, so it will probably be called something like compiled-code-module in the future.

2.8.3 Internal Markers

The type si:internal-marker is used for various things in NIL, none of which should ordinarily be visible to (or touched by) the user. Objects of this type are meant to be checked for by things like the debugger and garbage-collector (to, for instance, parse stack frames), and manipulating them out of context will confuse these programs.

These objects print out as #! followed by the name followed by !. For instance,

```
#!AFM-3!                    ;Stack marker for 3-arg function
                            ; call frame
#!PC-MARK!                  ;Next slot on stack is a PC
#!DOUBLE-FLOAT-MARK!        ;Next two stack entries are the
                            ; representation of a double-float
#!NULL-ARG!                 ;Stack slot is for an argument which
                            ; has not yet been computed, in a
                            ; function call frame.
```

2.8.4 Unused Types

There are a number of unused type codes in NIL. Certain internal routines, upon encountering them, bomb out to the VMS debugger because your NIL is then undoubtedly losing its lunch.

2.9 Coercion

coerce *object type*

If *object* is already of the type *type*, then it is simply returned; otherwise, it is converted to be of that type.

Only certain forms of coercion are defined. **coerce** will perform coercion of one sequence type to another; its capabilities in this regard are similar to those of **concatenate** (page 50). Note, however, that **concatenate** will *always* return a copy of the sequence, whereas **coerce** will only create a new one if *object* is not of the proper type already.

coerce also allows some non-sequence coercions, with the following types:

float

Coerce the object to a floating point number. If *object* is already a float, it is returned; otherwise, it is coerced to the default type of float (double-float). See **float**, page 78.

short-float
single-float
double-float
long-float

Coerce the object to that particular type of floating point number. Again, this is similar in functionality to what may be obtained by giving **float** a second argument.

complex

object, which must be a number, is coerced to a complex number. If it is already complex, it is returned. If it is rational, it is also returned; this is because complex numbers with rational components and a zero imaginary part are automatically reduced to rationals. If, however, it is a floating-point number, then a complex number with *object* as its real component and a floating-point zero of the same format as *object* as its imaginary component, is returned.

(complex float)
(complex short-float)
(complex single-float)
(complex double-float)
(complex long-float)

Effectively, these are like coercing the number to **complex**, and then returning a complex number whose **realpart** and **imagpart** have been coerced to the specified floating point type.

character

Convert *object* to a character. This coercion is only defined for integers (see **int-char**, page 96), symbols one character long, and vectors whose element-type is a subtype of **character** (i.e., character vectors and strings). Moreover, in the

integer case, an error is signalled if the coercion does not succeed (i.e., **int-char** would return nil when given the integer *object*).

Note that **coerce** does not provide for coercion to rational or integer types. This is because the issue of what to do about truncation or rounding is a matter of the intent; the functions **rational** (page 78) and **rationalize** (page 78) may be used to convert numbers into rational numbers, and the functions **floor**, **ceiling**, **truncate**, and **round** (section 10.7, page 79) are useful for converting both floating-point numbers and ratios to integers with various sorts rounding behaviour.

coerce does not accept all of the general forms of *type* that it should: however, most the simple formats, and certainly all those that are listed above, are handled properly.

When compiled, calls to **coerce** with a constant second argument are changed by the compiler into calls to a routine specific to the task; a few of these, most notably the conversions to (non-complex) floating point, are handled especially efficiently.

There are a number of other functions which perform specific types of coercions. For instance, **string** will coerce a symbol or a string to a string.

3. Scope, Extent, and Binding

The NIL interpreter uses lexical scoping. What this means, simply, is that variable references which are "textually within" the code which binds them, are valid. Those references which are not "textually within" the binding form are not, and will (typically) cause unbound-variable errors. Consider the definition

```
(defun make-associations (keys single-value)
   (mapcar #'(lambda (key) (cons key single-value)) keys))
```

which takes a list of keys, and returns an association list associating all of those keys with the same single value (perhaps for use by **assoc**). The first argument to **mapcar**, the lambda expression, is technically a function. (The #' construct is explained below.) It is, however, textually within the binding of the argument **single-value**, so that variable reference is lexical, and that function works in NIL as desired. Consider the alternative form

```
(defun make-associations (keys single-value)
   (mapcar #'make-one-association keys))
(defun make-one-association (key)
   (cons key single-value))
```

which might appear to be equivalent. The reference to **single-value** in the definition of **make-one-association** is *not* textually within the binding of that variable, hence appears "free". Although this function (in the absence of extra declarations, as described below) would function "properly" in the MACLISP or LISP MACHINE LISP interpreters, it will not in NIL. It is interesting to note that (again without special declarative information) both the MACLISP and LISP MACHINE LISP compilers will treat the second example as an error (or at least produce incorrect code), because although the interpreters do not enforce lexical scoping rules, code is compiled that way.

A short note may be in order on the #' construct which appeared above. #' is an abbreviation for (**function** ...), just as ' is an abbreviation for (**quote** ...). In MACLISP, the two are equivalent. However, in NIL (and to some extent in LISP MACHINE LISP too), use of this special form is necessary to cause the proper (functional) interpretation of the form being evaluated. In fact, in the **make-associations** example, it is that special interpretation which makes the lexical reference to **single-value** "work". If **quote** was used instead of **function**, the example would not work as desired. **function** (or #') need not just be used around lambda expressions. It may also be used around function names (as in the second **make-associations** example). The effect of evaluating (**function** *name*) is equivalent to what the interpreter does when it "evaluates" *name* in the function position of a list being evaluated.

NIL does not restrict one to using only lexical scoping rules. It is possible to declare to NIL that a variable is *special*, and should be able to be referenced by code *not* textually within the binding construct. Or, perhaps a variable should have a global toplevel value and not be bound anywhere, or maybe even have a toplevel value, and be bound in some places. This is the purpose of the **special** declaration, which NIL implements compatibly with COMMON LISP, and which is about the same as it is in LISP MACHINE LISP and MACLISP.

Most of the time, *special variables* are declared to be special globally. This means that the NIL interpreter (and compiler) will always treat the variable as being special, even if there is no declaration for it at the place it is bound. As a matter of style, variables declared **special** are usually given names which begin and end with the character * so that they can be visually

distinguished from more "ordinary" lexically scoped variables. One way to globally declare a variable special is with defvar (page 24). For instance,

```
(defvar *leaves*)
(defun find-all-leaves (tree)
  (let ((*leaves* nil))          ;Empty set of leaves
    (find-all-leaves-1 tree)     ;Grovel over the tree
    *leaves*                     ;And return the leaves found
    ))
(defun find-all-leaves-1 (tree)
  (cond ((atom tree)
         (cond ((not (memq tree *leaves*))
                (setq *leaves* (cons tree *leaves*)))))
        (t (find-all-leaves-1 (car tree))
           (find-all-leaves-1 (cdr tree)))))
```

There are more esoteric (or SCHEME-like) ways in which the above could have been performed, without the use of the special variable *leaves*, but the above is fairly straightforward, fairly efficient, and will also run (both interpreted and compiled) compatibly in MACLISP and LISP MACHINE LISP.

The above intuitive (or, if you prefer, hand-waving and vague) description can now be used to more formally define the terms of *scope* and *extent* which are used to describe the accessibility and lifetimes of things, of which variable bindings are one instance. The *scope* of something tells *where* it may be validly referred to. To say that something has *lexical scope* then means that it may be used anywhere "textually" within the construct which "creates" the object (e.g., the lambda-expression which binds a variable). Note that this does not in itself imply that the reference becomes invalid if that construct is exited. That dimension is the *extent* of the object, which tells the *time* during which the object (e.g., variable binding) is valid. *dynamic extent* means that the object (reference) is only valid during the execution of the construct. *indefinite extent* means that there is no such limitation. Variable bindings in the NIL interpreter (which are not special) have lexical scope and indefinite extent. This means upward funarg capability.

indefinite scope means that there is no restriction on where a valid reference may occur from. This is the case with special variables; the "free" references may be made from any piece of code. The bindings of such variables, however, have only *dynamic extent*; they become invalid (are "unbound") when the binding construct is exited. This combination of scope and extent, which is quite common, is referred to as *dynamic scope*.

Now, for the pragmatics. The current NIL compiler actually only implements local scope instead of lexical scope. Its capabilities lie only in determining when it is losing. In many cases, this does not matter because the constructs being used are expanded out into other constructs, making the references local. This is what happens for mapcar, for instance: in the construct

```
(let ((zz (computate)))
   (mapc #'(lambda (x) (mumblify x zz)) some-list))
```

the reference to zz within the lambda expression is a non-local (but lexical) reference. That expression is recoded by the compiler, however, as an iteration without a separate function, in which the reference become local.

reference This is actually a moderately standard way to handle lexical variables; rewrite the form when possible to cause the reference to become local. The MACLISP compiler does this with the mapping functions: even if the open-code-map switch is turned off, if such a reference occurs it will expand out the iteration to allow the local reference.

Environment transfer is implemented with *closures*. A closure is essentially an encapsulation of a function, and some portion of a binding environment. The closures with which lexical environment transfer is performed in the interpreter, *interpreter closures*, bundle up the lexical environment as of the time of their creation. Thus,

```
(setq fn (let ((x 5)) (function (lambda () x))))
  => #<Interpreter-Closure (Anon) 1 259ABC>
(funcall fn)
  => 5
```

One may test for a closure in general with (typep *x* 'closure), or with the closurep predicate (page 19).

NIL actually has the capability for giving "dynamic" variables *indefinite* extent. This can be used to implement old-fashioned closures as created by the Lisp Machine closure function (which exists in NIL).

In NIL, what has been said for *variables* as far as scope, extend, binding, and shadowing is concerned, is equally true for *functions*. Variable value and function value are handled in virtually identical fashions. The primary differences between the two are that the interpreter does not warn you when you create a new toplevel special function value (it does when you create a new toplevel special variable value when the variable is not globally declared special), and the compiler makes its special assumption quietly.

The design of the NIL binding mechanism is described by White in [13].

3.1 Lambda Application

Application of a lambda expression in NIL is much like that of LISP MACHINE LISP. A lambda expression is of the general form

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

In the simplest case, *lambda-list* is a (possibly empty) list of variable names, which are the formal parameters to the lambda expression when it is treated as a function. There must be as many arguments to the lambda-expression as there are variables. Thus,

```
((lambda (a b c) (list a b c)) 1 2 (+ 3 4))
 => (1 2 7)
```

The lambda-list may also contain special symbols which begin with the character &. These are often called *lambda list keywords*, but they are "keywords" only in the general sense, i.e., they are not typed in with a preceding colon. They are typically used to drive the matching of the formal parameters (variables) in the lambda list with the values they should be bound to. There are basically just four such keywords, each of which is optional, and which should appear in the order they are shown in:

&optional

The items from the &optional to the next &-keyword (or end of the lambda-list) describe

optional arguments to the function. Each such item may be of one of the following forms:

variable

If a corresponding argument is supplied, then *variable* will be bound to that. Otherwise, it will be bound to nil.

(*variable*)

Same as an isolated *variable*.

(*variable init-form*)

If there is a corresponding argument, then *variable* is bound to that. If not, then *init-form* is evaluated, and *variable* bound to that result. The evaluation of *init-form* is performed in an environment where all of the variables in the lambda list to the left of this one have been bound already.

(*variable init-form init-p-var*)

Just like the previous format. Additionally, *init-p-var* will be bound to t if there was an argument supplied, nil if not.

&rest

There must be exactly one item between an **&rest** keyword and the next &-keyword (or the end of the lambda-list). This variable is bound to a list of all the remaining arguments to the function.

&key

The items between **&key** and either **&aux** or the end of the lambda-list describe *keyworded arguments* to the function. These are arguments which are passed by keyword rather than by position: when given, it must be preceded by the keyword naming which argument it is. The function fill is defined with the lambda list

```
(sequence item &key (start 0) end)
```

which means it takes two required arguments (*sequence* and *item*), and two keyworded arguments (*start* and *end*). The calls

```
(fill sequence new-item :start start :end end)
(fill sequence new-item :end end :start start)
```

are effectively the same. All keyworded arguments are by default optional. The specification of a keyworded argument in the lambda list is normally the same as that of an optional argument; thus, if no :start keyword and associated value is specified in a call to fill, the *start* parameter defaults to 0, and for *end*, the default is nil. The name of the variable is used to generate the keyword which flags that particular parameter. Additionally, with the non-atomic forms of optional parameter specification, a list of the actual keyword which should be used and the variable to bind the argument to may be used instead. For example, if it were desired that the keyword :start be used to flag the starting index, but that the formal parameter be named i, then the lambda-list could have been written as

```
(sequence item &key ((:start i) 0) end)
```

It is important to note that if both &key and &rest are given, then the list the &rest variable is bound to is *the same* list from which the keyworded arguments are extracted. This is sometimes useful if the arguments are going to be passed en-mass to some other function using apply, or perhaps to reconstruct the original arguments to the function in order to signal an error.

&aux

Bindings specified with **&aux** are for *auxiliary variables*; they have no correspondence to the "arguments" given to the lambda expression. The only things which may follow **&aux**

in the lambda list are bindings specs for these auxiliary variables, which may take one of the following forms:

variable

variable will be bound to **nil**.

(*variable*)

variable will be bound to **nil**. However, because this syntax may eventually be either disallowed or made to mean something else, one should use either just *variable* or (*variable* **nil**).

(*variable init-form*)

init-form is evaluated, and *variable* bound to the result.

The first **&aux** *init-form* is evaluated within the environment produced by the preceding portion of the lambda list. As each **&aux** binding specification is processed, the variable is bound, and will be available to any following *init-forms*. Because the stuff with **&aux** has little to do with the lambda application, it may be clearer for the body of the lambda expression to be wrapped in **let** (page 26) or **let*** (page 26); in fact, the portion of the lambda list following **&aux** could be given as the binding-list to **let***, and have the same meaning.

The use of **&rest** in NIL results in consing. If the keyword **&restv** is used in place of **&rest**, then the variable will be bound to a *stack vector* rather than to a list. This is an object which is of the type **simple-vector**, but has only *dynamic extent*; it loses its validity when the function with **&restv** in its lambda-list is exited. Essentially, the stack vector is just a pointer into the stack where the values are stored. This feature should be used with great care.

4. Predicates

4.1 Type Predicates

4.1.1 Type Specifiers

A *type specifier* is an expression which may be used to express a data-type constraint.

nil No object is of the type nil; nil is a subtype of all types.

t All objects are of the type t. For instance,

```
(make-array '(10 10) :element-type t)
```

makes an 10x10 array which can hold objects of any type.

type-name
This is the most common form of type specifier; just a type name, for instance number, double-float, string. *type-name* may be the name of a flavor defined with defflavor (page 173), the name of a structure defined by defstruct (page 125) (assuming defstruct is not told to implement the structure in some other fashion), or one of the many NIL types noted in various places in this document.

(not *type-specifier*)
All objects which are *not* of type *type-specifier*.

(and *ts1 ts2 ... tsn*)
The intersection of the types corresponding to the given type specifiers.

(or *ts1 ts1 ... tsn*)
The union of the types corresponding to the given type specifiers.

(member *x1 x2 ... xn*)
This describes a type which is one of the objects *x1*, *x2*, ... *xn*. Equality is defined by eql (page 20).

(satisfies *function-name*)
An object is a "member" of this type if *function-name* returns a non-null value when applied to it, otherwise it is not.

There are some more complex forms which are used variously as synonyms for, and constraints on, more general types. For instance:

(integer *low high*)
An object is of this type if it is an integer between *low* and *high*. *low* and *high* may be integers, in which case the boundary check is *inclusive*, lists of integers, in which case the boundary check is *exclusive*, or the atom *, which signifies infinity of the appropriate sign. Thus, (integer 0 *) is a type specifier for all non-negative integers, and (integer (0) *) or (integer 1 *) for all positive integers.

(signed-byte *size*)
(unsigned-byte *size*)
An object is of these types if it can be represented in the appropriate form in twos-

complement notation in a field of the specified size. (Without a hidden-bit convention.) Thus, (signed-byte 3) is the same as (integer -4 3), and (unsigned-byte 3) is the same as (integer 0 7).

bit Either 0 or 1.

(array *element-type dimensions***)**

An object is of this type if it is an array with "the same" element type and dimensions as those specified. *dimensions* may be the symbol * (or not supplied), which matches an array of any rank and dimensions. Otherwise, it should be a list: it then matches an array whose rank is the length of the list. The elements of the list may be either the symbol *, or a non-negative integer which is the size of the corresponding dimension. For instance, the dimensions list (3 4 5) refers to a 3x4x5 array, and (3 * 5) an array whose first dimension is 3, second is of any size, and third is 5.

There are two different contexts the type specifier can be used in, and they affect the interpretation of the *element-type*. These are

declaration

in which something is being declared to take on values of this type (declarations are discussed in chapter 6). What the type specifier says is that the type is the same as would be returned by make-array were it given an element-type of *element-type* and dimensions conforming to *dimensions*.

discrimination

which is really just use of the type specifier as a second argument to typep. In this case, the question being asked is whether the object being tested is of *exactly* this type.

These two "questions" are different because of the way make-array (page 103) interprets its *element-type* argument, which is fully discussed in section 12.2, page 104. Basically, what you get from make-array is an array whose element-type is the closest specialization to the specified element-type which make-array can provide.

Finally, the *element-type* can be *, which means any element type at all. Thus, (array t *dims*) is a not a subtype of (array * *dims*), because the former only refers to arrays which can hold elements of any type, whereas the latter includes bit arrays, string-char arrays, etc.

(simple-array *element-type dimensions***)**

This is like (array *element-type dimensions*), but additionally states that the array has been created without any "fancy options"; see chapter 12, page 103.

(vector *element-type size***)**

This is equivalent to (array *element-type* (*size*)).

4.1.2 General Type Predicates

typep *object* &optional *type-specifier*

If only one argument is supplied, this is (somewhat) like MACLISP typep, and returns the exact implementation type of *object*. In NIL, this is usually something too specific to be worth looking at.

Otherwise, returns t if *object* is of type *type-specifier*, nil otherwise. See the description of type specifiers, above.

subtypep *type-specifier-1 type-specifier-2*

subtypep attempts to determine if *type-specifier-1* is a subtype of *type-specifier-2*. It returns two values; the *second* will be t if the relation could be determined, nil if it could not be. The first is t if the relation can be determined and *type-specifier-1* is in fact a subtype of *type-specifier-2*, nil otherwise.

In NIL, if the type specifiers are atomic type names and *type-specifier-1* is a subtype of *type-specifier-2*, then subtypep will probably succeed in determining this. If *type-specifier-1* is *not* a subtype of *type-specifier-2*, then subtypep may fail to be able to determine the relation, because it may not know that particular supertypes are mutually exclusive.

For instance,

```
(subtypep 'fixnum 'number)       => {t, t}
(subtypep 'character 'number)    => {nil, t}
(subtypep 'number 'fixnum)       => {nil, t}
(subtypep '(satisfies fixnump) 'number)  =>  {nil, nil}
```

4.1.3 Specific Type Predicates

null *object*

This returns t if *object* is nil, nil otherwise. Stylistically, null is used to test for *object* being the empty list, whereas not (page 28), which is functionally equivalent because of the empty-list/boolean-false duality of nil, is used to test for boolean falseness. This is why constructs of the form

`(if (not (null x))` *frob-non-null-x frob-null-x*`)`

are so prevalent.

symbolp *object*

Returns t if *object* is a symbol, nil otherwise.

consp *object*

Returns t if *object* is a cons, nil otherwise. This is the same as (not (atom *object*)).

listp *object*

consp or null.

fixnump *object*
characterp *object*
These test for the exact types fixnum and character.

string-char-p *character*
Tells if *character* is a character which may be stored in a string. This will be a type of sorts in COMMON LISP, such that a string is a vector with elements of type string-char. Note that string-char-p requires its argument to be a character, as opposed to just any object; it is not really a general type predicate.

In NIL, this is characters with *bits* and *font* of 0 (see chapter 11).

stringp *object*
vectorp *object*
Tell if *object* is a string or vector respectively. These are equivalent to doing (typep *object* 'string) and (typep *object* 'vector).

numberp *object*
floatp *object*
These are the same as (typep *object* 'number) and (typep *object* 'float) respectively.

closurep *object*
Tells if *object* is a closure.

The following are supported for MACLISP compatibility:

pairp *object*
This is synonymous with consp. Once upon a time, the name for cons in NIL was pair, and the function pairp was installed in MACLISP. Now, NIL and COMMON LISP call a cons a cons.

bigp *object*
(typep *object* 'bignum)

fixp *object*
Equivalent to integerp, i.e., (typep *object* 'integer). This name should be avoided because of possible confusion with fixnump.

flonump *object*
This is the same as (typep *object* 'double-float), for historical reasons, *not* the same as floatp. It is provided for MACLISP compatibility only.

4.2 Equality Predicates

Note also null (page 18) and not (page 28), for testing for nil.

eq *x y*

This tells if *x* and *y* are the exact same object. Implementationally, this is true if *x* and *y* are the same "pointer". For instance,

```
(setq *foo* (cons 'a 'b))        =>  (a . b)
(eq *foo* *foo*)                 =>  t
(setq *bar* (cons 'a 'b))        =>  (a . b)
(eq *foo* *bar*)                 =>  nil
```

Philosophically, what this predicate says is that if one can side-effect the object *x*, then the equivalent side-effect will happen to *y* simultaneously; in other words, they are functionally indistinguishable. There are certain kinds of objects which have absolutely no internal structure, and thus cannot be side-effected. These objects have the behaviour that two of them created the same will then be **eq**. As a rule, for code transportability, resilience, and clarity, this behaviour is something which should not be depended on. In NIL, it happens that objects of type **fixnum** and **character**, among some other more obscure ones, exhibit this behaviour; this may not be true in other LISP implementations, however (it is not in MACLISP, for instance). For comparisons of such objects, **eq** is not the proper test; **eql** is.

eql *x y*

eql is a predicate for testing for equality on non-structured objects. It is true if *x* and *y* are both numbers of the same type and numerically equal, or if *x* and *y* are both objects of type **character** and represent the same character, or (otherwise) if they are the same object (**eq**). This is the default predicate for many functions such as **member** and **subst**, and also for the **case** special form (page 29).

equal *x y*

Fairly standard **equal**. Numbers, characters, symbols, and most random types are compared as by **eql**. Conses are **equal** if their **car**s and **cdr**s are **equal**. Arrays are considered **equal** only if they are **eq**, with the following exceptions:

(1) Two strings are **equal** if they have the same length and their corresponding characters are **equal**. Note that they therefore will not be **equal** if any of those characters differ in case, making this incompatible with LISP MACHINE LISP usage.

(2) Two bit vectors are **equal** if they have the same length, and the same elements.

(3) In NIL, and in NIL only (not COMMON LISP), two general vectors are **equal** if they have the same length and **equal** components. The lengths are determined as by **length**, that is, the fill pointer of a vector is used if it has one. This exception to the COMMON LISP definition is to compensate for an earlier definition of **equal** which compared arrays by comparing their elements if they had the same rank, dimensions, and element-types, and should not be depended upon.

equal is implemented recursively, so may fail to terminate (or blow out with stack overflow) on circular structures.

equalp *x y*

This is a more "general" version of **equal**. Numbers are considered to be **equalp** if they are numerically equal; type conversion will occur if needed to perform the comparison (see =, page 73). Characters are compared with **char-equal** (page 95), so are considered **equalp** even if they differ in case. Conses are **equalp** if their respective **cars** and **cdrs** are **equalp**. Arrays are **equalp** if they have the same rank and dimensions, and if their corresponding elements are **equalp**: thus, a string will be **equalp** to a general vector containing **equalp** characters:

```
(equalp #(#\F #\o #\o) "foO")  =>  t
```

equalp, like **equal**, is implemented recursively so may die on circular structures.

5. Programming Constructs

The NIL special and toplevel forms.

5.1 Compositions

progn *{form}**

progn evaluates each of the forms, and returns the value of the last. It is, therefore, a way to get multiple expressions in a position where only one is called for.

```
(progn)  <=>  nil
(progn form)  <=>  form
```

and

```
(progn form-1 form-2 ... form-n)
```

evaluates all of the *forms*, returning the result of the evaluation of *form-n*.

Back in the olden days of LISP, many special forms only allowed a single expression where we now allow multiple expressions to be evaluated in sequence; for instance, in the consequents of the cond special form (page 28). This often necessitated the explicit use of the progn special form in such places, and is where the term *implicit progn*, which describes a situation where such multiple forms are allowed, comes from.

prog1 *first-form {form}**

prog1 evaluates *first-form*, and saves the value; then it evaluates all of the other *forms*, but instead of returning the value of the last like progn does, it returns the value saved from the the evaluation of *first-form*.

prog2 *first-form second-form {form}**

This is entirely equivalent to

```
(progn first-form (prog1 second-form form...))
```

In other words, it evaluates all of the forms in order, and returns the value (it saved) of the second.

5.2 Definition Forms

5.2.1 Defining Functions

defun *name lambda-list {declaration}* [documentation] {form}**

Defines *name* as a function, equivalent to a lambda expression formed using the lambda list, declarations, and forms.

For MACLISP-compatibility, the following idiosyncratic syntaxes are recognized specially:

(1) If *lambda-list* is the atom macro, then this is assumed to be a MACLISP-style macro definition, and is transformed appropriately.

(2) If *lambda-list* is a non-null atom, then this is considered to be a MACLISP lexpr definition. *name* will be defined as a function of a variable number of arguments,

and the symbol *lambda-list* bound to that number. The functions arg and setarg may then be used (compatibly with MACLISP) to access the arguments.

(3) If *lambda-list* is the atom fexpr, then this is assumed to be a MACLISP-style fexpr definition, and an attempt is made to turn it into a NIL special form. Note, of course, that due to evaluator semantics this will usually not work (calls to eval will, for instance, utilize a new lexical contour).

In principle, *name* is a general function specifier ("function-spec") as is done in LISP MACHINE LISP. However, these have not been put into NIL yet, so one additional idiosyncratic syntax (also MACLISP compatible) is recognized: *name* may be a list of the form (*name propname*), in which case the function is placed on the *propname* property of *name*. That is, one can do something like

```
(defun (foo hack) (x y)
  (list (* (sinh x) (cos y)) (* (cosh x) (sin y))))
```

with the result that

```
(funcall (get 'foo 'hack) x y)
```

will invoke a compiled function to perform that computation.

5.2.2 Defining Macros

A LISP *macro* differs from an ordinary function in that the code of the macro is run, not to obtain a value for the form, but rather to obtain a *new* form to be used in place of the original form. In LISP this is not done through any strange and miraculous string-processing and substitution, but by LISP code itself; LISP program code is just LISP data, which can be manipulated and constructed by ordinary LISP programs. When a *macro call* is encountered by a LISP compiler, the code for the macro is run then and there, during the compilation, to construct the new form which must be compiled instead. For that reason, LISP macros, while general LISP functions, should not depend on their dynamic environment (although if properly arranged they may have global declarative or definitional information around).

defmacro *name pattern* {*declarations*}* {*documentation*}* {*form*}*

This is the preferred way for macros to be defined. *name* is defined as a macro. When a call to *name* is encountered by the interpreter or compiler, the list of arguments to *name* (specifically, the cdr of the calling form) is matched against *pattern*; the *forms* are then evaluated in an environment where the variables specified by *pattern* are bound to the components of the arguments which they match, and the resulting value is used in place of the original form.

pattern is generally a pattern of symbols and conses, but it may also have in it, intermixed, &optional, &rest, and &body. (&body is treated just like &rest by defmacro pattern matching. However, it is supposed to be an indication to a code formatter (pretty-printer, or an editor) that the following forms are a "body" rather than just a sequence of more arguments.) The following defines foo as a macro to be synonymous with car:

```
(defmacro foo (x)
  (list 'car x))
```

In NIL, the &*mumble* keywords need not be "top-level" within the pattern:

```
(defmacro with-output-to-string ((var &optional string)
                                 &body form)
  ...)
```

Especially useful for constructing code in the bodies of macros is the backquote reader macro, section 20.1.1, page 216.

Eventually, defmacro will be extended to support these additional lambda-list keywords:

&key *keyspec1 keyspec2 ...*
: Pretty much like the ordinary use of &key in lambda lists (section 3.1, page 13).

&whole *variable*
: Binds *variable* to the macro call form, so that one may get a handle on the entire form.

macro *name bvl {declarations}* [documentation] {form}**
: Primitive macro definition. You probably shouldn't use this, at least not for routine macro definitions.

The macro function receives one argument, which is the macro call form; thus, *bvl* should be a lambda list for a function of exactly one argument. It is possible that this will change in the future, so one should stick to defmacro if at all possible.

5.2.3 Defining Variables

defvar *var [init [documentation]]*
: Globally declares *var* to be special. If there is an *init* form specified, then when this form is loaded (evaluated), if *var* is not already bound (dynamically), it will be set to the value of *init*.

defparameter *var init [documentation]*
: This is like defvar, only *var* is *always* set to the value of *init*.

defconstant *var init [documentation]*
: Similar, and additionally states that the value of *var* is not intended to change. A correctable error is signaled if, when this form is loaded (evaluated), *var* has a value not equal to the value of *init*. (Continuing from this error will set *var* to its new value; continuing from similar errors not signalled by defconstant may not do the update.)

The NIL compiler will, at its discretion, utilize the (defined or implied by *init* being a constant) value of *var* inline. So if you will be changing the value of a defconstant variable out from under other compiled code, you should perhaps be using defparameter.

5.2.4 Controlling Evaluation Time

Macros often need to return multiple forms to be processed as if they all appeared independently at toplevel. For instance, **defvar** and its variants could all be trivially implemented as macros (if they aren't already). In NIL, they may be returned within a **progn** special form. For instance, a simplified **defparameter**, which did not handle documentation, could have been written like this:

```
(defmacro defparameter (variable value-form)
  '(progn (proclaim '(special ,variable))
          (setq ,variable ,value-form)))
```

proclaim and declarations are described in chapter 6, page 41. The MACLISP hack where the first element of the **progn** had to be the form **(quote compile)** is not needed (but will work just fine).

This behaviour and the special casing only applies to toplevel forms.

eval-when *kwd-list* {*form*}*

This is as if each of the *forms* appeared at toplevel, but were only there to be processed at the times specified by *kwd-list*. The allowable keywords for *kwd-list* are

eval
When the **eval-when** form is evaluated by the interpreter.

load
When the *forms* of the **eval-when** are loaded compiled (they will be treated as if they were seen at toplevel by the compiler; e.g., **defuns** will be compiled, etc.).

compile
The *forms* will be evaluated immediately when processed by the compiler. Note that **eval-when** keywords are *not* symbols in the keyword package (that is, you don't type them in with a colon in front).

Note that for historical reasons, these "keywords" are only keywords in the general sense—they are not symbols in the keyword package (typed in with a colon prefix).

5.3 Binding and Assignment

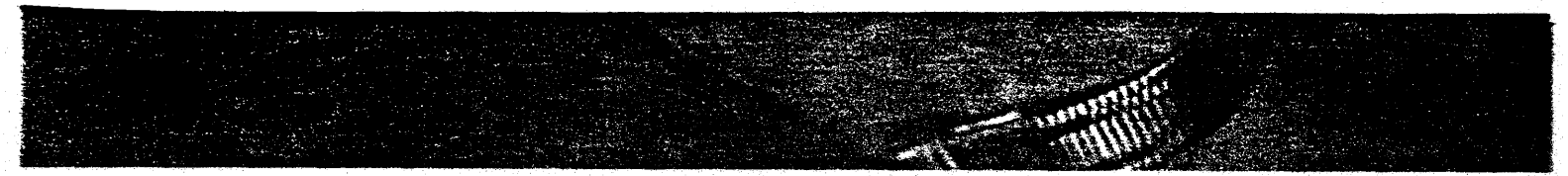
setq {*variable value*}*

setq changes the values of variables.

```
(setq a (f) b (g))
```

makes the value of **a** be the result of evaluating **(f)**, then makes the value of **b** be the result of evaluating **(g)**. The value returned by **setq** is the last value stored/computed, which in this case is what **b** now will evaluate to, and what **(g)** evaluated to.

The choice of whether to change the lexical or dynamic value of the variable which **setq** makes is the same as that which would be made by the evaluator in evaluating the variable; and if there *is* no binding of the variable, **setq** creates a global dynamic value for it.



psetq {*variable value*}*

psetq is syntactically like setq, but it updates the variables in parallel. That is, all of the *values* are computed, before any of the *variables* are side-effected. A common idiom which uses this is

```
(psetq x y y x)
```

which exchanges the values of x and y.

psetq always returns nil.

let ({*variable value*}*) {*declaration*}* {*form*}*

let evaluates all of the *values*, and then binds all of the *variables* to the corresponding values. All of the values are computed before any of the bindings are performed.

```
(let ((a (f)) (b (g)))
  forms...)
```

evaluates (f) and (g), then binds a to the result from the evaluation of (f) and b to the result of the evaluation of (g), and then evaluates all of the *forms*; the value of the last form is returned as the value of the let.

Various other constructs in NIL accept a list of lists of variables and values syntactically the same as that used by let. This is what is meant by the term *letlist*.

In NIL, let will accept, in place of a variable, a pattern used for *destructuring*. The variables within the pattern are bound to the corresponding parts of the value. This is similar to the interface to destructuring used by defmacro; see it, page 23, for more information on this pattern-directed destructuring.

Because COMMON LISP let is not defined to support destructuring, it is recommended that, if destructuring is used, it be hidden in a macro. This will make it both easier to read (all the extra parentheses needed to use let with destructuring make it hard to read), and also make it easier to change should let eventually be changed to *not* support destructuring (at which time there will be a primitive provided which does). For instance, the NIL compiler defines the macro debind-args to destructure argument lists of forms being compiled:

```
(defmacro debind-args (arglist-pattern form &body body)
  `(let ((,arglist-pattern (cdr ,form))) ,@body))
```

Eventually NIL will provide some special forms and functions for getting at the destructuring and argument-list matching functionality provided by defmacro.

let* ({(*variable value*)}*) {*declaration*}* {*form*}*

Syntactically, let* is similar to let. However, rather than binding the variables in *parallel*, it binds them *sequentially*. That is, when each *value* form is evaluated, the corresponding *variable* is bound to that value, and the following *values* are evaluated in that environment. For instance,

```
(let* ((a form1) (b form2))
  compute)
```

first evaluates *form1*, and binds a to that value. Then, it evaluates *form2*, and binds b to *that* value. Finally, it evaluates *compute* and returns that value as the value of the let*.

5.3.1 Dynamically Binding Variable Variables

The routines in this section may be used to bind variables which are not known at code-writing (or compile) time. They all bind the special (dynamic) value of the variable.

There are four variations on how the variables may be bound. Which is used is a matter of which is best suited for the particular task at hand. Basically, the variations all involve how the value is computed.

progv *varlist vallist forms...*
progv binds each of the variables in *varlist* to the corresponding values in *vallist*. If there are too many values in *vallist*, the extras are ignored. If there are too few, then the extra variables will be bound "unbound". Each of the *forms* is then evaluated in an environment in which the variables are so bound; they are all bound dynamically.

progv is defined by COMMON-LISP. None of the following special forms are, however.

The following three special forms all take an a-list as a specification of what variables to bind to what values. The car of each entry in the a-list is the variable to be bound. If the cdr is nil (the entry has a length of 1), then the variable will be bound "unbound". Otherwise, the cadr (*not* the cdr) of the entry tells what value the variable should be bound to: the exact interpretation varies with the special form in use. In all cases, the bindings are performed sequentially: each variable is bound to its value before the value for the next variable to be bound is determined.

progw *a-list forms...*
The second element of each entry in *a-list* is a form to eval to get the value to bind the variable to.

progwq *a-list forms...*
This is sort of a trivial special-case of progw. The second element of each entry in *a-list* is the actual value to bind the variable to; no evaluation is performed.

progwf *a-list forms...*
This function may not yet be in the existing NIL.

This is a variant of progw more in keeping with a LISP implementation in which things get compiled. The second element of each entry in *a-list* is a function to be called on no arguments to determine the value to bind the variable to. This is much more practical than progw because it can eliminate a lot of interpreter overhead.

The typical use of these special forms is to provide a dynamic environment, the specification of which may be augmented modularly, and hence might not be totally known to the writer of the code which performs the binding. progw and its variants are usually more convenient for this purpose than progv, because the specification of the environment is in a single datastructure, which might be kept around as the value of a variable. Note however that progw, progwq, and progwf are *not* defined by COMMON-LISP. progw is defined by LISP MACHINE LISP, but progwq and progwf are not.

5.4 Conditionals

if *predicate consequent* [*elseform*]

if evaluates *predicate*. If the result is not false (i.e., not nil), then the result of the if is the result of evaluating *consequent*. Otherwise, if *elseform* is specified, the result of the if is the result of evaluating *elseform*, otherwise nil.

not *x*

not is used logically to invert the sense of a predicate. That is, it is by convention used to test for the object representing *boolean false*. Because of the empty-list/boolean-false duality of the symbol nil, it is functionally equivalent to null (page 18), which logically checks for the *empty list* which is represented by the type named null. Thus, one often sees constructs of the form

```
(if (not (null l)) consequent elseform)
```

because null is used to check for empty-listness (for instance, being at the end of an iteration down a list), and the not is used to invert the sense, so that the *consequent* will be run if there is in fact something left to the list l.

cond {(*predicate* {*consequent*}*)}*

General historical cond. Each *predicate* to the cond is evaluated, in order. If the result of an evaluation is false, then the cond evaluates the corresponding *consequents* in that "clause", returning as its value the value(s) of the last one, unless there were no *consequents*, in which case the value of the cond is the value of the *predicate* evaluation.

```
(cond (p1 c1)
      (p2 c2)
      (t e))
```

is equivalent to

```
(if p1 c1
    (if p2 c2 e))
```

however cond allows multiple *consequents*, and also may more clearly show the selection by clearly listing the sequentially processed tests.

If all of the *predicate*s are false, then cond returns nil.

when *predicate-form* {*consequent-form*}*

```
(when predicate-form
  consequent-1
  consequent-2
  ...
  consequent-n)
 ==>
(cond (predicate-form
        consequent-1
        consequent-2
        ...
        consequent-n))
```

If *predicate-form* evaluates non-null, then the *consequent-forms* are evaluated in order and the value(s) of the last one returned as the value of the when form; otherwise, the when form returns nil.

unless *predicate-form* {*consequent-form*}*

```
(unless predicate-form
  consequent-1
  consequent-2
  ...
  consequent-n)
 ==>
(when (not predicate-form)
  consequent-1
  consequent-2
  ...
  consequent-n)
 ==>
(cond ((not predicate)
       consequent-1
       consequent-2
       ...
       consequent-n))
```

and {*form*}*

Evaluates each *form*, and if any returns nil, and immediately returns nil without evaluating any subsequent forms; otherwise, the result of the and is the value(s) of the last *form*. (and) => t.

or {*form*}*

Evaluates each *form*, and if one returns a non-null result, that value is returned by or without evaluating any of the following forms. (or) => nil.

or is supposed to return exactly one value, no matter how many were produced by the evaluation of a *form*, *except* for the last *form* which is evaluated tail-recursively (with respect to multiple value propagation). It doesn't currently behave quite this way.

case *keyform* {(({*key**}) {*consequent*}*)}*

A dispatch form utilizing the eql predicate. In general,

```
(case keyform
  ((key-1-1 key-1-2 ...) form-1-1 form-1-2 ...)
  ((key-2-1 key-2-2 ...) form-2-1 form-2-2 ...)
  ...)
```

is essentially the same as

```
(let ((tem keyform))
  (cond ((or (eql tem 'key-1-1) (eql tem 'key-1-2) ...)
             form-1-1 form-1-2 ...)
        ((or (eql tem 'key-2-1) (eql tem 'key-2-2) ...)
             form-2-1 form-2-2)
        ...))
```

Since the keys are constant, however, it is possible for the compiler to determine the cheapest way to perform the comparisons; see eql, page 20. In fact, if there are a moderate number of fixnum keys in a small range, the NIL compiler may use the VAX CASE instruction to perform the dispatching.

In place of a list of keys, one may use a single atomic key. Also, the symbols t and otherwise are special-cased and cause that "clause" to *always* be selected; no subsequent clauses will be examined. For example,

```
(case (* 2 2)
  (1 'one)
  (2 'two)
  (3 'three)
  (t 'many))
=> many
```

Note that if one needs to use nil, t, or otherwise as keys, they should be enclosed in a list: otherwise, nil will be interpreted as an empty list of keys, and t and otherwise as signifying the "otherwise" clause.

This function is what selectq thought it would once be, and may be used in place of MACLISP caseq.

typecase *object* {(*type-specifier* {*form*}*)}*

typecase examines each of its "clauses" in turn. If *object* is of the type specified by that *type-specifier* (see typep, page 18), then the *form*s in that clause are evaluated, and the value of the last form is returned by typecase. If a *type-specifier* is one of the symbols t or otherwise, then that clause will *always* be selected, and all subsequent clauses will be ignored.

typecase can often produce moderately better code than repeated calls to typep, by factoring out operations needed for more than one of the checks. The NIL typecase does not yet do any clever pointer-type dispatch, however.

Remember that each "key" is a type specifier; it need not be just a type name, and it is definitely not a list of type names. To test for more than one type in one key, use (or *type1 type2* ...).

5.5 Function Invocation

Often the function one desires to call is not constant; for instance, it may have been passed in as an argument (like the second argument to sort, page 55, or as a :test keyworded argument to member, page 61, or something obtained from a property list, hash table, or association list). NIL does not support the old "functional variable" interpretation of a functional form; that is, in a form like (foo x y), the function called is *never* obtained from the value of a variable foo. Instead, because the "name space" for functions and variables is completely distinct, so special functions are provided for invoking functions which are obtained by "normal" evaluation.

funcall *function* &rest *args*

funcall calls *function* on all of the arguments in *args*. It is *not* a special form; the *function* is evaluated in the regular fashion, and that value should be a function. For instance, sort (page 55), which could be defined something like

```
(defun sort (sequence pred &key key) ...)
```

could invoke its *pred* argument with something like

```
(funcall pred (elt sequence i) (elt sequence j))
```

apply *function* &rest *args*

apply is somewhat like funcall, but the last of the *args* is a list of the remaining arguments to invoke *function* on. That is, if apply is called like

(apply *fn a1 a2 ... an list*)

then *fn* will be invoked like

(funcall *fn a1 a2 ... an e1 e2 ... em*)

where *e1 e2 ... em* are the elements of *list*.

Often, the last argument to apply is a list which is the value of an &rest variable. In NIL, this list may instead be a simple vector (it *must* be simple), as might be obtained by use of &restv instead of &rest.

Note that the MACLISP apply function only accepts two arguments, if you don't count the optional third argument which is an evaluation environment.

lexpr-funcall *function* &rest *args*

Old name for what is now done by apply. lexpr-funcall, unlike apply, is compatible with MACLISP.

5.6 Iteration Constructs

5.6.1 Mapping Functions

mapc *function list* &rest *more-lists*
mapl *function list* &rest *more-lists*
mapcar *function list* &rest *more-lists*
maplist *function list* &rest *more-lists*
mapcan *function list* &rest *more-lists*
mapcon *function list* &rest *more-lists*

These are the standard complement of LISP mapping functions, which iterate down all of the lists in parallel. They accumulate results in three different ways, and apply *function* in two different ways. In all cases, if more than one list is supplied, they are stepped in parallel and the iteration terminates when the end of the shortest one is reach.

mapc and mapl each returns its first argument as its value; that is, they are typically for effect. mapc applies *function* to the cars of successive sublists, that is, to the elements of the lists, whereas mapl applies *function* to the sublists themselves. Thus,

```
(mapc #'print '(a b c))
```

prints

```
a
b
c
```

whereas

```
(mapl #'print '(a b c))
```

prints

```
(a b c)
(b c)
(c)
```

Both return *list*, their first argument. Note that the function is not called on the null list, even though that might be thought of as a sublist. Random example where the return value is useful:

```
(mapl #'(lambda (sub1)
          (rplacd sub1 (delete (car sub1) (cdr sub1))))
      some-list)
```

eliminates (destructively) all duplicate elements from *some-list*.

mapl is what used to be called map in MACLISP. map is now a generic sequence function (page 54).

mapcar and maplist each returns a list of the results of applying *function* to their successive arguments; for mapcar, as with mapc, the arguments are the elements of the lists, and for maplist, as with mapl, they are the sublists themselves. For example:

```
(mapcar #'(lambda (x y) (plus x y))
        '(1 2 3) '(9 10 11))
 => (10 12 14)
```

That could have been written as

```
(mapcar #'plus ...)
```

mapcan and mapcon "splice together" the results of applying *function* to its successive arguments, using (essentially) nconc. One common use of mapcan is to mapcar conditionally:

```
(mapcan #'(lambda (x) (and (pred x) (list (f x))))
        some-list)
```

is kind of like doing (mapcar *#'f some-list*) having first deleted those elements which do not satisfy *pred*.

Because of the constrained nature of the iteration, mapcar, maplist, mapcan, and mapcon probably generate better code than you (or even something like the loop macro) could write for generating the return value.

5.6.2 Other Iteration Forms

dotimes (*var count* [*endform*]) {*declaration*}* {*tag* | *form*}*

var is stepped from 0 (inclusive) to the value of *count* (exclusive); for each value, each of the *form*s is evaluated. When the iteration terminates, the value of *endform* is returned. The "body" of dotimes is a "tagbody" body (see tagbody). Atomic tags may be inserted among the forms and jumped to with go (page 34), to produce strange and wondrous unstructured code.

Additionally, dotimes establishes an implicit block named nil around the dotimes form (see block); one may thus use return (or return-from with a block name of nil) to exit and return a value(s) from the dotimes, without running *exitform*. block and tagbody are described in section 5.6.3, page 34.

dolist (*var list* [*endform*]) {*declaration*}* {*tag* | *form*}*
Similar to dotimes, but steps *var* through the elements of *list*, the way mapc (page 31) does.

dovector (*var vector*) {*declaration*}* {*tag* | *form*}*
Similar to dolist, but for vectors. dovector will work on any type of vector; its name and existence dates from the early days of NIL when all vectors were simple vectors. If *vector* is going to be of some known specialized type, one might be able to get better code by using one of the specialized loop constructs available in NIL, as discussed in section 17.7.1.2, page 163.

do ({*var* [*init* [*step*]]}*) (*endtest* {*endform*}*) {*declaration*}* {*tag* | *form*}*
do is a more general iteration construct. The specified variables are bound in parallel (as by let, page 26) to the values of the corresponding initial values. These are their values for the first iteration. For all subsequent iterations, the variables which have step forms specified are all set in parallel to the values of those step forms. (That is, the values of all of the step forms are computed before *any* of the assignments have been made.)

On each iteration, *endtest* is evaluated. If the result is not nil, then the iteration terminates and evaluates the *endforms* as an implicit progn (returning nil if there are no *endforms*). Otherwise, the "body" of the do is interpreted as by tagbody (page 34). Thus,

(do (...) (nil) *tagbody...*)

is idiomatic for looping forever, and

(do (...) (t ...) *tagbody...*)

will never get around to interpreting the forms in *tagbody*.

do establishes an implicit block named nil around the entire do form; see section 5.6.3, page 34.

As a special hack for MACLISP code, if *endtest* is not specified (and hence no *endforms* could be specified), then the body of the do is interpreted exactly once, and nil returned; that is, the do should have been written as prog (page 35), except that MACLISP prog does not accept initial values for the variables.

The MACLISP "old style do" is also supported in NIL. This takes the form

(do *var init step endtest tagbody...*)

and is equivalent to

(do ((*var init step*)) (*endtest*)
tagbody...)

New code should not use this format.

do*
Special form
({(*var* [*init* [*step*]])}*) (*endtest* {*endform*}*) {*declaration*}* {*tag* | *form*}*
This is like do, except that the variables are bound to their initial values sequentially (like let* (page 26) rather than let), and the steps are also performed sequentially (like setq rather than psetq).

See also the loop macro, page 144. loop is a "programmable iteration facility", which allows one to combine various sorts of iterations (such as those provided by dotimes and dovector) with various sorts of result accumulation (such as those provided by mapcar, mapcan, every, some, and other things such as summing and counting). Because it is complicated, it is documented fully in chapter 17, page 144. An earlier version of that chapter appeared as [5], which is expected to be revised similarly.

5.6.3 Block and Tagbody

block and tagbody together implement the flow-of-control functionality provided by standard prog. prog could have been implemented as a macro in terms of these and let, and in fact is described in that fashion by COMMON LISP.

block *name* {*declaration*}* {*form*}*

block evaluates the *forms*. If a lexically apparent return-from is evaluated with a tag of *name* (or *name* is nil and a return is evaluated), then the value(s) of the form given to return or return-from are returned as the value of the block form. Otherwise, the block form returns the value(s) of the evaluation of the last *form*.

Note that the argument to return or return-from is evaluated in the environment in which it occurs, *not* the environment where the block was established.

return *form*

Evaluates *form*, and returns the value(s) it returns from the nearest lexically apparent block with a name of nil. Many special forms implicitly establish blocks named nil, such as prog, do, dolist, dotimes, dovector, and (usually) loop.

Note that this differs subtly from LISP MACHINE LISP. In LISP MACHINE LISP, return returns from the innermost prog (i.e., block) which is *not* named t. In NIL (and COMMON LISP), a return to a block name of nil only matches a block name of nil, and the block name t is not distinguished in any way.

return-from *name form*

Evaluates *form*, and returns the value(s) it returns from the nearest lexically apparent block with a name of *name*. *name* is not evaluated.

tagbody {*tag* | *form*}*

The body of a tagbody is examined sequentially. If a form is atomic, then it is a tag and is ignored, otherwise it is evaluated. If during the evaluation a lexically apparent call to go is evaluated with an argument of one of the tags, then control is returned to that point within the tagbody form, which resumes its interpretation. If the interpretation reaches the end of the tagbody, the result is nil.

go *tag*

tag is not evaluated. Control is returned to the nearest lexically apparent tagbody form with a tag name of *tag*, which resumes interpretation of the tagbody at the form following that tag.

It is important to note that the name matching of block/return-from and tagbody/go is *lexical*. For instance,

```
(defun f (x)
  (block foobar
     (g #'(lambda (x) (return-from foobar x)) x)))
(defun g (fn x)
  (list (block foobar
          (funcall fn x))))
(f 'foo)  =>  foo
```

not (foo). The NIL compiler cannot handle this example, however. Also, the named block only has *dynamic extent*; if an attempt is made to return to a lexically apparent block construct which has been exited, the interpreter will complain.

prog *varlist* {*declaration*}* {*tag* | *form*}*

Standard prog. *varlist* may be a list of variables, or a list of lists of variables and their initial values. They will be bound in parallel. prog can be built from the above primitives:

```
(let varlist
   the declarations
   (block nil
     (tagbody the tags and forms)))
```

For compatibility with LISP MACHINE LISP, if the first "argument" to prog is a non-null atom, then that is used as the name of the block, with the varlist following.

prog* *varlist* {*declaration*}* {*tag* | *form*}*

Like prog, but binds the variables sequentially rather than in parallel, i.e., like using let* (page 26) rather than let.

5.7 Non-Local Flow of Control

catch *tag* {*form*}*

COMMON LISP catch. *tag* is evaluated and the result saved. Then, if during the evaluation of the *forms*, if a throw to that tag (as tested for by eq) occurs, the catch form so named will return the values given to the throw. The tag so named has *dynamic extent*.

The tag to catch is allowed to be any LISP object. This means that one can generate a guaranteed unique tag by (for instance) (ncons nil), or by using a datastructure which is somehow associated with the control point of the catch. This is, in fact, how the NIL interpreter implements the block and tagbody constructs; it uses the datastructures in which it stores its control-flow information as tags to catch.

Note that this is incompatible with the standard MACLISP catch function documented in the 1974 manual ([9]). However, PDP10 MACLISP has been bitching and moaning about use of catch for some year or more now, advising the use of *catch instead (which is equivalent to NIL's catch).

throw *tag form*

tag and *form* are evaluated. Control is returned from the nearest **catch** established with a tag **eq** to the value of *tag*, and that **catch** then returns all the values produced by *form*.

special form because of multiple value passback. of course that doesn't work reliably yet...

Note that this is incompatible with the old-fashioned MACLISP **throw** function. However, in PDP10 MACLISP **throw** has been out of vogue for some time, supplanted by ***throw**, which has syntax and semantics identical to this **throw**, and which is supported in NIL.

unwind-protect *protected-form* {*cleanup-form*}*

protected-form is evaluated, and the result returned. Upon exit, the *cleanup-forms* are evaluated. No matter how the exit is achieved (**throw**, error, whatever).

In principle, **unwind-protect** returns whatever extra values *protected-form* did. In the current implementation, this cannot be guaranteed because no state is saved around the evaluation of the *cleanup-forms*.

***catch** *tag* {*form*}*

Old name for what is now **catch**. Will be supported and identical to what **catch** is now indefinitely, for the sake of MACLISP programs, which use this name with identical syntax. (Note however that MACLISP **catch** allows *tag* to be a list of tags, which means it can't be just any object.)

***throw** *tag form*

Old name for what is now **throw**. Will be supported and identical to what **throw** is now indefinitely, for the sake of MACLISP programs, which use this name with identical syntax.

This seems to be an ordinary function. But then multiple-value passback is unreliable.

5.8 Multiple Values

documentation

NIL contains a kludgey implementation of *multiple values*, similar to what exists in LISP MACHINE LISP. The implementation is based on the hack put into MACLISP some time ago, and suffers from approximately the same deficiencies: namely, that multiple values passed back to forms receiving a single value "normally" might hang around and be picked up later if no other multiple-value passing is done.

values &rest *values*

Returns as many values as it is given arguments; the first value being the first argument, etc. It is permissible for there to be no values.

values-list *list*

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

values-vector *vector*

Because NIL makes such great use of vectors, this is provided also; it returns all the elements of *vector* as multiple values.

For example, in NIL **values** is defined by

```
(defun values (&restv vec)
  (values-vector vec))
```

Of course, the NIL compiler will open-compile a call to **values**.

multiple-value *variables values-form*

The variables in *variables*, which must be a list of variables, are set to the corresponding multiple values returned by the evaluation of *values-form*. Extra values are ignored; if too few values are returned, the extra variables are set to **nil**.

multiple-value always returns exactly one value, the value of the first value returned by *values-form* (or **nil** if none were returned).

In NIL, as in LISP MACHINE LISP, one may use **nil** in place of a variable to cause the corresponding value to be ignored. This is specifically disallowed by COMMON LISP, and should not be used. The preferred way to handle this uses **multiple-value-bind**, below.

The name of this will be changed to **multiple-value-setq** by COMMON LISP...

multiple-value-bind *variables values-form* {*declaration*}* {*form*}*

Somewhat like **multiple-value**, except the variables in *variables* are *bound* to the values produced by *values-form*, and each of the *forms* evaluated in that environment.

Note that, although in NIL **nil** may be used as a placeholder in *variables* for a value which will not be used, it may not in COMMON LISP. The preferred way to ignore a value is to use a name for it, and declare that name to be ignored; for instance,

```
(multiple-value-bind (quo rem) (%bignum-quotient-norm x y)
  (declare (ignore quo))
  (hack-about-with rem))
```

This additionally provides the benefit of having the value "documented" by virtue of being associated with a named variable.

multiple-value-list *form*

form is evaluated, and all of the values it produces are returned as a list. For example:

```
(multiple-value-list (values 1 2))  =>  (1 2)
(multiple-value-list (values))      =>  nil
```

multiple-value-prog1 *first-form* {*form*}*

This is like **prog1** (page 22), but returns all of the values produced by the evaluation of *first-form*. **prog1** is supposed to return only the first value, *never* the other values; however, in the current multiple value implementation NIL uses, this cannot be depended on.

5.9 Generalized Variables

setf {*place value*}*

setf is sort of a generalized setq. Essentially, a setf form expands into the code needed to store each *value* into each *place*. For example, just as

```
(setq x 3)
```

stores 3 into x,

```
(setf (car l) 5)
```

stores 5 into the car of the value of l. As with setq, multiple place/value pairs are handled sequentially.

setf always returns the last value stored.

setf works on variables, all defined car/cdr functions, array and sequence accessing functions (aref, elt, vref, char, bit), get, various attribute accessors (symbol-plist, symbol-value, symbol-function, etc.), and all accessors defined by either defstruct or defflavor. In fact, it is the canonical (and the only formally defined) way to modify slots of structures defined with defstruct. It also operates on a number of low-level NIL primitives, including nibble, get-a-byte, and get-a-byte-2c.

setf also operates on certain other forms whose "inversions" are not strictly side-effecting, by performing the setf on an argument of the function (which must be valid as a *place* to setf). These include ldb and load-byte. For instance, the side-effect performed by

```
(setf (ldb bytespec place) val)
```

is the same as

```
(setf place (dpb val bytespec place))
```

although the return value should be different.

The setf methodology is even more helpful when the logical operation being performed on the *place* is "read-modify-write". This means that the *place* needs to only be specified once in the form.

As of this writing, the COMMON LISP compatible setf has not yet been installed in NIL, so not all of the following macros may be in the NIL when it is released. This pertains to incf, decf, shiftf, and rotatef, and anything having to do with how one might define such a macro. setf, push, and pop have been in NIL for some time already.

push *item place*

Approximately

```
(setf place (cons item place))
```

except that order of evaluation is preserved, and the forms of *place* are evaluated only once.

pop *place*

Approximately

```
(prog1 (car place) (setf place (cdr place))
```

but the forms in *place* are evaluated only once. In otherwords, pop treats the contents of *place* as a list stack; it returns the top of the stack and pops it.

incf *place* &optional (*delta 1*)
Sets *place* to *place* plus *delta*.

decf *place* &optional (*delta 1*)
Sets *place* to *place* minus *delta*.

shiftf *place1 place2 ... placen form*
(shiftf *place form*)
is just like using setf.
(shiftf *place1 place2 form*)
stores the value of *place2* into *place1*, and then the value of *form* into *place2*.

rotatef *place1 place2 ... placen*
rotatef "rotates" the value of the places. The value of *place2* is stored into *place1*, *place3* into *place2* etc., and *place1* into *placen*, all in parallel.

5.10 Property Lists

A *property list* is a list of even length, of alternating indicators and values. This is, of course, the same datastructure which is the property list of a symbol. Note that this is not the same as a *disembodied property list*, which has *odd* length. COMMON LISP has generalized the property list mechanism so that is attached neither specifically to symbols or disembodied property lists, but rather interfaces to *generalized variables*.

Note: as of this writing, the COMMON LISP compatible setf, on which both remf and setf on getf are dependent, has not been installed in NIL. As a result, there is a chance they might not appear in this release.

getf *place indicator* &optional *default*
getf fetches the value under the *indicator* indicator from the property list which *place* evaluates to. If no such indicator exists, then *default* is returned. The standard MACLISP get function could have been written as

```
(defun get (x indicator)
  (typecase x
    (symbol (getf (symbol-plist x) indicator))
    (cons (getf (cdr x) indicator))
    (t error)))
```

It is not really necessary for *place* to actually be a generalized variable. However, getf may be used with setf (and all such similar constructs), in which case *place* obviously must be such a thing. MACLISP putprop could have been written as

```
(defun putprop (x value indicator)
  (typecase x
    (symbol (setf (getf (symbol-plist x) indicator) value))
    (cons (setf (getf (cdr x) indicator) value))
    (t error)))
```

The utility of getf is that it may be used to manipulate "property lists" stored in places other than symbols or the cdr of a cons. One might, for instance, define a structure with defstruct (page 125) which has one slot used to hold a property list:

```
(defstruct (frobozz :named :conc-name)
   initial-data
   plist
   (hyperspace-shift-count 0))
```

Then, items in the "property list" may be accessed by

(getf (frobozz-plist *frobozz*) *indicator*)

and set by

(setf (getf (frobozz-plist *frobozz*) *indicator*) *value*)

One may also do such things as

(incf (getf (frobozz-plist *frobozz*) *indicator*))

to increment a value stored on the property list.

remf *place indicator*

remf removes an indicator/value pair from the property list in *place*. In order to do so, *place* must be a generalized variable usable with *setf*. remf returns nil if there was no such pair for indicator, t otherwise.

get-properties *place names*

get-properties allows one to search for a set of properties. It looks down the property list in *place*; when an indicator is found which is in the list *names*, then it returns three values: the value under that indicator, the indicator, and the sublist of the property list beginning with the indicator. (Thus, the car of that sublist is the indicator, and the cadr the value. This last value is what is returned by the MACLISP getl function, page 69.) If none of the indicators specified in *names* are found, then all three values get-properties returns are nil.

Not only can get-properties be used for searching for more than one indicator at once, but the third value returned can be fed back to get-properties to continue searching from that (or some later) point in the property list.

Note also the property-list function get (page 65) and remprop (page 65).

6. Declarations

In LISP, there is only one declaration that affects program semantics (and is thus the only one needed to make them run correctly): the special declaration. All other declarations are for the purpose of providing extra information about the program—this information may be for the compiler, for a code analyzer, or just as documentation for humans. The assertions made by declarations may be tested by the LISP interpreter or compiler in order to find program errors, and may be used to direct compilation strategies, so *incorrect* declarations are illegal and may cause erroneous results.

6.1 Local Declarations

The normal declaration mechanism associates the declaration information with a particular context established by some special form or construct. This is what happens with a lambda-expression, for instance, which is "described" as

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

A *declaration* is either a form

```
(declare {decl-spec})
```

or a form which is a macro call which expands into such a declare form. Declarations are handled specially by the forms they are within; it is an error for them to occur in other contexts, even though this may not be detected. The various special forms which accept declarations show where those declarations may appear in their descriptions in this manual. Typically, this is just preceding any "body" forms, as in the above examples. A *decl-spec* is a list, the car of which must be a symbol which is recognized as naming some declaration; the possible symbols, and what they mean, are shown a bit below.

Individual declarations fall into two categories: those which affect variable bindings, and those which do not. Those which do, associate with bindings performed by the special construct they are associated with. For instance, both

```
(lambda (a b c) (declare (type long-float c)) ...)
(let ((a (f)) (b (g)) (c (h)))
  (declare (type long-float c))
  ...)
```

say that the variable c is being bound to something of type long-float, and that that particular "instantiation" of c will always have a long-float as its value. The declaration in effect "attaches" to the binding of the variable; only that particular binding is affected. The declaration in

```
(let ((x (f)))
  (declare (type x single-float))
  (let ((x (cons x y))) ...)
  ...)
```

only affects the outer binding of x, and makes no statement about the inner binding. The declaration in

```
(let ((x (f)))
  (declare (type simple-vector v))
  ...)
```

is in error, because there is no variable v being bound in the construct the declare is associated with, let.

The other category of declaration is the *pervasive declaration*, which does not associate with variable bindings. These have lexical scope, delimited by the form the declaration occurs in. This sort of declaration is in effect for all parts of the special form: this includes such things as forms which are evaluated to obtain values for variables being bound, even if those forms are not strictly part of the "body" of the special form. For instance, the optimize declaration in

```
(let ((x (f)))
  (declare (type x single-float) (optimize (speed 3)))
  (let ((x (cons x y)))
    (mapc #'(lambda (z) ...) x)
    ...)
  ...)
```

is "in effect" everywhere within the outer let, unless it were to be locally shadowed by another optimize declaration hidden in one of the forms represented by the elipses. The form (f) to which the x in the outer let is being bound, *is* affected by that declaration. On the other hand, in the similar construct

```
((lambda (x)
    (declare (type x single-float) (optimize (speed 3)))
    ...)
 (f))
```

the form (f) is *not* under the influence of the optimize declaration. However, the form (g) in

```
((lambda (x &optional (y (g)))
    (declare (type x single-float) (optimize (speed 3)))
    ...)
 (f))
```

is within the scope of the optimize declaration. Of course, one doesn't normally write lambda combinations like that "in-line" in code, but associates them with named functions by use of defun; the function definition

```
(defun frobnicate (x &optional (y (g)))
  (declare (type x single-float) (optimize (speed 3)))
  ...)
```

has the same declaration scoping.

6.1.1 The Special Declaration

```
(special var-1 var-2 ... var-n)
```

The special declaration differs from all other declarations in two ways:

- It is the only declaration which affects program semantics, and
- it is the only declaration which can both "attach to" variable bindings, and also be a pervasive declaration.

This duality is partly for convenience, but also follows from what this declaration does.

For those variables in the special declaration which are being bound by the special form the declaration is associated with, for instance x in

```
(let ((x (f)))
  (declare (special x y))
  forms...)
```

the declaration is an *immediate*, or *variable-binding* declaration. It says that the special value of x should be bound, so that x has dynamic rather than lexical scope. As a consequence of this, unshadowed references to x within the *forms* will refer to the special value of x—the one it was bound to, which is no doubt what is desired. This does not affect bindings which are not so declared; in

```
(let ((x ((f)))
  (declare (special x y))
  (let ((x (g)))          ;x number 3
    ... x ...)   ;x number 4
  ...)
```

the inner binding of x is a lexical, rather than special, binding, because there is no special declaration for it. This binding shadows the outer (special) binding of x, with the result that reference to x within that inner let refers to the lexical value, which gives what (g) evaluated to. This, then, is all really the same as how the type declaration associated, as was described above.

If, however, there are variables being declared special for which there no bindings, then the special declaration for them is a *pervasive* declaration, which affects their reference (it will still not affect inner bindings). For instance, in both

```
(let ((x (f)))
  (declare (special x y))
  ... y ...)
```

and

```
(let ((x (f)))
  (declare (special x y))
  (let ((x (g)))
    ... x ...
    ... y ...))
```

the reference to y amidst the elipses is a special reference, due to the special declaration for y. Note that by contrast the reference to x between elipses is lexical because of the shadowing effect of that inner lexical binding.

The example function find-all-leaves from chapter 3, on page 12 (q.v.), can be rewritten as

```
(defun find-all-leaves (tree)
  (let ((*leaves* nil))          ;Empty set of leaves
    (declare (special *leaves*))
    (find-all-leaves-1 tree)     ;Grovel over the tree
    *leaves*                     ;And return the leaves found
    ))
(defun find-all-leaves-1 (tree)
  (declare (special *leaves*))
  (cond ((atom tree)
         (cond ((not (memq tree *leaves*))
                (setq *leaves* (cons tree *leaves*)))))
        (t (find-all-leaves-1 (car tree))
           (find-all-leaves-1 (cdr tree)))))
```

to demonstrate the common uses of the special declaration. The declaration in find-all-leaves is an *immediate* declaration; it causes *leaves* to be dynamically bound. In find-all-leaves-1, it is a *pervasive* declaration which says that the references to *leaves* within that function are references to the special value of *leaves*. It happens that in find-all-leaves-1, both the interpreter and compiler would figure out that the references to *leaves* would have to be special references because they are "free" references (there is no lexically apparent binding), however the compiler would issue a warning about this. So, that declaration serves to tell the compiler, the interpreter, and a reader of the code, that the references to *leaves* in find-all-trees-1 are deliberately special.

6.1.2 Declarations Affecting Variable Bindings

Here are the other declarations NIL accepts which associate with variables being bound. It is an error for a variable to be specified in one of them, when it is not being bound by the construct the declaration is associated with.

(ignore *var-1 var-2 ... var-n*)

This says that the specified variables, although bound by the associated construct, are not actually used. Its purpose is to tell the NIL compiler that you are aware that you will not ever reference the values of those variables; otherwise, it might warn you that the variable is never referenced.

One common use for this declaration is for arguments to functions which are given, but (possibly because of an early state of development) not used. For instance,

```
(defun program-deterministic-p (program)
  (declare (ignore program))
  nil)
```

which defines a function which takes a program as an argument, and returns t if it can be determined that is deterministic and it is, nil otherwise. The other common use is with special binding constructs where something must be specified positionally, but what is obtained from that position is not needed. For instance,

```
(defun mod (x y)
  (multiple-value-bind (quo rem) (floor x y)
    (declare (ignore quo))
    rem))
```

which is how mod could be defined; see page 81—mod is defined to be the second value

returned by the COMMON LISP **floor** function.

In the current NIL compiler, a variable declared with **ignore** but actually referenced, will probably not be detected, and will produce (erroneous) code just as if the variable had never been bound.

(**type** *type var-1 var-2 ... var-n*)
(*type var-1 var-2 ... var-n*)

These declare that the specified variables are of the type *type*. For the first form, *type* may be any valid type specifier (see section 4.1.1, page 16). For the second, only certain type specifiers which are symbols will be recognized: in the current NIL compiler, only **fixnum** is recognized. (The symbols **flonum** and **notype** are recognized for MACLISP compatibility.)

The current NIL compiler simply recognizes this declaration and throws it away.

6.1.3 Declarations Affecting Compilation Strategies

These declarations make no statements about the program, so (in the absence of compiler bugs!) can have no effect other than efficiency on programs which are correct.

(**optimize** (*quality-1 value-1*) (*quality-2 value-2*) ...)

The **optimize** declaration is used to tell the compiler how it should go about making decisions when it compiles code, in a fairly general way; there are four different qualities which may be specified. Each may take on a value which is an integer from 0 to 3 (inclusive); 0 says that that aspect should be discounted completely, and 3 says that it is very important. The default value for each is 1. The qualities are:

speed
: Speed is of the essence. The compiler should try harder to make the code run faster. Obviously, doing so is going to have to trade off against at least one of the other following qualities—if it did not, then the compiler wouldn't have to make a choice.

space
: This quality attempts to quantify for the compiler how important compactness of code is.

safety
: This attempts to quantify to the compiler how important the "safety" of the code is. The exact meaning of this is somewhat hazy; the NIL compiler takes it to mean that operations should detect erroneous inputs and situations when possible, or barring that, at least do things so that they might be less likely to trash your environment irrevocably in such a situation.

compilation-speed
: The speed of compilation. In the NIL compiler, specifying a higher value for this quality will make it do a bit less in the way of minor or special-case optimizations which do not affect program operation all that much, individually.

What the NIL compiler does for the various values of these qualities is discussed in section 24.2, page 246.

(inline *function-1 function-2 ...*)
(notinline *function-1 function-2 ...*)
The inline declaration says that the compiler should attempt to code calls to the named functions "in line"—that is, essentially code the body of the function in place of the call to the function. The notinline declaration says that this should not be done. Because the NIL compiler normally inline-codes anything it knows how to, the inline declaration is really only useful in NIL to shadow a notinline declaration.

There are a couple reasons why one might specify that a function not be coded inline. First, things like trace (page 220) can only "trace" the function call if there actually is a function call. Second, inline compilation, while it does not always eliminate error checking, sometimes aggravates debugging by causing an error to be signalled other than where it would have been in the interpreted code, or (worse yet) causes some other error to happen.

The compiler is free to ignore the inline declaration; if it does not know how to inline a function, it simply cannot do so. However, anything declared notinline will be compiled as a function call. It is an error to specify this for a special form, e.g., cond, and it is probably meaningless and in addition an error to do so for a macro.

It is important to note that many functions in NIL provide an intermediate possibility between complete open-compilation without error checking, and calling the regular function: there is a large body of special subroutines in the NIL kernel which have calling sequences optimized for how the compiler can compile calls to them, but which do error checking. This means that, for these routines, one does not lose error checking by not having the routine coded as a function call, although one does lose trace capability, and the visibility of the function call on the stack when it is examined with the debugger. Whether such routines code as these "minisubr" calls or are *completely* coded inline without error checking (where that is possible) is generally controlled by the optimize declaration. The detailed low-level specifics of this are discussed in section 24.2, page 246.

6.2 Proclamations: Global Declarations

Often one wants to make declarations "globally". For instance, to assert that a particular variable is always special, to set an optimization parameter for the compilation of entire file, etc. The function proclaim is used for this.

proclaim &rest *dcl-specs*
Each *dcl-spec* is a declaration specification (just like for declare—see the beginning of the previous section). However, it is put into force "globally". Many of the declarations one may make with proclaim are similar to those one might locally declare with declare, however their semantics are different because of their global nature. They are listed below.

Note: it seems the COMMON LISP proclaim only takes one *dcl-spec* argument?

It is important to note that proclaim subsumes previous usage of declare which was not in a special position (as described in the previous section). While declare in NIL is still accepted in "abnormal" positions (and might continue to be indefinitely for MACLISP compatibility), use of

proclaim is recommended to emphasize the nature of the declaration. Also, remember that proclaim evaluates its argument.

The NIL compiler will recognize calls to proclaim at top-level within a file, and, if the argument to it is constant (i.e., it is quoted), put that declaration in force for the remainder of the compilation, in addition to outputting it to the file; that it, it is as if it gets wrapped by (eval-when (eval compile load) ...) (see eval-when, page 25).

(special *var-1 var-2 ... var-n*)
A special proclamation not only globally declares that all of the named variables should be referenced special, but also that all bindings should be special. However, one normally uses defvar (page 24) without an initialization form to globally declare a variable special, or, when giving an initialization, uses defvar or defparameter.

(type *type var-1 var-2 ... var-n*)
Syntactically, this is the same as the regular type declaration. However, the type information is associated with the special values of the variables, not with lexically bound variables of the same names, so this is typically paired with a special proclamation, or with defvar (page 24), defparameter (page 24), or defconstant (page 24).

NIL, of course, ignores this right now.

6.3 Declaring the Types of Forms

Often one might want to associate type information with a form, where there is no variable being bound. The the special form allows one to do this.

the *type-specifier form*
the declares that the value returned by the evaluation *form* is of the type *type-specifier*. Were the compiler to do nifty things with type constraining, use of this could greatly enhance its ability to optimize; currently, the compiler generally throws away the type information, with the exception of a rare misguided special-form which explicitly looks for a the form. The NIL interpreter, however, does verify that the returned value is of the specified type.

The *type-specifier* may be any valid type specifier, and also may be of the form
(values *typespec-1 typespec-2 ... typespec-n*)
in which case the returned values must be of the specified types.

7. Sequences

A sequence is considered to be either a list or a vector (which is by definition a one-dimensional array). NIL supports a number of operations on sequences, which may be applied equivalently to lists, vectors, strings, and bit-vectors.

Many sequence function take *start* and *end* arguments to delimit some subpart of the sequence being operated on. As a general rule, the *start* is *inclusive*, and the *end* is *exclusive*; thus the length of the subsequence is the difference of the *end* and the *start*. The *start* typically defaults to 0, and the *end* to the length of the sequence. Also, the *end*, where it is an optional argument, may be explicitly specified as nil, and will still default to the length of the sequence. Thus,

```
(find item sequence)
```

searches the entire sequence,

```
(find item sequence :start 5)
```

and

```
(find item sequence :start 5 :end nil)
```

search from element number 5 onward,

```
(find item sequence :end 5)
```

checks the first five elements, and

```
(find item sequence :start 5 :end 10)
```

searches the subsequence (which has a length of 5) which consists of the elements with indices 5, 6, 7, 8, and 9.

Functions which operate on two sequences generally take the start and end for each of the sequences separately; these are keyworded arguments named :start1 and :end1 to specify the subsequence of the first sequence, and :start2 and :end2 to specify the subsequence of the second sequence. See, for instance, replace (page 52).

Many sequence functions (and many other functions) perform comparisons of various sorts. For instance, find searches for an item in a sequence. (find *item sequence*) looks for an element of *sequence* which is eql to *item*, and returns that element. The particular test used may be customized: a different test function may be specified by use of the :test keyworded argument, as in

```
(find item sequence :test #'equal)
```

which uses equal rather than eql. The sense of the test can be reversed by using :test-not instead of :test:

```
(find item sequence :test-not #'equal)
```

returns the first element of *sequence* which is *not* equal to *item*.

Often, what one wants to compare against is not each element of the sequence, but some subpart of each element. For this, rather than composing a new :test function, one may specify a :key function. For instance,

```
(find item sequence :key #'car)
```

will return the first element of *sequence* whose car is eql to *item*. This may also be used with :test or :test-not. Note that the key function is only applied to elements extracted from sequences, never to things like the *item* argument to find.

Another class of functions do "searching" by means of a unary predicate. These functions invariably come in pairs; one in which the test is satisfied if the test function returns a non-nil value, and one in which the test is satisfied if the test function returns nil: for instance, position-if and position-if-not, and assoc-if and assoc-if-not. Often these functions also take a :key keyworded argument also; if that is specified, then that function is applied to the datum being tested and the result given to the test function. For the sequence functions, the datum is the element of the sequence being examined.

Finally, for things which go grovelling through sequences sequentially, one may specify the direction by use of the :from-end keyworded argument, which if not nil means that the result will be as if the subsequence was processed from the higher to lower indices, rather than from lower to higher. Unless explicitly specified for a particular function, it may *not* be depended on that the sequence is actually processed in that order, only that the end result is the same; for this reason, it is generally a bad idea if the comparison or key functions have any side effects, or depend on the ordering of the elements on which they are called.

7.1 Accessing Sequences

elt *sequence index*

This is the general sequence access function. It returns the *index*th element of *sequence*; the index is taken to be zero-origined. This will work generally on lists, vectors, strings (which are by definition vectors anyway), etc. One may modify an element of a sequence by using setf. For instance,

```
(setq v (make-vector 10))
  => #(nil nil nil nil nil nil nil nil nil nil)
(setf (elt v 5) 'foo)
  => foo
```

And now,

```
v => #(nil nil nil nil nil foo nil nil nil nil)
```

It is an error for *index* to be negative, or not less than the length of *sequence* as defined by length. This means that it is an error to index pass the end of a list (in this elt differs from nth, page 57), and also in its treatment of vectors with fill pointers (the fill pointer defines the length—to access anywhere within such a vector, use aref (page 103)).

length *sequence*

Returns the length of *sequence*. If *sequence* is a vector with a fill pointer, the fill pointer is returned. In NIL, length will detect a circular list, and signal an error; in other implementations, it may fail to terminate—contrast the definition of list-length, page 59.

If *sequence* is a list, it is an error for it to not terminate in nil—in this, length differs from previous implementations of NIL.

7.2 Creating New Sequences

Many functions which create sequences take an argument which is the type of sequence to be created; it is typically called the *result-type*, or perhaps just the *type*. This may be, in general, a type specifier suitable for use with typep, but must of course be a subtype of sequence—that is, a subtype of list or vector. The type of sequence created will be the most specific type of sequence which is a subtype of that specified type. At this time, the NIL sequence code has not been integrated with the NIL array type code, so it is possible that complicated type specifications will not work. The following types will always work: list, string, simple-string, bit-vector, simple-bit-vector, simple-vector, and vector. vector, which is really an abbreviation for the type specifier (array * (*)), i.e., a one-dimensional array of unspecified element-type, will create a vector of element-type t, i.e., a general vector.

make-sequence *result-type size* &key *initial-element*

Makes a sequence of the given type and size. The types of most interest are list, string, vector, and bit-vector. If the :initial-element keyworded argument is given, then the sequence is initialized with that element. Otherwise, the initialization depends on the type of the sequence. For instance,

```
(make-sequence 'list 5 :initial-element t)
  => (t t t t t)
(make-sequence 'string 5 :initial-element #\*)
  => "*****"
```

concatenate *result-type* &rest *sequences*

Creates a sequence of type *result-type* (as might be given to make-sequence), and stores in it the concatenation of all the elements of *sequences*. For instance,

```
(concatenate 'string "foo" "bar" "baz")
  =>  "foobarbaz"
(concatenate 'list "foo" "bar" '(1 2))
  => (#\f #\o #\o #\b #\a #\r 1 2)
```

subseq *sequence start* &optional *end*

Returns a sequence of the same general type as *sequence*, containing elements from *start* up to (but not including) *end*.

```
(subseq "foo on you" 4)          =>  "on you"
(subseq "foo on you" 4 6)        =>  "on"
(subseq "foo on you" 4 3)        =>  is an error
(subseq '(a b c d) 1 3)          =>  (b c)
```

Note that the result of subseq never shares with the original sequence. Thus, (subseq *list* 5) is not the same as (nthcdr 5 *list*). In fact, subseq would signal an error in this case if the *list* did not have at least 5 elements.

The result of subseq will always be a simple sequence; if, for instance, *sequence* is an adjustable array of element-type string-char and has a fill pointer, the result will just be a simple string.

copy-seq *sequence*

Copies the sequence *sequence*. This might be necessary if the result is going to be modified, for instance. The result of **copy-seq** will always be a simple sequence, as described under **subseq**.

See also the **map** function, page 54, which produces a sequence of the results of applying a function to the corresponding elements of some input sequences.

7.3 Searching through Sequences

find *item sequence* &key *from-end start end test test-not key*

find searches through the specified subsequence of *sequence* until it finds an element which satisfies the specified comparison against *item*, in which case it returns that element; if no match is found, find returns **nil**. If a non-null *from-end* argument is specified, then the result (if there is a match) will be the "rightmost" element which matches *item*, otherwise it will be the "leftmost".

find-if *test sequence* &key *from-end start end key*
find-if-not *test sequence* &key *from-end start end key*

find-if returns an element of the specifed subsequence of *sequence* which satisfies the function *test*, or nil if no such element is found. If a *key* argument is specified, then *test* is called on the result of calling *key* on an element of the sequence; otherwise, *test* is called on the element directly. If a non-null *from-end* argument is specified, then the result will be the rightmost such element within the subsequence; otherwise, it will be the leftmost.

find-if-not is similar, but succeeds if the result of calling *test* is nil.

position *item sequence* &key *from-end start end test test-not key*

If there is an element of the specified subsequence of *sequence* which matches *item* according to the specified test, then **position** returns its index (the index within *sequence*, not within the subsequence); otherwise, **position** returns nil. If a non-null *from-end* argument is specified, then the result is the index of the leftmost element satisfying the test; otherwise, it is the index of the rightmost such element.

position-if *test sequence* &key *from-end start end key*
position-if-not *test sequence* &key *from-end start end key*

position-if returns the index of an element within the specified subsequence which satisfies the test function *test*. If *key* is specified, it is a function called on the element of the sequence, and that result is given to *test* instead of the element itself. If a non-null *from-end* argument is specified, then the index will be of the rightmost such element; otherwise, of the leftmost. If no such element is found in the subsequence, nil is returned. As with **position** and all similar functions, the index is the index within *sequence*, not within the subsequence.

position-if-not is similar, but succeeds if the result of calling *test* is nil.

count *item sequence* &key *from-end start end test test-not key*

count returns the count of elements within the specified subsequence of *sequence* which satisfy the specified comparison against *item*. The test function may depend on the order in which the elements are processed: if a non-null *from-end* argument is specified, then the elements will be processed from right to left (i.e., decreasing indices); otherwise, from left to right (increasing indices). As usual, the first argument to the test is *item*, and the second is the element of *sequence* if no *key* is specified, otherwise the result of calling *key* on the element.

count-if *test sequence* &key *from-end start end key*

count-if-not *test sequence* &key *from-end start end key*

count-if returns the count of the elements of subsequence which satisfy the unary predicate *test*. If no *key* is specified, *test* is called on the elements of the subsequence; otherwise, it is called on the result of applying *key* to each element. If a non-null *from-end* argument is specified, then the elements are processed from right to left (decreasing indices), otherwise from left to right.

count-if-not is similar, but reverses the sense of *test*.

7.4 Miscellaneous Operations on Sequences

reverse *sequence*

Returns a copy of *sequence*, with the elements in the opposite order.

nreverse *sequence*

Reverses *sequence*, destructively; it does not create a copy. Note that if *sequence* is a list, one should always use the return value of nreverse; that is, do something like

```
(setq l (nreverse l))
```

rather than just

```
(nreverse l)
```

This is in general true for all destructive list operations, such as sort and delq. The reason is that although the cons cells of the input list are reused, the pointer returned is not necessarily the same as the original "first" cons of the list.

fill *sequence element* &key *start end*

Replaces the elements of *sequence* with *element*, from *start* (default 0) up to *end* (default length of the sequence).

```
(setq a '(0 1 2 3 4 5 6))
(fill a nil :start 2 :end 4)
  => (0 1 nil nil 4 5 6)
```

And now,

```
a => (0 1 nil nil 4 5 6)
```

replace *sequence1 sequence2* &key *start1 end1 start2 end2*

Replaces the elements of the specified subsequence of *sequence1* by the elements of the specified subsequence of *sequence2*.

substitute-if *new test sequence* &key *from-end start end key count*
substitute-if-not *new test sequence* &key *from-end start end key count*
Similar, by extension.

nsubstitute *new old sequence* &key *from-end start end test test-not key count*
The destructive version of substitute; *sequence* is modified.

nsubstitute-if *new test sequence* &key *from-end start end key count*
nsubstitute-if-not *new test sequence* &key *from-end start end key count*
Similar.

7.5 Iteration over Sequences

The following functions iterate a user-specified function over one or more sequences in various ways.

map *result-type function* &rest *sequences*
This is the general sequence mapping function. Note that this is different from the MACLISP and LISP MACHINE LISP map function, which is renamed to mapl by COMMON LISP.

The result is a new sequence of type *result-type* (see section 7.2, page 50), containing the results of applying *function* to the elements of *sequences*. There must be at least one *sequence* specified; *function* gets as many arguments as there are sequences—first it gets called on all of the first (index 0) elements, then on all the second elements, etc. The iteration terminates when the end of any of the sequences is reached, so the result will have the same length as the shortest input sequence.

```
(map 'list #'cons "abc" '(a b c d e f))
  => ((#\a . a) (#\b . b) (#\c . c))
```

If *result-type* is list, and the input sequences are all lists, then this is effectively the same as mapcar (page 31).

some *predicate* &rest *sequences*
some applies the function *predicate* to the corresponding elements of *sequences* (of which there must be at least one), in order. If the result of some application is not nil, then some immediately terminates the iteration and returns that value; if all the applications returned nil, some returns nil.

The predicate may depend on being called on the elements of the sequences in order. Only as many elements as there are in the shortest sequence are processed.

The effect of this may also be obtained by use of the thereis clause in the loop macro (page 144).

```
(setq v (make-sequence 'vector 10))
  => #(nil nil nil nil nil nil nil nil nil nil)
(replace v '(1 2 3 4 5 6))
  => #(1 2 3 4 5 6 nil nil nil nil)
v => #(1 2 3 4 5 6 nil nil nil nil)
```

The number of elements transfered is the minimum of the lengths of the two subsequences, i.e.,

```
(min (- end1 start1) (- end2 start2))
```

remove *item sequence* &key *from-end start end test test-not key count*

remove returns a copy of *sequence* of the same general type, except that elements within the specified subsequence which match *item* according to the specified test are not copied. If the *count* argument is specified, then it should be a non-negative integer which limits the number of matching elements which get "ignored"; if it is not specified, *all* matching elements within the specified subsequence will be missing from the result. The *from-end* flag is only really meaningful if a *count* is specified; if a non-null *from-end* is specified, then the rightmost *count* elements of the specified subsequence will be missing from the result, otherwise the leftmost.

remove-if *test sequence* &key *from-end start end key count*
remove-if-not *test sequence* &key *from-end start end key count*

Like all similar extensions.

delete *item sequence* &key *from-end start end test test-not key count*

This is the destructive version of **remove**, q.v.; it will attempt to use *sequence* to construct its result. The result may or may not be **eq** to *sequence*, and *sequence* may or may not be actually modified to produce the result: NIL will attempt to do this the "best way" it can. For this reason, **delete** should *only* be used for value, never strictly for effect.

delete will succeed in side-effecting *sequence* to produce its result if *sequence* is (1) a list, (2) an adjustable vector of any type, or (3) a vector with a fill pointer. The result is still not guaranteed to be **eq** to *sequence*.

delete-if *test sequence* &key *from-end start end key count*
delete-if-not *test sequence* &key *from-end start end key count*

Similar, by extension.

substitute *new old sequence* &key *from-end start end test test-not key count*

substitute returns a new sequence in which the elements within the specified subsequence matching *old* according to the specified test have *new* substituted. While **substitute** does not modify *sequence*, the result may share with *sequence*; in particular, if *sequence* is a list, the result may share a tail.

If a *count* argument is specified, then this is a non-negative integer which limits the number of substitutions made. In this case, specifying a non-null *from-end* argument causes the rightmost *count* elements matching *old* to undergo substitution, otherwise the leftmost.

every *predicate* &rest *sequences*

Like some, but returns t if the result of applying *predicate* to the elements of *sequences* is never nil; if some application of *predicate* is nil, then every terminates immediately and returns nil.

The effect of this may also be obtained by use of the always clause in the loop macro (page 144).

notany *predicate* &rest *sequences*

Returns t if the result of applying the function *predicate* to the corresponding elements of *sequences* is always nil; if the result of that application is not nil, then notany immediately terminates the iteration and returns nil.

The effect of this may also be obtained by use of the never clause in the loop macro (page 144).

notevery *predicate* &rest *sequences*

If the result of some application of *predicate* to the corresponding elements of *sequences* is nil, then notevery terminates its iteration and returns t; otherwise, it returns nil.

7.6 Sorting Sequences

sort *sequence predicate* &key *key*

This is the COMMON LISP sort function; when it is used without a *key*, it is MACLISP compatible.

sequence is destructively sorted according to the predicate *predicate*, which receives two arguments and should return a non-null value only if its first argument is strictly less than its second argument. If *key* is specified, then it is a function of one element which is applied to the sequence element before being passed on to the predicate.

For MACLISP compatibility, if *sequence* is a list, then the sort is *stable*; equivalent pairs of items (those where the two keys are neither strictly less than each other) remain in their original order. When *sequence* is a vector, a quicksort algorithm is used.

sortcar *sequence predicate*

This is provided for MACLISP compatibility. It is just like

```
(sort sequence predicate :key #'car)
```

stable-sort *sequence predicate* &key *key*

Like sort, but *guarantees* that the sort will be stable (see sort). If *sequence* is a vector, then a bubble sort is used.

8. Lists

Note also consp (page 18), listp (page 18).

8.1 Creating, Accessing, and Modifying List Structure

car *cons*
cdr *cons*
c....r *cons*

NIL defines all compositions of car and cdr up to four levels deep; for instance, (cddar *cons*) is equivalent to (cdr (cdr (car *cons*))).

Actually, *cons* may be either a cons or nil; the car and cdr of nil are *always* nil.

Normally in NIL, car, cdr, and their compositions compile into special subroutine calls into the NIL kernel which do error checking, but are much faster than function calling. Directing the compiler to produce faster code by use of the speed quality in the optimize declaration will cause all of these accesses to be completely inline coded, without checking; see section 24.2, page 246. (In NIL, car and cdr each take only one instruction.)

All of these functions may be used with setf in order to update the particular component of *cons*.

rplaca *cons new-car*
rplacd *cons new-cdr*

rplaca modifies the car of *cons* to be *new-car*, and returns *cons*; rplaca modifies the cdr. See also setf (section 5.9, page 38) which can be used to update any of the above car/cdr references.

These functions, and the use of setf with car, cdr, and friends, are normally coded by the NIL compiler as special subroutine calls into the NIL kernel which do argument checking, but are faster than function calls. Use of the speed quality with the optimize declaration may override this; see section 24.2, page 246.

first *list*
second *list*
third *list*
fourth *list*
fifth *list*
sixth *list*
seventh *list*
eighth *list*
ninth *list*
tenth *list*

car, cadr, etc. These may all be used with setf.

Note that the names of these functions use one-origin indexing:

```
(third x)  <==>  (nth 2 x)
```

rest *list*
cdr. This may be used with **setf**.

nth *index list*
Returns the *index*th element of *list*, zero origined. Note also that this takes its arguments in a different order than **elt** (and other more specialized sequence accessors).

If *index* is not less than the length of *list*, **nth** returns **nil** by analogy to **car** and **cdr**. In this it also differs from **elt**.

nthcdr *ntimes list*
Returns *list* after *ntimes* **cdr**s have been taken on it.

Note also the ' reader-macro (section 20.1.1, page 216), which is convenient for creating list structure in template form, especially if large portions of it are constant, and **push** (page 38) and **pop** (page 38) which can be used to maintain a list in FIFO form in an arbitrary **setf**able place.

cons *x y*
Makes a **cons** whose **car** is *x*, and whose **cdr** is *y*.

ncons *x*
Equivalent to (**cons** *x* **nil**).

xcons *x y*
"Exchanged" cons. Equivalent to (**cons** *y x*). This function is not normally used; it is inherited from MACLISP, where its existence is mainly for the benefit of the compiler in rewriting calls to **list** into calls it could chain together the computations of better:

```
(list x y z)
  ==>  (xcons (xcons (ncons z) y) x)
```

list &rest *elements*
Returns a freshly created list of its arguments.

```
(list)                  =>  nil
(list 'x)               =>  (x)
(list 'x 'y)            =>  (x y)
```

list* *first-thing* &rest *other-things*
Sort of like **list**, but the last argument to **list*** is used as the **cdr** of the final cons, instead of **nil**. Alternatively, it may be thought of as many nested conses:

```
(list* 'a)              =>  a
(list* 'a 'b)           =>  (a . b)
(list* 'a 'b 'c)        =>  (a b . c)
(list* 'a 'b nil)       =>  (a b)
```

make-list *size-of-list* &key *initial-element*

Creates a list of nils *size-of-list* long, whose elements are *initial-element* (which defaults to nil).

append &rest *lists*

(append *x y*) returns a list which has first all of the elements of *x*, followed by all of the elements of *y*; for instance,

```
(append '(a b) '(x y))  =>  (a b x y)
```

The subpart of this list (in the example, the cddr) is the original last argument to append; append never copies its last argument.

```
(append x y z)
  ==> (append x (append y z))
```

When given one argument, append returns that argument; with no arguments, it returns nil.

If just copying a list is desired, it is stylistically better to use (copy-list *list*) (page 59) rather than (append *list* nil).

revappend *list1 list2*

This is like (append (reverse *list1*) *list2*), but is more efficient because it only has to make one pass over *list1*.

last *list*

Returns the last cons of *list* (*not* the last element!), unless *list* is nil, in which case it returns nil. In NIL, last deals properly with a non-null last cdr of *list*. The only non-cons it will accept as an argument however is nil.

```
(last '(1 2 3 4 5))   =>  (5)
(last '(a b . c))     =>  (b . c)
(last nil)            =>  nil
```

nconc &rest *lists*

Joins together all of the *lists* by destructively modifying them. Specifically, for each of the *lists* which is not nil, the final cons (as might be returned by last) is modified by rplacd to be the next list.

```
(setq l1 '(a b))
(setq l2 '(x y))
(nconc l1 l2)          =>  (a b x y)
```

and now,

```
l1                     =>  (a b x y)
```

One should be careful, however, about using nconc strictly for effect (i.e., not using the returned value), because if the first list is nil (the empty list) the desired side-effect will not occur.

nreconc *list1 list2*

This is like (nconc (nreverse *list1*) *list2*), but can be faster because it only has to make one pass over *list1*.

list-length *list*

list-length returns the length of the list *list*. If *list* is circular, then list-length returns nil—in this it differs from **length**, which in NIL will signal an error but in other COMMON LISP implementations may fail to terminate.

It is an error for *list* to not terminate with nil (assuming it is not circular). In this, list-length differs from its implementation in previous releases of NIL.

copy-list *list*

Copies the top-level conses of *list*. This may be used to replace the common idiom

```
(append list ())
```

and, additionally, handles a non-null last **cdr** of *list* gracefully.

copy-alist *a-list*

Like **copy-list**, and additionally, each top-level element of *a-list* which is a cons has that first cons copied also (*not* the entire top-level of the a-list entry).

Note that this function name contains "alist", a term which is spelled "a-list" everywhere else now.

This replaces the LISP MACHINE LISP function **copyalist**, which exists identically in NIL.

copy-tree *tree*

Returns a copy of *tree*. Recurses through both the *car* and the **cdr**, terminating at non-conses. That is, only conses are copied. This replaces the old MACLISP idiom

```
(subst nil nil tree)
```

which has been changed incompatibly by COMMON LISP (page 60).

This function is the COMMON LISP equivalent of the LISP MACHINE LISP **copytree** function, which is defined in NIL as a synonym for **copy-tree**.

butlast *list* &optional (*n 1*)

This returns a copy of *list*, but without the last *n* elements. If the length of the list is less than or equal to *n*, nil is returned.

```
(butlast '(1 2 3))     =>  (1 2)
(butlast '(1 2 3) 2)   =>  (1)
(butlast '(1))         =>  nil
(butlast nil)          =>  nil
```

nbutlast *list* &optional (*n 1*)

Destructive version of **butlast**; the last *n* elements of *list* are "spliced out", by **rplacd**. nbutlast should be used for value, however, because if the length of the list is less than or equal to *n*, nil is returned and no "splicing" is performed.

tailp *sublist list*

tailp returns t if *sublist* is a sublist of *list*, nil otherwise. What this means is that after taking some number of cdrs of *list* (possibly zero), one gets to a list **eq** to *list*.

ldiff *list sublist*

If (tailp *sublist list*) is true, ldiff ("list difference") returns a list of the elements of *list* up to but not including *sublist*. This is the same as

```
(nbutlast list (length sublist))
```

If (tailp *sublist list*) is false, ldiff simply returns a copy of *list*.

8.2 Substitution

These functions are extensions of the subst and sublis functions which are defined by MACLISP and LISP MACHINE LISP. Note that they by default use eql to test for equality; this is incompatible with MACLISP and an early release of NIL, in which subst used equal but sublis used eq (but only worked on symbols). A different test may be specified by use of the :test keyworded argument; this is a predicate of two arguments used to test "equality". If it is more convenient, the sense of the predicate may be reversed by use of the :test-not keyword.

Note also that these functions only descend through list structure ("trees"); they do not look inside of vectors, arrays, or other structures.

subst *new old tree* &key *test test-not key*

Returns a copy of *tree*, with *new* substituted when a of the tree matches *old* according to the test.

This is *incompatible* with MACLISP subst, in that the result is *not* guaranteed to always copy even if substitution is not performed. The MACLISP idiom (subst nil nil *tree*) is replaced by the copy-tree function (page 59). Be on the lookout for the use of this idiom in old code, for it can cause obscure bugs when uncopied structure is modified. The NIL compiler will warn about use of this idiom and turn the subst into a call to copy-tree if (1) there are no keyworded arguments supplied (2) *new* and *old* are constants determinable at compile-time and (3) their values are eql. The runtime function cannot detect this idiom.

nsubst *new old tree* &key *test test-not key*

Like subst, but does not copy: the new components are destructively stored in *tree*. However, this should be used for value, for if *tree* matches *old*, the result is *new* but no bashing of list structure is done.

sublis *a-list tree* &key *test test-not key*

Like subst, but performs substitution for several things at once. *a-list* is an association list of the objects to match, and their replacements. For example,

```
(sublis '((yes . no) (t . nil)) '(t generally means yes))
  => (nil generally means no)
```

sublis, remember, looks at cars and cdrs equivalently. If we attempt to invert the above example, we get

```
(sublis '((no . yes) (nil . t)) '(nil generally means no))
  => (t generally means yes . t)
```

nsublis *a-list tree* &key *test test-not key*
Like sublis, but destructive. See also nsubst.

8.3 Using Lists as Sets

One common use of lists is as sets of objects. NIL (and COMMON LISP) provide a complement of functions for doing this.

All of the functions take similar arguments. Normally, they use eql as their predicate, so that they work on numbers properly also. If this is not suitable for the purpose, then a predicate may be specified by giving it as the :test keyworded argument. For instance,

```
(union '((a b) (b) (c)) '((d) (e) (a b) (b)) :test #'equal)
  => ((a b) (b) (c) (d) (e))
```

The sense of the predicate can be reversed by using :test-not instead of :test.

Sometimes the elements of the set are datastructures of some sort, and one desires to only compare one part of the datastructure, but not write a predicate to compare things. If the :key keyworded argument is used, then that is a function which will be applied to each element as it is tested, and the results of that will be given to the equality predicate, rather than the elements themselves. For example,

```
(union '((a) (b) (c)) '((d) (e) (a) (b)) :key #'car)
  => ((a) (b) (c) (d) (e))
```

The :key-specified function is only applied to elements extracted from lists, never to single *item* arguments given to any of these functions (such as member, below). The ordering of the result may not be depended on; neither may the result if either of the inputs contains duplicate elements (as defined by the predicate), nor the particular choice of element (that is, the one from the first list or the one from the second list). Thus,

```
(union '((a) (b x) (c)) '((d) (e) (e) (b y)) :key #'car)
```

might return either of the sets

```
((a) (b x) (c) (d) (e))
```

or

```
((a) (b y) (c) (d) (e))
```

since they are equivalent according to the test criteria.

member *item list* &key *test test-not key*
If *item* is a member of *list* according to the specified test (which defaults to #'eql), then member returns the sublist whose car satisfied the test. Otherwise, nil is returned. The other functions in this section are implemented in terms of this. Note that if a function is specified with :key, it is only applied to items of *list*, not to *item*.

Note that the default predicate for member, #'eql, *is incompatible with* MACLISP member. This provides consistency with all other similar functions.

memq *item list*
This is member with a test of #'eq. memq is not defined by COMMON LISP, but is inherited from MACLISP. It is specially handled by the NIL compiler.

union *list1 list2* &key

The destructive

ses obtained

intersection

2 are used to construct the result.

set-difference *list1 list2* &key *test test-not key*
Returns the set difference of *list1* and *list2* (a list of the elements of *list1* which are not present in *list2*, according to the predicate).

nset-difference *list1 list2* &key *test test-not key*
Destructive set-difference: the result is constructed from the conses of *list1* and/or *list2*.

set-exclusive-or *list1 list2* &key *test test-not key*
Returns a list of the elements which occur in either *list1* or *list2*, but not both, according to the predicate.

nset-exclusive-or *list1 list2* &key *test test-not key*
Destructive set-exclusive-or: the result is constructed from the conses of *list1* and/or *list2*.

subsetp *list1 list2* &key *test test-not key*
Returns t if *list1* is a subset of (but not necessarily a proper subset of) *list2*, nil otherwise.

adjoin *item list* &key *test test-not key*
If *item* is a member of *list* according to the specified test, this just returns *list*; otherwise, it conses *item* onto the front of *list*, and returns that. adjoin could have been defined as

```
(defun adjoin (item list &rest keyworded-args
                         &key test test-not key)
  (if (apply #'member item list keyworded-args)
      list
      (cons item list)))
```

which also shows the utility of the &rest keyword combined with &key.

8.4 Association Lists

Association lists are an abstraction built from lists which are useful in many cases. An *association list* (or *a-list*, sometimes misspelled *alist* in this document), is a list, all elements of which are conses; the car of each cons is the *key*, and the cdr of each cons is the data associated with that key. Association lists differ slightly from just lists used as sequences utilizing a key of car (or cdr): if an entry in an association list is the atom nil, then it will be ignored totally. Functions which deal with more general sequences or lists, such as member, find, and position, would blindly apply the supplied key (e.g., car) to nil, and try matching this against the item being searched for.

There are two common ways in which association lists are used. For one, what is being constructed is basically just a table with entries of a single key. In this, the entries in the a-list are conses of the key and the data associated with the key, and one uses assoc with an appropriate test to do the lookup; the data of the key may be replaced by using rplacd on the result of the assoc, assuming assoc did not return nil. There are two potential disadvantages with this approach. First, the lookup is linear, so the lookup time grows linearly with the length of the table. How much this matters depends on the efficiency of the equality predicate in use; eq is extremely fast, and eql is fairly fast, so this could matter a fair amount if they are the predicates in use. Second, because a-lists are constructed out of conses, they can be scattered all over virtual memory and cause poor paging performance; it is possible for each cons in an a-list to lie on a different page. If entries are only rarely added to or removed from the table (entire entries, not just update of the car or cdr of an a-list entry), it might be reasonable to ensure that the a-list lies in fairly contiguous virtual memory by copying it with copy-alist (page 59) when something is added to it. Some rough guesses, with a test of eq: if the a-list has over 50 entries, it is probably better to use a hash table. For 70 or 80, a hash table is virtually guaranteed to be better, unless most of the lookups will succeed, and find entries near to the front. For between 5 and 10 entries, an a-list is almost undoubtedly best, and can be optimized by copying with copy-alist. For values between 10 and 50, the decision on which to use requires balancing additions and deletions against wasting time (and creating garbage) using copy-alist. Probably it is best to only do the copying with copy-alist for tables which are not updated in the "normal course of program running", i.e., which might only be updated when new files are loaded.

The other common use of a-lists is where the implicit ordering of a list comes into play. One entry in the association list might have the same key as another which occurs "later" in the a-list; since association lists are always searched in left-to-right order, the first occurence will *shadow* any other occurences. A simple LISP lexical interpreter might use an association list to hold the lexical variable bindings, for instance. "Binding" pushes the new variable/value pair on the front of the "environment" a-list, and "unbinding" pops the entry off. For this sort of use, hash tables do not present an alternative without some hairy hacking of "contexts".

A third possible use of an a-list is for two-way associations; that is, the car and cdr may both be viewed as keys used to look up the other; one uses assoc to find the entry by the car key, and rassoc to find the entry by the cdr key. Hash tables do not provide this sort of service; if a-list lookup time is prohibitive in a particular case, the alternative is to use *two* hash tables.

assoc *item a-list* &key *test test-not*

Searches *a-list* for an entry whose car matches *item* according to the specified test. If one is found, that cons is returned; otherwise nil is returned. If a non-null result is returned, the datum of it may be modified by use of rplacd.

Note also that since the default test is #'eql, this is incompatible with MACLISP assoc. To retain the same effect, one must use

```
(assoc item a-list :test #'equal)
```

or simply assq (below). Of course, often the choice of assoc in MACLISP is because *item* is a number, so equal is not needed in NIL because eql compares numbers correctly.

rassoc *item a-list* &key *test test-not key*

Like assoc, except that *item* is matched against each datum in *a-list*, rather than each key.

assq *item a-list*

This is inherited from MACLISP and also is defined by LISP MACHINE LISP; it is *not* defined by COMMON LISP. It is identical to

```
(assoc item a-list :test #'eq)
```

Compilation of assq does *not* produce any different code than the above form, so the choice between the two is between COMMON LISP compatibility and verbosity in the source code.

rassq *item a-list*

This is inherited from LISP MACHINE LISP; it is not defined by COMMON LISP. It is the same as

```
(rassoc item a-list :test #'eq)
```

See assq (above) for more qualifications.

acons *key datum a-list*

This is the same as

```
(cons (cons key datum) a-list)
```

but shows the intent better.

pairlis *keys data* &optional *a-list*

Returns an a-list made by associating *keys* and *data* and adding them to the front of *a-list* (which defaults to nil). *keys* and *data* must be lists of the same length.

```
(pairlis '(foo bar) '("Foo" "Bar") '((baz . "Baz")))
  => ((foo . "Foo") (bar . "Bar") (baz . "Baz"))
```

The result will share structure with *a-list* (that is, (tailp *a-list pairlis-result*) is t), so modifications to the associations in the returned result will affect those assocations in *a-list* also.

9. Symbols

9.1 The Property List

See also section 5.10, page 39, which deals with property lists in a more general way.

symbol-plist *symbol*

Returns the property list of *symbol*. Unlike the MACLISP **plist** function, this only works on symbols, not disembodied property lists.

symbol-plist may be used with **setf** to change the property list of a symbol. This is generally not recommended practice, however, because doing so might cause properties essential to the NIL system to be lost.

get *symbol indicator*

Standard MACLISP **get**. The value stored under *indicator* on the property list of *symbol* is returned; if there is no such value, **nil** is returned. As in MACLISP, *symbol* may also be a disembodied property list. Unlike in MACLISP, **get** does not arbitrarily check the type and then return **nil** if it is neither a symbol nor a list; in NIL one gets an error for an invalid type.

For both MACLISP compatibility and convenience, **get** is not going to disappear in the future.

putprop *symbol value indicator*

Standard MACLISP **putprop**. If *symbol* already has an *indicator* property, this replaces it with *value*, otherwise puts a new one. As in MACLISP, *symbol* may also be a disembodied property list.

COMMON LISP does not define **putprop**; rather, one uses **setf** with **getf** (page 39). putprop is not going to be flushed from NIL, however. Note also that **putprop** takes its arguments in a different order from the "standard **setf** order" in which the value being stored comes last.

remprop *symbol indicator*

Standard MACLISP **remprop**. If *symbol* has an *indicator* property, then that is removed by being spliced out of the property list, and the sublist of the property list whose **car** is the value (being removed) is returned. If there is no such property, **nil** is returned. For instance, if one does

```
(putprop 'kitty 'yu-shiang 'flavor)
```

then

```
(remprop 'kitty 'flavor)
  => (yu-shiang and maybe some sublist of the property list)
```

See also the **remf** macro, page 40, for removing properties from property lists stored in arbitrary places.

Note: COMMON LISP does not define the actual value returned by **remprop**, only that it will be null if the property was not found, non-null if it was. The value returned by NIL **remprop** is compatible with MACLISP, and will be retained for that compatibility.

9.2 The Print Name

symbol-name *symbol*

Returns the name of *symbol*, which is a string.

Note that a string which is a symbol name should never be modified; neither should the name of a symbol be changed.

In NIL, the name of a symbol will always be a simple string.

The name of a symbol has variously been called a *print name* and a *pname*, terms which are being phased out but still pervade the NIL implementation.

samepnamep *sym1 sym2*

Returns t if *sym1* and *sym2* have **equal** print names. Case *is* significant. *sym1* and *sym2* may be strings too. Either (or both) of the symbols may be specified as strings instead.

samepnamep is really just a MACLISP function. It is obsoleted by **string=** (page 113), because the string comparison functions in NIL will accept symbols and perform the comparison on their names.

9.3 Creating Symbols

make-symbol *pname*

Makes a new uninterned symbol whose print-name is the string *pname*. It will have no value or function bindings, no package, and an empty property list.

That the print-name (as returned by **symbol-print-name**) of the returned symbol will be **eq** to *pname* should not be depended upon.

copy-symbol *sym* &optional *copy-props*

If *copy-props* is nil, then this is the same as (**make-symbol** (**symbol-name** *sym*)); that is, it returns a virgin symbol with the same print-name as *sym*. If *copy-props* is not nil, then the value and function definition and property-list and package of *sym* will be copied to the new symbol.

Actually, the current dynamic bindings (value and function) of *sym* are copied to the new global dynamic bindings of the new symbol.

copy-symbol with a null *copy-props* argument is a reasonable way to generate a unique symbol which is somewhat mnemonic although not completely visually unique. The NIL compiler copies symbols like **if-false** to generate tags, for instance; no new print name is created, just the basic symbol structure, on which properties can be placed. If the symbol is to be used as a variable in macro expansions, however, it may be better to use

gentemp (below).

gensym &optional *x*

Standard inherited-from-MACLISP gensym. gensym creates new uninterned symbols. The print name of the symbol is constructed by prepending a single character prefix to the decimal representation of a counter which gets incremented every time gensym is called. The name has been around for so long that automatically generated names are commonly refered to as *gensyms*, and the act of doing so as *gensyming*.

(gensym) returns such a constructed symbol.

(gensym *name*) sets the prefix to be *name*, which must be a string or a symbol, and then makes a gensym. (In MACLISP, only the first character of *name* is used; in NIL, the entire name is.)

(gensym *integer*) sets the counter to *integer*, and then makes a gensym. *integer* must not be negative.

gentemp &optional *prefix package*

gensym creates a symbol in a manner similar to gensym, but interns it in *package* (which defaults to the current package). Additionally, gentemp guarantees that the symbol is unique by continuing to increment its internal counter until it succeeds in constructing a symbol which has *not* already been interned in *package*.

Unlike gensym, the *prefix* argument to gentemp is not "sticky"; that is, it does not default to the last one supplied. If it is not supplied, it defaults to t. Also, there is no provision for resetting the internal counter.

Use this for creating variables for use in macro expansions, because the symbol can be typed in. Also, gentemp will leave some information around so that code analyzers or the compiler can see that the variable is a generated variable so may be optimized away without loss of debugging information. (Normally such a test would be that the symbol is not interned, i.e., it was created by gensym or copy-symbol. This does not work for gentemp because gentemp interns the symbol so it can be typed in for debugging.) In NIL, gentemp-created symbols are flagged by having a non-null si:gentemp-marker property.

symbol-package *symbol*

This returns the "home package" of *symbol*, or nil if *symbol* does not have one (it is not interned).

9.4 The Value and Function Cells

See also chapter 3, page 11 for discussion about scope, extent, and binding, and chapter 3, page 11 for a description of the NIL internal mechanism for performing variable and function binding.

Special implementation qualification: because of the hairy value cell mechanism in NIL, value cells are not just allocated in the heap, so (due to lack of code to do some relocation right now) there is an assembly-time limitation on how many may be created. Thus, generating symbols and using the value cells to store things may not work as well as you expected (an error complaining `NEW_SLINK wants to grow the SLINK` occurs). This limitation is not a function of the mechanism but rather of the lack of garbage-collector, however.

symbol-value *symbol*
Returns the current dynamic (special) value of *symbol*.

symbol-value may be used with **setf**.

boundp *symbol*
Returns t if *symbol* has a defined dynamic (special) value, **nil** otherwise. Note that

```
(setq *foo* 1)    =>    1
(let ((*foo* 3))
   (declare (special *foo*))
   (makunbound '*foo*)
   (boundp '*foo*))
 => nil
*foo*    =>    1
```

makunbound *symbol*
"Undefines" the current dynamic value of *symbol*.

An error is signalled if *symbol* is a constant (as defined by **defconstant**).

symbol-function *symbol*
Returns the current dynamic (special) function value of *symbol*. The result of **symbol-function** on a symbol defined as a macro or special form is undefined in COMMON LISP. In NIL, special form definitions are not stored here, but other kinds of function definitions are.

symbol-function may be used with **setf**.

fboundp *symbol*
Returns t if *symbol* has a defined dynamic function value, **nil** otherwise. (Like **boundp**.)

fmakunbound *symbol*
Analogous to **makunbound**.

9.5 Additional Names

Some other MACLISP-compatible, LISP MACHINE LISP-compatible, and older NIL-compatible names.

symeval *symbol*
Same as symbol-value, page 68.

set *symbol value*
Same as (setf (symbol-value *symbol*) *value*).

fsymeval *symbol*
Same as symbol-function, page 68.

fset *symbol function*
Same as (setf (symbol-function *symbol*) *function*).

get-pname *symbol*
Same as symbol-name, page 66.

plist *symbol*
Same as symbol-plist, page 65. Note that this is *not* exactly the same as the MACLISP plist function (maybe it should be made to be?). The MACLISP plist function "works" on disembodied property lists (specifically, it takes the cdr of a list) — this is sort of an artifact of the MACLISP implementation.

setplist *symbol new-plist*
The same as (setf (symbol-plist *symbol*) *new-plist*). The same MACLISP-compatibility qualifications hold as for plist.

getl *symbol indicator-list*
Standard MACLISP getl. Returns the subpart of the property list of *symbol* beginning with the first indicator found in the list *indicator-list*, or nil if none was found. As in MACLISP, *symbol* may also be a disembodied property list. Essentially, this is the third value returned by

```
(get-properties
  (if (symbolp symbol) (symbol-plist symbol) (cdr symbol))
  indicator-list)
```

copysymbol *symbol* &optional *copy-props*
MACLISP name for copy-symbol (page 66). Note that in MACLISP, however, *copy-props* is a required argument.

9.6 Symbol Concatentation

The following routines are not defined by COMMON LISP, but are fairly useful in their own right by macros etc.

symbolconc &rest *frobs*

symbolconc returns a new symbol (interned in the current package) whose name is the concatenation of the names of *frobs*. Typically, each *frob* is a symbol, however symbolconc also allows it to be a fixnum (in which case its decimal printed representation is used), a string, or a character (which must be convertible to a string, i.e., satisfy string-char-p).

si:package-symbolconc *package-spec* &rest *frobs*

Similar to symbolconc, but uses the package specified by *package-spec* as the package in which the resultant symbol is interned. *package-spec* may be either a package or the name of a package.

9.7 Internal Routines

%symbol-cons *string*

Internal symbol conser. Creates a symbol with *string* as its name. (Note there is no mechanism provided for modifying the name of a symbol.) *string* must be a simple string.

The following routines are the primitives from which the earlier routines could be built. They are open-coded by the compiler, and work on all symbols including nil. They are intended for low-level code like that which might be found in intern. However, they are (for the most part) now being phased out in favor of just using the ordinary functions (e.g., symbol-name) and the optimize declaration (see section 24.2, page 246).

%symbol-name *symbol*

Returns the print name of the symbol *symbol*.

%symbol-package *symbol*

Returns the contents of the package cell of *symbol*. May be used with setf.

%symbol-plist *symbol*

Returns the property list of *symbol*. May be used with setf.

%symbol-link *symbol*

This returns the contents of the *link cell* of *symbol*. This is the thing used to implement the totally hairy NIL value cell scheme, which is not actually described in this manual.

10. Numbers

10.1 Types, Contagion, Coercion, and Confusion

10.1.1 The Types

NIL provides several different representations for numbers. It provides integers, of essentially unlimited precision, and floating-point. There is also the **ratio** data type, for representing non-integer rational numbers. The **complex** data type has been added, but may not yet be trustworthy for anything other than simple arithmetic operations, and has not yet been specialized to a point where any significant "crunch" might be performed on it.

NIL integers are currently of two kinds. There are *fixnums* and *bignums*. *Fixnums* have 30 bits of precision, including the sign, and are represented without consing (i.e., no memory consumption). Integers which require more than 30 bits to represent are implemented as *bignums*. *Bignums* are an extended data-type, and can grow to any size, limited only by whatever system parameter happens to be limiting the growth of your NIL, and your patience.

There are four primitive floating-point formats supported in NIL, as described in section 2.1, page 3. These are

short-float
: This is implemented as a single-float (VAX `f_float` format) with some of the fraction bits truncated, so that it can fit into a NIL pointer without any memory consumption. All operations on this type essentially convert it to a single-float to perform the operation, and then pack it back (truncating some of the fraction bits) when it is returned. 5 of the fraction bits get truncated, leaving 19, or about 5 decimal digits of precision. Because the exponent is the same as for **single-float**, it has close to the same range, however.

single-float
: This corresponds to the VAX `f_float` format; this provides 24 bits of precision (about 7 decimal digits), and a range of about 2.9e-39 to 1.7e+38.

double-float
: This utilizes the VAX `d_float` format, which has 56 bits of precision (about 16 decimal digits). The exponent format is identical to that of **single-float**, so the range is essentially the same.

long-float
: This utilizes the VAX `h_float` format. This has a whopping 113 bits of precision (33 decimal digits), and binary exponent range from -16383 to +16383. This is a range of about 8.4e-4933 to 5.9e+4931. Because the machine instructions which operate on this format of float are not supported by all VAX hardware, NIL makes use of a VMS-supplied condition handler to cause emulation of the missing instructions. As a compromise, it also tries to avoid such instructions when it can (for instance, just for data movement).

10.1.2 Contagion and Coercion

Most of the NIL arithmetic functions are *generic*. That is, they accept numbers of any type (subject to the semantics of the function, of course), and automatically "coerce" as necessary when there are mixed types. Coercion is mainly a function of the floating-point types. For instance, division of integers might yield a ratio:

```
(/ 2 3)  =>  2/3
```

However, rational numbers are always "normalized", and automatically "convert" back to integers:

```
(+ 1/3 2/3)  =>  1
```

When a floating-point number meets an integer or a floating-point number of a "shorter" type, the latter is converted automatically to the format of the former before the operation proceeds:

```
(* 2 2.0s0)  =>  4.0s0
(* 2 2.0d0 2.0s0)  =>  8.0d0
```

Such contagious conversion is never performed in the other direction; floating point numbers are never converted to integers just because the result might be integral, for example, or to a shorter format of float.

Complex numbers in NIL are restricted to having components which are either both rational, or both floats of the same format. A complex number with rational components will automatically be converted back to a rational (non-complex) number if the result of a computation gives a zero imaginary part. This never occurs if the components are floats:

```
(* #c(0 1) #c(0 1))  =>  -1
(* #c(0.0 1.0) #c(0.0 1.0))  =>  #c(-1.0 0.0)
```

10.1.3 Confusion

As was described in section 2.1, page 3, the format of a floating-point number may be selected by use of a particular character as the exponent specifier. Thus, 2.0s0 is short-float, 2.0f0 is single-float, 2.0d0 is double-float, and 2.0l0 is long-float. COMMON LISP specifies that the default format is single-float. This is the format which is used when a floating-point number is read in, and either no exponent character is used, or one of e is used (e.g., 2.59e10). This is also the format specified as being returned by exponential, transcendental, or irrational functions when given rational inputs. NIL has historically had a default (in fact, it used to be the *only*) format of double-float. Because single-float (and short and long) are all quite new to NIL, the default format is still double-float. That is, currently

```
(typep 2.59 'double-float)  =>  t
(typep (log 3/4) 'double-float)  =>  t
```

However, *this will be changed*. The NIL reader/printer combination has been twiddled somewhat so that, while the default format chosen by read is double-float, the printer will always print the exponent character which will force the format.

read-default-float-format *Variable*

This variable (which is defined by COMMON LISP) names the type of float which read should create when it encounters one which does not specify the format. It should have as its value short-float, single-float, double-float, or long-float. The COMMON LISP default for this is single-float.

As a special interim hack, NIL allows this variable to have nil as its value. This is taken to mean (to the reader) to produce double-float by default, but (to the printer) to not use an "unspecified" format on output. That is, with *read-default-float-format* being nil, the printer will never produce "1.0", but rather "1.0d0".

It is strongly recommended that explicit use of coercion (using the float function, page 78) be made wherever the type of the result might matter, when using functions which must convert into floating-point before proceeding. This includes things like log, sin, and sqrt. Note also that certain operations on complex numbers with rational components, such as abs, implicitly use these other functions; for instance,

```
(abs #c(3 4))  =>  5.0d0
```

10.2 Predicates on Numbers

Note also the type predicates numberp, bignump, integerp, fixnump, floatp, ratiop, rationalp, and complexp.

zerop *number*
Returns t if *number* is integer, floating-point, or complex zero, nil otherwise.

plusp *non-complex-number*
minusp *non-complex-number*
Return t if *non-complex-number* is of the appropriate sign, nil otherwise.

oddp *integer*
evenp *integer*
Return t if *integer* is odd (even), nil otherwise.

In NIL (but *not* COMMON LISP), oddp and evenp have been extended to gaussian integers. evenp is defined as divisibility by $1+i$, and oddp as being not evenp. This definition has the property that exactly half of the gaussian integers are odd and half are even. While there is a remote possibility that this definition will be changed, I consider it very unlikely.

10.3 Comparisons on Numbers

= *number* &rest *more-numbers*
/= *number* &rest *more-numbers*
< *number* &rest *more-numbers*
> *number* &rest *more-numbers*
<= *number* &rest *more-numbers*
>= *number* &rest *more-numbers*
These functions each take one or more arguments. If the sequence of arguments satisfies a certain condition:

=	all the same
/=	all different
<	monotonically increasing
>	monotonically decreasing
<=	monotonically nondecreasing
>=	monotonically nonincreasing

then the predicate is true, and otherwise is false.

Complex numbers are only acceptable as arguments to = and /=; the others require their arguments to be non-complex.

These functions also have fixnum-only and double-float-only versions.

greaterp *num1 num2* &rest *more-numbers*
lessp *num1 num2* &rest *more-numbers*
These are implemented as synonyms of < and >, and exist for MACLISP compatibility.

max *number* &rest *more-numbers*
min *number* &rest *more-numbers*
Generic max/min. Works on any non-complex numbers. Note also the existence of max& and min& (page 87), which only work on fixnums, and max$ and min$ (page 91), which only work on double-floats.

10.4 Arithmetic Operations

+ &rest *numbers*
plus &rest *numbers*
Returns the sum of all of the numbers, performing type coercion as appropriate. If there are no numbers, 0 is returned. The name plus is retained for MACLISP compatibility. Note also the fixnum-only +&, and double-float-only +$.

1+
add1 *number*
(plus *number* 1)
The name add1 is retained primarily for MACLISP compatibility.

- *number* &rest *numbers*
With one argument, - returns the negative of that argument. With more than one argument, it subtracts all of the others from the first. Type coercion is performed as necessary. Note also the fixnum-only -&, and double-float-only -$ functions.

difference *number* &rest *numbers*
When given more than one argument, difference subtracts from the first argument all the others, and returns the result. When given one argument, difference returns it. This is noteworthy because it is compatible with the MACLISP difference function, and it is *incompatible* with - above.

1- *number*
sub1 *number*
(- *number* 1)
The name sub1 is retained for MACLISP compatibility.

minus *number*
This is the same as (- *number*); the name is retained for MACLISP compatibility.

* &rest *numbers*
times &rest *numbers*
Returns the product of all of the numbers, coercing as appropriate. The identity for this operation is 1. The name times is retained for MACLISP compatibility.

/ *number* &rest *numbers*
This is the generic rationalizing division function. With one argument, / reciprocates the argument; with more than one, it divides the first by all the others, and returns the result. / will return a ratio if the mathematical quotient of two integers is not an exact integer; truncating integer division is performed by the MACLISP compatible quotient function, and various sorts of truncation and rounding (not limited to floating-point inputs) are provided by the floor, ceiling, truncate, and round functions (section 10.7, page 79).

quotient *number* &rest *more-numbers*
This is the MACLISP-compatible quotient function. When given more than one argument, quotient divides the first by all the others, and returns the result. With one argument, returns that argument.

If quotient is given integer arguments, it performs truncating division (see truncate, page 79). However, if any of the arguments are ratios, it will instead do rational division, like /.

conjugate *number*
Returns the complex conjugate of *number*, which is *number* itself if *number* is not complex.

gcd &rest *integers*
Returns the greatest common divisor of the integers. With no arguments, gcd returns 0.

In NIL, gcd works on all gaussian integers. The result will always lie within the first quadrant, or along the positive real axis.

lcm *integer* &rest *more-integers*
Returns the least common multiple of the integers.

Like gcd, lcm in NIL works on all gaussian integers.

10.5 Irrational and Transcendental Functions

Remember that the functions in this section which must perform their operations in floating-point, will convert rational numbers to **double-float** currently, but to **single-float** in the future.

pi *Constant*

The value of this constant is *pi*, in the longest floating-point format available; in NIL, this is a long-float. To use a shorter format of *pi*, for instance double-float, one should use something like one of the forms

```
(+ (float pi 0.0d0) n)
(+ (coerce pi 'double-float) n)
```

to avoid having the value of **pi** cause coercion to **long-float** in your computations. Uses of both **coerce** and **float** like the above will be optimized by the compiler into the appropriate value.

If other similar constants are added to NIL, they also will be in long-float format, and should be treated similarly.

10.5.1 Exponential and Logarithmic Functions

exp *number*

Returns *e* raised to the power *number*, where *e* is the base of the natural logarithms. If *number* is a float, the answer is computed and returned in the same format: if it is rational, it is first converted to **double-float**.

expt *base-number power-number*

Returns *base-number* raised to the power *power-number*. If *base-number* is a rational number and the *power-number* is an integer, the calculation will be exact and the result will be a rational number. Otherwise, the calculation devolves into floating-point, and will use a logarithmic computation if *power-number* is not an integer.

log *number* &optional *base*

Returns the logarithm of *number* in the base *base*, which defaults to *e*, the base of the natural logarithms. The rules of contagious coercion apply here; *number* and *base* are converted to the largest format of the two, unless both are rational, in which case they are converted to **double-float**.

It is an error if *number* is zero, unless *base* is zero, in which case the log is taken to be zero.

sqrt *number*

Returns the principal square root of *number*. If *number* is rational, it is converted to a double-float first; sqrt in NIL never produces a rational number even if *number* might have a rational square root, even though this is not disallowed by COMMON LISP.

If *number* is a negative non-complex number, a complex number is returned.

isqrt *integer*

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.

10.5.2 Trigonometric and Related Functions

abs *number*

Returns the absolute value of the argument, which may be any type of number.

When applied to the complex plane, **abs** returns the distance of the point from the origin, which is

`(sqrt (+ (expt` *realpart* `2) (expt` *imagpart* `2)))`.

Thus, for complex numbers, the result will always be floating-point because there is a **sqrt** hidden away in the computation. Applications which require only *comparing* distances from the origin may use the above form without the **sqrt** operation; this is done, for instance, by the internal code for **gcd** of complex integers.

signum *number*

By definition,

`(signum` *x*`) <=> (if (zerop` *x*`)` *x* `(/` *x* `(abs` *x*`)))`

signum of an rational number will return `-1`, `0`, or `1` according to whether the number is negative, zero, or positive. For a floating-point number, the result will be a floating-point number of the same format with one of the mentioned three values.

signum of a complex number is a complex number of the same phase but with unit magnitude. As a result, it will always be a complex number with floating-point components (see the discussion under **abs**, above).

sin *radians*
cos *radians*
tan *radians*

Standard trig functions. The number *radians* will be converted to a **double-float** if it is rational. These accept complex arguments.

asin *number*
acos *number*

asin returns the arcsine of the argument, and **acos** the arccosine. The result is in radians. *number* will be converted to a **double-float** first if it is rational; it may also be complex.

Note that **(asin -5)**, for instance, results in a complex answer.

atan *y* &optional *x*

Fairly standard arctangent.

With one argument (which may be complex), the arctangent is returned; for a non-complex argument, the result is between $-\pi/2$ and $\pi/2$ (exclusive). More generally, the following definition is used:

```
atan(z) = -i*log((1+i*z)*sqrt(1/(1+z^2)))
```

With two arguments, neither may be complex. The arctangent of y/x is returned, and the signs of y and x are used to derive quadrant information. Also, x may be zero as long as y is not zero. The result will be between $-\pi$ (exclusive) and π (inclusive).

sinh *number*
cosh *number*
tanh *number*
Hyperbolic sine, cosine, and tangent. *number* may be complex.

asinh *number*
acosh *number*
atanh *number*
Etc.

10.6 Numeric Type Conversions

float *number* &optional *other*
If *other* is not supplied, then if *number* is a float of any type, it is returned; otherwise, it is converted to a float of the default format, which is currently double-float, but will be single-float in the future.

If *other* is supplied, then it must be a floating-point number. *number* is converted to a float of the same type. Thus,

```
(float frob 0.0f0)
```

causes explicit conversion of *frob* to a single-float. Note that this will convert to a shorter format also.

The NIL compiler contrives to recognize the cases where float is given a constant *other* argument, and call special efficient routines for converting to the given type.

rational *number*
This converts *number* to a rational number. If it is already rational, it is returned. Otherwise, a rational number is computed corresponding to the *precise* representation of the number. For instance,

```
(rational (float 7/11 0.0s0))  =>  333637/524288
```

It is always the case that for a floating-point number *f*,

```
(= (float (rational f) f) f)  =>  t
```

rationalize *number*
If *number* is rational, it is returned. Otherwise, the floating-point representation is taken to be an *approximation* of a rational number, and that rational number is returned. For instance,

```
(rationalize (float 7/11 0.0s0))  =>  7/11
```

rationalize is not the most efficient coercion routine around...

complex *realpart* &optional (*imagpart 0*)

This constructs a complex number with real part of *realpart*, and imaginary part of *imagpart*. If *realpart* is rational and *imagpart* is either (integer) 0 or not supplied, then the result is just *realpart*, because complex numbers with rational components and a zero imaginary part automatically turn back into rational (non-complex) numbers. If either argument is floating-point, then both components will be converted to the longest floating point format of the two. For instance,

```
(complex 5 3)            =>  #c(5 3)
(complex 1/5 3.0d0)      =>  #c(0.2d0 3.0d0)
(complex 2.510 0.7s0)    =>  #c(2.510 0.710)
(complex 2)              =>  2
(complex 2.0)            =>  #c(2.0 0.0)
```

10.7 Integer Conversion and Specialized Division

truncate *number* &optional *divisor*
floor *number* &optional *divisor*
ceiling *number* &optional *divisor*
round *number* &optional *divisor*

These functions serve two similar purposes. They may be used to convert from a floating-point number to an integer in various useful ways, and they can also perform division/remainder operations similarly.

If only one argument is given, it is converted to an integer by the appropriate method: by truncation for **truncate**, by rounding towards negative infinity for **floor**, by rounding towards positive infinity for **ceiling**, or by rounding towards the nearest integer for **round**. (The NIL and COMMON LISP **round** will round an exact half by rounding towards the even integer; in this it may differ from rounding in other languages.) Two values are returned: the first is the result of the conversion, and the second is the remainder for the operation. If *number* is an integer, then the first value is that integer and the second is 0. If it is a ratio, then the second will also be a ratio, and if it is a float, then the second will be a float of the same format. For example,

```
(truncate 2.5)          => {2, 0.5}
(truncate 5/2)          => {2, 1/2}
(truncate -2.5)         => {-2, -0.5}
(truncate -5/2)         => {-2, -1/2}
(ceiling 2.5)           => {3, -0.5}
(ceiling 5/2)           => {3, -1/2}
(ceiling -2.5)          => {-2, -0.5}
(ceiling -5/2)          => {-2, -1/2}
(floor 2.5)             => {2, 0.5}
(floor 5/2)             => {2, 1/2}
(floor -2.5)            => {-3, 0.5}
(floor -5/2)            => {-3, 1/2}
(round 2.5)             => {2, 0.5}
(round -2.5)            => {-2, -0.5}
(round 2.6)             => {3, -0.4}
(round -2.6)            => {-3, 0.4}
(round 3.5)             => {4, -0.5}
(round -3.5)            => {-4, 0.5}
```

If the *divisor* argument is given, the first value is the result of integer conversion of the result of dividing *number* by *divisor*, after the appropriate type of rounding or truncation is performed. The second value is the remainder for that division after the rounding/truncation. That is, if the values are *q* and *r*, then

```
(= number (+ (* q divisor) r))  => t
```

For instance,

```
(truncate 5 2)          => {2, 1}
(truncate -5 2)         => {-2, -1}
(truncate 5 -2)         => {-2, 1}
(truncate -5 -2)        => {2, -1}
(floor 5 2)             => {2, 1}
(floor -5 2)            => {-3, 1}
(floor 5 -2)            => {-3, -1}
(floor -5 -2)           => {2, -1}
```

What this means is, if *truncate* is given two integer arguments, then the two values are the quotient and remainder, as obtained by those functions. If *floor* is given two integer arguments, then the second value is the standard mod (page 81).

rem *number divisor*

When rem is given integer arguments, it is identical to the MACLISP remainder function. More generally, however, it performs the truncate operation on its two arguments, and returns the second value from that as its value.

remainder *number divisor*

This is identical to rem. The name is retained for MACLISP compatibility: if *number* and *divisor* are both integers (which is all the MACLISP remainder function accepts), then the operation is the same.

One slight discrepency exists with the MACLISP **remainder** function: in MACLISP, remainder will accept a *divisor* argument of 0 and return *number*. This is incompatible with the definition of **rem** (and also with fixnum open-compilation of **remainder** in MACLISP!) because of the implicit division involved—in NIL, a divisor of 0 results in a division-by-zero error.

mod *number divisor*

When *number* and *divisor* are both integers, then **mod** is the standard "modulus" function. More generally, **mod** performs the **floor** operation on its two arguments, and returns the second value from that as its value.

Note that by virtue of its definition in terms of **floor** which is a kind of division, a *divisor* of zero results in a division-by-zero error.

ffloor *number* &optional *divisor*
ftruncate *number* &optional *divisor*
fceiling *number* &optional *divisor*
fround *number* &optional *divisor*

These functions behave just like **floor**, **truncate**, **ceiling**, and **round**, except that the first value returned is converted to a float if it is not. While this is fairly useless for rational arguments, it can result in some efficiency gain for arguments of type **float**. Note that if both arguments are rational, only the first value will be a double-float, not the second.

10.8 Logical Operations on Numbers

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

One common use of integers in this manner is as sets; each bit which is "on" (is 1) in the integer represents the presence of a particular item in the set. If the integer is negative, then it is an infinite set, because the sign is virtually extended to infinity. The presence of a particular item in a set can be tested for with **logbitp**; one refers to the item by its zero-origined bit index. Other set operations can be performed with the various boolean functions: **logand** performs intersection, **logior** performs union, **logxor** performs **set-exclusive-or**, and **logandc2** performs **set-difference**. A new set with an item represented by bit-index *index* added can be constructed by a form like

```
(dpb 1 (byte 1 index) integer)
```

logbitp *index integer*

logbitp is true if the bit in *integer* whose index is *index* (that is, its weight is (**expt** 2 *index*)) is a one-bit; otherwise, it is false.

logior &rest *integers*
logxor &rest *integers*
logand &rest *integers*
logeqv &rest *integers*

These return the bit-wise logical *inclusive or*, *exclusive or*, *and*, or *equivalence* (also known as *exclusive nor*) of their arguments. If no arguments are given, the results are 0

for logior and logxor, and -1 for logand and logeqv, which are the identities for those operations. Note also the fixnum-only versions of these, logior&, logxor&, logand&, and logeqv& (page 88).

lognand *integer1 integer2*
lognor *integer1 integer2*
logandc1 *integer1 integer2*
logandc2 *integer1 integer2*
logorc1 *integer1 integer2*
logorc2 *integer1 integer2*

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 number.

The "c1" and "c2" in some of the above names should be read as "having complemented argument 1 (*or* 2)"; for instance, logorc1 is the logical *or* of the logical complement of *integer1*, with *integer2*.

logandc1 and logandc2 are often used as bit-clearing functions. However, the ordering given to such names as bit-clear or logclr is often confusing (and historically, has been incompatible from one macro-package to another). logandc1 returns *integer2* with all bits which are on in *integer1*, cleared; logandc2 returns *integer1*, after clearing any bits which are set in *integer2*.

boole *op integer1 integer2*

The function boole takes an operation *op* and two integers, and returns an integer produced by performing the logical operation specified by *op* on the two integers.

There are sixteen variables (the names of which are listed below) which have the boolean functions as their values; the boolean functions are represented as fixnums from 0 to 15 (inclusive).

The NIL implementation of boole defines the boolean functions such that they map into the standard "truth table" used in MACLISP. That is, if the binary representation of *op* is *abcd*, then the truth table for the boolean operation is

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

For example, the boolean function 4 has binary representation 0100. This shows that the result will have a bit set only when the corresponding bit of *integer1* is 1 and *integer2* is 0. This is the logandc2 operation. New code, especially that intended to be transported between COMMON LISP implementations, should *never* rely on this—this coincidence is provided *only* for MACLISP compatibility.

Also for MACLISP compatibility, when NIL **boole** receives more than three arguments, it goes from left to right, thus:

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

In certain cases it may accept less than three arguments. Again, new code should not rely on this behaviour.

lognot *integer*

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

logtest *integer1 integer2*

logtest is a predicate which is true 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)))
```

except that it can be done more efficiently, because it does not have to actually compute the logand.

ash *integer count*

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*.

The actual implementation of **ash** works on, and will produce, bignums. There is also an **ash&** function which deals only with fixnums.

In practice, *count* is only allowed to be a fixnum...

logcount *integer*

If *integer* is non-negative, **logcount** returns the number of "1" bits on in its twos-complement representation. If it is negative, then the number of "0" bits in that representation is returned; this is then the same as the count of "1" bits in the logical complement of it.

integer-length *integer*

integer-length returns the zero-origined index of the sign bit of the field needed to represent *integer* in twos-complement notation. That is, any integer *i* may be represented in twos-complement notation in a field (1+ (integer-length *i*)) long. If *integer* is non-negative, then it may be represented in unsigned binary form in a field (integer-length *integer*) long. For instance,

```
(integer-length 5)  =>  3
```

because the binary representation of 5 is ...0101.

The following two functions are provided for MACLISP compatibility. They both do some semblance of manipulation of the binary representation of integers. However, the results of these functions are defined in terms of the absolute value of the integer they are examining—because in NIL integers are oriented towards manipulation of their twos-complement representation, these routines may not be particularly efficient on negative integers.

haulong *integer*

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

For example:

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

haulong is provided for MACLISP compatibility: integer-length should be used in preference.

haipart *integer count*

This function exists primarily for MACLISP compatibility. Its functionality is subsumed by the byte manipulation functions ldb and dpb, which are described in the following section.

haipart returns the high *count* bits of the binary representation of the absolute value of *integer*, or the low -*count* bits if *count* is negative.

10.9 Byte Manipulation Functions

There are various functions in NIL and COMMON LISP to deal with arbitrary-width contiguous fields of bits within integers. These functions are not restricted to operations on fixnums, but rather deal with the twos-complement representation of arbitrarily large integers.

Most of these functions use an object called a *byte specifier*. This object is used to refer to a *field* within an integer, which is determined by the *size* of the field, and the *position* of the field within an integer. Both of these must be non-negative integers.

byte *size position*

byte constructs a byte specifier. You may not depend on the format of the object returned, only that it will be acceptable for use as a byte specifier by the following functions; in particular, the type of the object returned by byte may not be distinct. In general, the restrictions on the magnitude of *size* and *position* are implementation dependent. In NIL, they may be any non-negative fixnums. Byte specifiers in which the size and position will both fit in an unsigned field 15 bits long (that is, they are both integers from 0 to 32767 inclusive) are represented more efficiently than others.

byte-size *bytespec*
byte-position *bytespec*

byte-size returns the size "component" of *bytespec*, and byte-position returns the position "component".

ldb *bytespec integer*

bytespec specifies a field of *integer* to be extracted. That field extracted from the twos-complement binary representation of *integer* is returned as a non-negative integer. For instance, the low three bits can be extracted using the byte specifier constructed by (byte 3 0):

```
(ldb (byte 3 0) 15)  =>  7    ;15 is ...0111111 in binary
(ldb (byte 3 0) 14)  =>  6    ;14 is ...0111110 in binary
(ldb (byte 3 0) -3)  =>  5    ;-4 is ...1111101 in binary
```

The third group of three bits can be extracted using the byte specifier (byte 3 6):

```
(ldb (byte 3 6) 15)  =>  0   ;15 is ...0000111111 in binary
(ldb (byte 3 6) 63)  =>  7   ;63 is ...0111111111 in binary
(ldb (byte 3 6) -3)  =>  7   ;-3 is ...1111111101 in binary
```

ldb may be used with setf (page 38), if the form *integer* is a valid *place* argument to setf. Rather than modifying the integer itself, doing setf on a ldb form will "nest" like

```
(setf (ldb bytespec integer) newbyte)
 ==>
(setf integer (dpb newbyte bytespec integer))
```

ldb-test *bytespec integer*

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

While a ldb-test can in general be done more efficiently than the second form, it probably is not done specially yet in NIL.

dpb *newbyte bytespec integer*

dpb is sort of the inverse of ldb. It returns an integer with the same binary representation as *integer*, except that the field referred to by *bytespec* is replaced by the twos-complement binary represention of *newbyte*.

```
(dpb 3 (byte 3 0) 0)  =>  3
(dpb 7 (byte 3 0) 0)  =>  7
(dpb 0 (byte 3 0) -1) =>  -8
```

mask-field *bytespec integer*

This does not seem to be in yet?

mask-field returns an integer whose binary representation is all zeros, except in the field referred to by *bytespec*, which has the same bits as that field in *integer*. That is,

```
(mask-field bytespec integer)
  <=>
(dpb (ldb bytespec integer) bytespec 0)
  <=>
(ash (ldb bytespec integer) (byte-position bytespec))
```

If the integer is being used as the representation of a set (section 10.8, page 81), then *mask-field* could be considered to be performing an intersection operation on the set represented by *integer*, and the set of all objects whose bit positions correspond to the positions within the field specified by *bytespec*.

deposit-field *newbyte bytespec integer*
This does not seem to be in yet?

This is the "inverse" of mask-field. deposit-byte returns an integer which is the same as *integer*, except that the field referred to by *bytespec* is contains instead the *bytespec* field of *newbyte*. That is, it is like

```
(dpb (ldb bytespec newbyte) bytespec integer)
```

10.10 Random Numbers

random &optional *modulus random-state*

If no *modulus* argument is given, then this is compatible with the MACLISP random function of no arguments: it returns a number randomly distributed over the range of all fixnums. (COMMON LISP does not define random of no arguments.) Otherwise, the answer returned by random is a number of the same type evenly distributed between zero (inclusive) and *modulus* (exclusive).

If *random-state* is supplied, it must be an object of type random-state; this is what holds the state of the random number generator. If it is not supplied, then the global random number state (the value of *random-state*) is used.

The random number returned is random over "all its bits"; *random-state* is used to compute a sequence of random bits which are used to construct the result. For a floating-point number, this is used as the significand (fraction) of a constructed number which is scaled to the appropriate range; for an integer, a sequence of bits is constructed approximately 10 bits longer than that needed for the result, and then a modulus operation performed.

random-state *Variable*

This holds the global random state used by default by random. One may, for instance, lambda-bind this variable to a new object of type random-state to save and restore the state of the random number generator.

make-random-state &optional *state*

This creates a new random-state object. If *state* is nil or not supplied, then a copy of the current value of *random-state* is returned. If *state* is t, then a new random-state is returned, seeded from the time. Otherwise, *state* should be a random-state; its state is copied.

When NIL is first loaded up, the random-state object in *random-state* is always in the same state. It may be seeded from the current time by doing

```
(setq *random-state* (make-random-state t))
```

if that is necessary for applications. There is, however, no officially defined way to get a "known" random-state from which the same sequence of pseudo-random numbers may be generated, other than copying one with make-random-state and saving it. If this is found to be necessary (for instance to reproducibly debug a program which uses the random number generator), the form

```
(si:make-random-state-internal)
```
will create a new random-state the same as the one NIL starts up with.

For MACLISP compatibility, the **random** option to the **status** and **sstatus** macros is supported. (status random) returns a copy of the current value of ***random-state***; (sstatus random *random-state*) restores ***random-state*** to a copy of that. One may also reseed by doing (sstatus random *integer*).

10.11 Fixnum-Only Arithmetic

Currently, the NIL compiler does not make any use of type declarations to help it decide to inline-code arithmetic routines. Primarily for this reason, NIL provides a full complement of fixnum-only and double-float-only arithmetic routines, which will be inline-coded by the compiler (when possible and reasonable) into fairly efficient code.

In NIL, as in MACLISP, the totally open-compiled fixnum-only routines behave "as the machine does"; that is, overflow is generally not detected. Note that the VAX hardware detects division by zero, however, and those routines not compiled as machine instructions, such as **^&**, may detect overflow and signal an error.

10.11.1 Comparisons

=& *number* &rest *more-numbers*
/=& *number* &rest *more-numbers*
<& *number* &rest *more-numbers*
>& *number* &rest *more-numbers*
<=& *number* &rest *more-numbers*
>=& *number* &rest *more-numbers*
Fixnum-only versions of the **=**, **/=**, **<**, etc. functions.

max& *fixnum* &rest *more-fixnums*
min& *fixnum* &rest *more-fixnums*
Fixnum-only **max** and **min**.

10.11.2 Arithmetic Operations

+& &rest *fixnums*
Fixnum-only **+** (**plus**).

-& *fixnum* &rest *more-fixnums*
One arg: unary negation. Otherwise, fixnum-only subtraction.

***&** &rest *fixnums*
Fixnum-only multiplication.

/& *fixnum* &rest *more-fixnums*
Fixnum-only division. With one argument, reciprocates, which seems singularly useless to me; since this is truncating division, reciprocation is an error if the argument is zero, one if it is one, otherwise zero.

\& *fixnum1 fixnum2*
Fixnum-only remainder. Although there is no COMMON LISP \ function to make the fixnum-only (as inherited from MACLISP) \ function change incompatibly, the name of \ is being changed to \& for consistency. Note that this function must normally be typed in as \\&, because \ is the "quoting" character in NIL.

1+& *fixnum*
1-& *fixnum*
Fixnum-only **1+** and **1-**.

abs& *fixnum*
Fixnum-only **abs**.

signum& *fixnum*
Fixnum-only **signum**.

^& *fixnum1 fixnum2*
Fixnum-only **expt**. It is an error for the result to exceed the range representable by a fixnum.

10.11.3 Bits and Bytes

logand& &rest *fixnums*
logior& &rest *fixnums*
logxor& &rest *fixnum*
logeqv& &rest *fixnums*
lognand& *fixnum1 fixnum2*
lognor& *fixnum1 fixnum2*
logandc1& *fixnum1 fixnum2*
logandc2& *fixnum1 fixnum2*
logorc1& *fixnum1 fixnum2*
logorc2& *fixnum1 fixnum2*
Fixnum-only boolean functions

boole& *op fixnum1 fixnum2*
Fixnum-only **boole**.

lognot& *fixnum*
Fixnum-only lognot.

logtest& *fixnum1 fixnum2*
Fixnum-only logtest.

logbitp& *index fixnum*
Fixnum-only logbitp. This is defined for an *index* larger than the number of bits in a fixnum; however, *index* must be a fixnum.

ash& *fixnum count*
Fixnum-only ash. Shifting by a positive *count* may shift bits into the sign position, thus changing the sign of the result (and losing bits). It is an error if *count* is not of type (signed-byte 8), that is, between -128 and 127 inclusive.

logcount& *fixnum*
Not yet in?

haulong& *fixnum*
Not inline-coded, but provided for completeness (see, perhaps, %fixnum-haulong, <not-yet-written>). Maybe this should be diked, since haulong should dispatch just as rapidly.

ldb& *bytespec fixnum*
Fixnum-only ldb. This is not strictly a version of generic ldb which takes a fixnum second argument, but rather a version of low-level fixnum byte-extraction which takes a general bytespec as an argument: it is an error for the byte to extend outside of the fixnum (for the position plus the size of the bytespec to be greater than 30).

dpb& *newbyte bytespec fixnum*
As ldb& is to ldb, so dpb& is to dpb. Other ldb& restrictions apply.

set-ldb& *bytespec fixnum newbyte*
Maybe this shouldn't be here, but it is in case setf gets used on ldb&.

Back in the olden days when there were few thoughts about integers greater than 34359738367 (or something like that), the byte-specifier to ldb and its friends used to be designated as *ppss*. The interpretation of this is that, if you consider *ppss* to be a 4-digit octal number, the number (octal) *pp* tells the *position* of the byte being referenced, and the *ss* the size. In order that a byte specifier not be so restricted in the size of the "byte" it is referencing (since the *pp* can be upwards-compatibly extended to the left but the *ss* cannot), NIL has incompatibly abandoned that format. So that code which uses this old format may be trivially converted, however, the old functionality may be obtained with the %ldb and %dpb functions, below.

%ldb *ppss fixnum*
Extracts the byte defined by *ppss* (as described above) from *fixnum*. It is an error if the byte so referenced lies outside of the fixnum (that is, the size plus the position is greater than 30).

%dpb *val ppss fixnum*
Returns a fixnum which is *fixnum* with the byte defined by *ppss* replaced by the fixnum *val* (truncated as necessary). It is an error if the byte so referenced lies outside of the fixnum (that is, the size plus the position is greater than 30).

10.11.4 The Super-Primitives

Getting closer still to the hardware...

load-byte *fixnum position size*
This is the primitive NIL extract-a-byte-from-a-fixnum function. In the style of many NIL primitives, and in the style of the VAX byte-extracting instructions, it takes arguments of position and size (different ordering from the byte function). It is an error for the byte described by *position* and *size* to lie out of bounds of the internal representation of a fixnum (30 bits).

deposit-byte *fixnum position size newbyte*
Modifies the byte, as per load-byte. Note argument ordering is different from dpb in that *newbyte* comes last. This is to make it convenient for setf to use.

sys:%fixnum-plus-with-overflow-trapping *x y overflow-code...*
sys:%fixnum-difference-with-overflow-trapping *Special form*
x y overflow-code...
sys:%fixnum-times-with-overflow-trapping *x y overflow-code...*
sys:%fixnum-ash-with-overflow-trapping *x y overflow-code...*
These are special forms which primarily exist for the benefit of implementing generic arithmetic functions. The appropriate binary operation on *x* and *y*, inline-coded, is performed; if afterwards there has been no overflow, that result is returned. Otherwise, *overflow-code* is run, and the resultant value returned.

Only the compiler knows how to use these right now.

10.12 Double-Float-Only Arithmetic

NIL provides some functions (like those in MACLISP) which operate only on double-floats. It is unlikely that corresponding functions will be provided for other floating-point types when they are added, however; inline-coded arithmetic on such numbers will be handled by declarations to the compiler eventually.

+$ &rest *double-floats*
***$** &rest *double-floats*
-$ *double-float* &rest *more-double-floats*
/$ *double-float* &rest *more-double-floats*
1+$ *double-float*
1-$ *double-float*
Double-float-only stuff. Essentially this is maclisp-compatible.

abs$ *double-float*
Double-float-only **abs**.

max$ &rest *double-floats*
min$ &rest *double-floats*

^$ *double-float fixnum*
The double-float only exponentiation function.

10.13 Decomposition of Floating Point Numbers

All of the following routines are defined by COMMON LISP. The basic premise is that a floating point number is represented by some number of digits in a base *b*, multiplied by *b* to some exponent, with a sign. In NIL on the VAX, *b* is of course 2 for the primitive floating-point data types, so the exponent is a binary exponent.

float-radix *float*
This returns the radix *b* of *float*. For the NIL primitive float data types (**short-float**, **single-float**, **double-float**, and **long-float**), it always returns **2**.

decode-float *float*
This function returns three values:
(1) The *significand* of *float*. This is a number between **1/***b* (inclusive) and **1** (exclusive), which represents the bits of *float*. The sole exception is for zero, for which the returned significand is zero (of the same floating-point type as *float*).
(2) The *exponent* of *float*. This is an integer which, if used to scale the significand (using **scale-float**, below), will produce a number with the same absolute value as *float*.
(3) The sign of *float*; this is either **1.0** or **-1.0**, in the same floating-point format as *float*.

Thus,

```
(multiple-value-bind (significand exponent sign)
                     (decode-float float)
  (= float (* (scale-float significand exponent) sign)))
 => t
```

scale-float *float integer*
This is like doing

(* *float* (**expt** *b integer*))

where *b* is the radix of *float*. It is done, of course, somewhat more efficiently and without danger of any sort of intermediate overflow or underflow if the final result can be represented. It is an error if the final result cannot be represented; **scale-float** will signal an error. (In a future NIL with a smarter compiler and type declarations, however, it may be the case that open-compilation of this function will not signal an error.)

Those familiar with the PDP10 MACLISP **fsc** function will recognize this as being somewhat of the same thing, although it is not subject to the kludges that **fsc** is (partly because the VAX does not represent unnormalized floating point numbers).

float-sign *float* &optional *other*

If *other* is not supplied, then this returns the sign of *float*, 1.0 or -1.0, in the same format as *float*. This is the same as the third value returned by decode-float.

If *other* is supplied, then the effect is of transfering the sign of *float* to *other*; the result will be of the same floating-point format and absolute value as *other*, but have the same sign as *float*.

float-digits *float*

This returns the number of base *b* digits in the representation of *float*. In NIL, this is a constant for any particular one of the primitive floating-point types. Specifically, short-float has 19, single-float has 24, double-float has 56, and long-float has 128.

float-precision *float*

This (he said while looking the other way) is the same as float-digits except that it returns 0 for zero.

integer-decode-float *float*

This returns three values, like decode-float. The difference is that the significand is returned as an integer, and the exponent differs accordingly. The magnitude of the integer is such that it is between 2^*p* (exclusive), and 2^(*p*-1) (inclusive), where *p* is the precision of *float* (as would be returned by float-precision). If the float radix is 2 as it is for the NIL primitive floating types, then the integer will have the same number of bits (integer-length page 83) as the precision. Again, an exception for zero exists; the first value will then be 0.

10.14 Implementation Constants

most-positive-fixnum *Constant*
most-negative-fixnum *Constant*

These have as their values the most positive and negative fixnums representable in NIL; integers of the same sign but greater magnitudes have to be represented as bignums. Because NIL uses twos-complement representation for fixnums, the absolute value of most-negative-fixnum is one greater than most-positive-fixnum.

most-positive-short-float *Constant*
most-positive-single-float *Constant*
most-positive-double-float *Constant*
most-positive-long-float *Constant*

These have as their values the most positive numbers of the corresponding formats which can be represented.

most-negative-short-float *Constant*
most-negative-single-float *Constant*
most-negative-double-float *Constant*
most-negative-long-float *Constant*

These have as their values the most negative numbers of the corresponding formats which can be represented.

least-positive-short-float *Constant*
least-positive-single-float *Constant*
least-positive-double-float *Constant*
least-positive-long-float *Constant*

These have as their values the smallest numbers of the corresponding formats that can be represented, which are still positive.

least-negative-short-float *Constant*
least-negative-single-float *Constant*
least-negative-double-float *Constant*
least-negative-long-float *Constant*

These have as their values the most positive negative numbers representable in the corresponding formats.

short-float-epsilon *Constant*
single-float-epsilon *Constant*
double-float-epsilon *Constant*
long-float-epsilon *Constant*

These have as their values the smallest positive numbers which, when added to 1.0 of the corresponding format, produce a different answer.

short-float-negative-epsilon *Constant*
single-float-negative-epsilon *Constant*
double-float-negative-epsilon *Constant*
long-float-negative-epsilon *Constant*

These are broken and wrong.

11. Characters

In NIL, *characters* are represented as a separate data type. This provides multiple benefits; among them, the object maintains some semantic identity when it appears in code (it is obvious that it is a "character"), and since it does maintain its identity as a character, the read/print/read "fixed-point" is capable of functioning across differing LISP implementations that internally utilize different character sets (e.g., ASCII vs. EBCDIC).

Characters in NIL have three different attributes: their *code*, their *bits*, and their *font*. The code defines the basic ("root") character. The bits are used as modifiers. Typically, an input processor (such as the editor, or even the prescan for the toplevel Lisp read-eval-print loop) will treat a character without any bits as "ordinary" and assume it is part of the text being typed in, but treat a character with some bits as being a command. Four of the special bits are named: they are **control**, **meta**, **super**, and **hyper**. The font of the character defines how certain things about the character are defined; for instance, equating characters of different fonts (**char-equal**), whether a character is upper or lower case, alphabetic, or a digit (**upper-case-p** etc.), and how to do case conversion (**char-upcase**). While NIL does not yet really define "fonts", there is a mechanism for how such definitions can interface to the low-level character manipulation primitives which utilize such things (section 11.7, page 100).

NIL character objects are immediate-pointer structures; they require no storage. Most of the routines which construct, dissect, and compare characters are open-coded by the compiler.

The NIL character set has not yet been cleaned up with respect to the confusion between the ASCII control characters and the characters it uses with the **control** bit. See section 11.6, page 99.

char-code-limit *Constant*
char-font-limit *Constant*
char-bits-limit *Constant*

These variables have as their values the upper exclusive limits on those attributes of characters. The values should not be changed. It happens that, in the VAX implementation, all three are 256 so that each quantity will fit into an 8-bit byte.

11.1 Predicates on Characters

standard-char-p *character*

This returns t if *character* is one of the "standard" ASCII characters. These are all the ordinary graphic characters (alphanumerics and punctuation characters), plus *Space* and *Return*. Only characters with 0 font and bits attributes can be standard.

graphic-char-p *character*

Returns t if *character* is a graphic (printing) character; that is, it has a single glyph representation. No character with non-zero bits attribute is graphic. Whether or not a character is graphic depends on its font.

alpha-char-p *character*
upper-case-p *character*
lower-case-p *character*
both-case-p *character*
alphanumericp *character*

Predicates on character objects. All are nil for characters with any *bits*; otherwise, they depend on the font.

char= *character* &rest *more-characters*
char< *character* &rest *more-characters*
char<= *character* &rest *more-characters*
char> *character* &rest *more-characters*
char>= *character* &rest *more-characters*
char/= *character* &rest *more-characters*

These routines collate or compare characters. The comparison *is* dependent on the bits, font, and case of the characters. Each routine returns t if all pairwise combinations of its arguments satisfy the appropriate predicate, nil otherwise; if a single argument is given, the result is t.

char= is the primitive function for telling if the characters are "the same character"—that is, if their code, bits, and font are all the same. It is what is used by eql and equal for comparing characters. char/= returns t if *none* of the pairs of characters are char=.

char<, char<=, char>=, and char> collate the characters according to some unspecified collating sequence. What is guaranteed is that the upper-case alphabetics, the lower-case alphabetics, and the digits will not be interspersed within the collating sequence, and, within each of these sets, they follow the obvious collating order.

char-equal *character* &rest *more-characters*
char-lessp *character* &rest *more-characters*
char-greaterp *character* &rest *more-characters*
char-not-lessp *character* &rest *more-characters*
char-not-greaterp *character* &rest *more-characters*

These routines compare characters independently of font, bits, and case. This does *not* mean that only the *code* portion of the characters is compared, but rather a font-dependent canonicalization is performed on the characters before they are compared. Thus, for two characters in different fonts, it is possible for them to not be char-equal even if they have the same *code* (one might be a "greek" character and the other "roman"), and conversely it is possible for them to be the same even with different codes (one might be upper-case and the other lower-case).

11.2 Character Construction and Selection

character *frobozz*

Coerces *frobozz* to a character. It may be a character, a fixnum, in which case char-int is applied, or a string, symbol, or character vector of length 1, in which case that single character is returned. This is what is used by the coerce function (page 9).

char-code *character*
char-bits *character*
char-font *character*

These three functions extract those attributes (as fixnums). All are efficiently open-coded by the compiler, and accept only character objects.

code-char *code* &optional (*bits 0*) (*font 0*)

Creates a character with code, bits, and font of *code*, *bits*, and *font*, unless that is not possible in the implementation, in which case nil is returned. In other words, it is an error for *code*, *bits*, or *font* to not be non-negative integers, however they need not be less than char-code-limit, char-bits-limit, and char-font-limit respectively.

make-char *char* &optional (*bits 0*) (*font 0*)

Creates a character with code of the the code of *char*, and with bits and font of *bits* and *font*, unless that is not possible in the implementation, in which case nil is returned. make-char could have been defined as

```
(defun make-char (char &optional (bits 0) (font 0))
  (code-char (char-code char) bits font))
```

11.3 Character Conversions

char-upcase *char*
char-downcase *char*

Upper- or lower-casify *char*, preserving the font and bits attributes of it. Note in particular that these functions just return *char* if it has a non-zero *bits* attribute, because such a character is not alphabetic and hence not subject to having its case changed.

char-int *char*

Returns a non-negative integer (in NIL, this will be a fixnum) encoding of the character *char*. If the bits and font of *char* are 0, this is the same as char-code. This is useful for hashing, and certain character-fixnum conversions such as those needed for the MACLISP tyi function are defined in terms of char-int.

int-char *integer*

If there is some character *c* for which (char-int *c*) equals *integer*, that character is returned; otherwise, nil is returned.

char-name *char*

Returns the name of the character *char*, if it has one. Supposedly, all characters which have zero font and bits attributes and which are non-graphic (see **graphic-char-p**) have names.

In NIL, the name of a character is by convention a symbol in the keyword package.

name-char *sym*

The argument *sym* must be a symbol. If the symbol is the name of a character object, that object is returned; otherwise **nil** is returned.

In NIL, character name symbols are symbols in the keyword package.

digit-char-p *digit* &optional (*radix 10*)

digit-char-p is a semi-predicate. If *digit*, which must be a character, is a digit in radix *radix*, then the weight of that digit is returned, otherwise **nil** is returned. By definition, a character with non-zero *bits* is not a digit, so for that **digit-char-p** will always return nil.

digit-char *weight* &optional (*radix 10*) (*font 0*)

If it is possible to construct a character with the given bits and font which has the weight *weight* in radix *radix*, then such a character is returned, otherwise nil is returned. *weight* must be a valid weight for the radix (a non-negative integer less than *radix*). Note the similarity to make-char (page 96) also.

```
(digit-char 5)       => #\5
(digit-char 10)      => nil
(digit-char 10 25)   =>  #\A
```

Note that unlike make-char, digit-char does not take a *bits* argument. This is because a character with a non-zero *bits* attribute is by definition not a digit.

11.4 Internal Error Checking Routines

The following may be of use to users writing their own routines for dealing with characters. (They should eventually be supplanted by more general type-checking macros, which will probably turn into calls to these routines...)

si:require-character *character*

Error-checks that *character* is in fact a character. This is what is called by (for example) the interpreted version of **char-code**.

si:require-character-fixnum *integer*

Error-checks that *integer* is in fact an integer for which there is a character representation; that is, on which **int-char** would return a character. All such integers in NIL happen to be non-negative fixnums.

11.5 Low-Level Interfaces

%int-char *fixnum*
Non-check version of **int-char**. It is an error for *fixnum* to not be the integer encoding of a valid character object, as would be returned by **char-int**.

The following four routines define digitness in the NIL character set, at a low level. These routines do *not* depend on the *font* of the character (if they take a character as an input); rather, they define the basic conversions for the "standard" NIL character set (NIL's interpretation of ASCII).

%valid-digit-radix-p *radix*
This defines the valid range of radices which %digit-char-in-radix operates on. The radix must be a fixnum.

%digit-char-in-radix-p *char radix*
Primitive predicate for testing whether the character object *char* is a digit character in the fixnum radix *radix* (which must be a valid numeric radix).

%digit-char-to-weight *char*
For any char which satisfies %digit-char-in-radix-p, this will return the weight of that digit.

%digit-weight-to-char *weight*
This inverts %digit-char-to-weight.

The following two routines perform low-level case mapping for the NIL character set.

%char-upcase-code *code*
%char-downcase-code *code*
These routines perform low-level case mapping for the NIL character set. *code* must be a valid character code; the returned value is a character with 0 bits and font attributes.

For example, if NIL did not provide low-level support for multiple fonts,

```
(char-equal c1 c2)
```

would be the same as

```
(char= (%char-upcase-code (char-code c1))
       (%char-upcase-code (char-code c2)))
```

The actual definition of char-equal is somewhat different in that the translation and canonicalization of the characters depends on their font.

11.6 The NIL Character Set

As can be seen, the format of character objects in NIL provides for a basic eight-bit *root* character (defined by the *code*), which can then have both *bits* and *font* attributes added to it. However, the I/O devices NIL must deal with only handle (at most) eight-bit characters—often only seven.

In principle, NIL will utilize an eight-bit character set; half of these will be *graphic* characters (the normal ASCII graphics, plus special symbols), and the other half reserved, format effectors (such as linefeed, backspace), and special commands for things like the editor and debugger (*Abort*, *Resume*, *Clear-Screen*, that kind of thing). All of these characters will then be able to have *bits* and *font* attributes added.

To deal with this on (for instance) an ordinary seven-bit ASCII terminal, NIL will have to do three things in the future. These are not done now, but are noted because they have bearing on the representation of character objects and input from devices.

1. Turn ordinary ASCII control characters into the NIL version of the control character. For instance, the ASCII character with code of 1, which is what you get when typing *Control-A* on a standard ASCII keyboard, will turn into the character *A* (uppercase A) with the *Control* bit set.
2. Provide "prefix" character-level commands in various places in order that other characters with bits may be entered (this is what is done in the editor). For instance, in the current input processor used by the reader, the character *Control-* is "prefix meta"—it "reads" another character and returns that with the meta bit added.
3. Provide some quoting or escape convention for inputting the extended graphic characters, since the codes for them are now normally being interpreted as characters with the *Control* bit set.

Secondarily, there will probably also be some translation of the actual codes involved, but that is irrelevant unless one is looking at the actual codes used in characters (which one generally should not). For instance, of the 256 "normal" characters, the low 128 would be graphics, and the high 128 the others. Some sort of symmetry would be maintained by having the ASCII format effectors and *Rubout* have their ASCII values plus 128. The others would be new characters to be used as various sorts of commands, but mostly left reserved for expansion. This is, in fact, approximately how the LISP MACHINE LISP character set is defined.

Unfortunately, none of this is done right now. When the character *Control-A* is typed on an ASCII terminal, it is read in as the character whose *code* is 1, not as what is actually the character *Control-A*. The editor, for instance, does some of the abovementioned transmutations on its input, and any prefixing commands for adding bits supplied by other input processors would be modeled after those used in the editor.

The algorithm which may be used to translate ASCII into what is *currently* used in NIL is this. Given an eight-bit character, if the *code* has the high bit set (it is greater than 127 decimal), then subtract out that bit, and remember to add the *Meta* bit to the character which will be eventually obtained. (This bit is what would be set by a terminal with a *Meta* key; such capability normally needs to be enabled by something like

```
$ set term/eightbit
```

to DCL.) Now we have a seven-bit character. If the seven-bit code is less than 32, and if it is *not* one of the codes for *Backspace*, *Tab*, *Linefeed*, or *Return* (8, 9, 10, and 13 respectively), then add 64 to it, and set the *Control* bit. Adding 64 forms the corresponding uppercase-alphabetic or punctuation character. Thus, 1 turns into *Control-A* (65 plus *Control*), and 135 turns into *Control-Meta-G* (135 = 128 + 7, = *Meta* + *Control* + 73 which is G).

11.7 Primitive Font Definitions

This section details how "fonts" are described at a low level in NIL in order to be interfaced to the character primitives described in this chapter. The contents of this section can be safely ignored by anyone who isn't going to try to define fonts and interface them to NIL.

First, remember that the *code* portion of a character is exactly 8 bits.

There are several "attributes" of a font which we need to encode in such a way that certain operations become trivial. They are these:

conversion to upper case
conversion to lower-case
: These want to be done rapidly

testing for certain attributes
: the predicates upper-case-p, graphic-char-p, both-case-p, etc.

comparison of characters of different fonts
: Two goals are desired here. It may be desirable for two characters in different fonts with the same code to *not* be char-equal, because they represent different things. It might also be desirable for two characters in different fonts with *different* codes to be char-equal—one might be *p*, the other P.

getting weights of digits
obtaining digit characters from weights

For efficiency, all of these goals are implemented through table lookup. There are several special variables whose values are simple vectors which contain one entry for each possible font (256 entries in each vector). In the initial NIL environment, all of the entries of each vector are initialized to the same thing.

Before these tables get listed, there are a couple of conventions which must be explained first. The first is what is called internally a *translation table* for characters. This is just a simple-string or simple-bit-vector 256 bytes (2048 bits) long, for doing code-to-code translation. For instance, translation to upper-case for a particular font utilizes such a table. The NIL primitives are defined such that it does not matter whether this datastructure is a simple-bit-vector or a simple-string. Another such datastructure is a *syntax table*. This is the same as a translation table, but each byte within the table is treated instead as a bit-mask with one bit set for each attribute about the code being defined. Both of these concepts will be abstracted away from characters in a future NIL.

The attributes about a font we define are these:

upper-case translation table
lower-case translation table

Each font has one of each of these. char-upcase works by returning a new character of the same font, with a *code* which is the result of extracting the byte from the upper-case translation table at the *code* index. (If the character has non-zero *bits* it is just returned however.)

syntax table
A syntax table (as described above) with syntax bits for the character primitives graphic-char-p, upper-case-p, lower-case-p, both-case-p, alphanumericp, and alpha-char-p. Constants for these bits are defined below.

digit weight table
This is a simple vector 256 long with one entry for each character code. All entries are fixnums. Those entries corresponding to characters which cannot ever be interpreted as digits should be -1; all others should have the weight of the digit.

digit char table
This is a simple-string (it must *not* be a simple-bit-vector) used for conversion from a digit weight to the corresponding character. The length of the string is the maximum radix which can be handled in that font, and all entries in-between must be valid. For instance, in the standard such table, the entry at index 9 is the character 9, and the entry at index 12 is the character C.

the typeface
The format of this object is not defined; the typefaces of different fonts are compared with eq by such predicates as char-equal. This is discussed later.

typeface canonicalization table
This is a simple vector 256 long, which is used to produce a canonicalization of a character within a typeface. The elements of the vector are fixnums. When two characters of the same typeface are compared, the code of each character is used as an index into this table (there is one per font, remember), and that result is used for the comparison. This allows a typeface to have, among several fonts, more than 256 characters.

si:note-primitive-font *font-number* &key *upper-case-translations syntax-table lower-case-translations typeface backwards-reference digit-weight-table digit-char-table typeface-canonicalization*
This assigns all of the information described above to the font *font-number*. Appropriate defaults are assigned for those entries which are nil.

The following constants define syntax bits used in the syntax table used. For each, the "$v_" constant has as its value the position of the bit within the field (byte), and the "$m_" constant a mask for testing the bit (with logtest&, for instance) or setting it (with logior&).

si:charsyntax$v_graphic *Constant*
si:charsyntax$m_graphic *Constant*
This bit tells if the character is graphic; that is, if it should satisfy graphic-char-p.

si:charsyntax$v_upper_case *Constant*
si:charsyntax$m_upper_case *Constant*
This bit tells if the character is upper case (should satisfy upper-case-p).

si:charsyntax$v_lower_case *Constant*
si:charsyntax$m_lower_case *Constant*
This tells if the character is lower case.

si:charsyntax$v_both_case *Constant*
si:charsyntax$m_both_case *Constant*
This tells if the character satisfies **both-case-p** (page 95).

si:charsyntax$v_digit *Constant*
si:charsyntax$m_digit *Constant*
This tells if the character is a (decimal) digit character.

si:charsyntax$v_standard *Constant*
si:charsyntax$m_standard *Constant*
This tells if the character, were it if font 0, is a standard char (**standard-char-p**, page 94).

si:charsyntax$v_alpha *Constant*
si:charsyntax$m_alpha *Constant*
This tells if the character is alphabetic. If the character is either upper or lower case, then it must have this bit set also. It is hypothetically possible, however, for a character to be alphabetic but neither upper or lower case.

Character comparison ala **char-equal**, **char-lessp**, etc., is performed as follows. We assume that it is possible to group characters into logical "typefaces"; canonicalizations within a typeface are valid, and comparisons of characters of different typefaces are not. Each font has associated with it one such typeface, and the assumption is that many different fonts map to just a few typefaces. When two characters are compared, if their typefaces are the same, then the codes of the characters are translated into fixnums by obtaining the *code*th elements of the typeface canonicalization tables, and the results compared in the appropriate manner (**=** for **char-equal**, **<** for **char-lessp**, etc.). If the typefaces are different, then the characters are not equal; for ordering, they are compared in a manner consistent with **char<**. NIL does not do anything at all with the typeface object described here; all that is done with typefaces is to compare them with **eq** to do this character comparison.

12. Arrays

Arrays in NIL encompass a large number of varied objects which share certain features and aspects of usage. NIL arrays may range in rank (number of dimensions) from 0 to about 250. All array indices in NIL are zero origined. One-dimensional arrays are *vectors*, are of the type **vector**, and may be used by the various sequence functions (in chapter 7). Restrictions on the types of the elements of an array (its *element type*) can result in special storage and access strategies. The data in multidimensional arrays is always stored in row-major order; this is compatible with MACLISP, although normally it does not matter.

12.1 Array Creation, Access, and Attributes

make-array *dimensions* &key *element-type displaced-to displaced-index-offset adjustable fill-pointer*

This is the general array creation function. *dimensions* may be an integer, in which case the rank of the created array will be one (it will be a vector), or a list of integers which are the sizes of the corresponding dimensions of the array.

The array will be created to hold objects of type *element-type*. If this is not supplied, **t** is assumed, and the created array will be able to hold any lisp objects. The most common types, aside from **t**, are **bit** (which creates a bit-array), and **string-char** (which creates a string-char-array). The special types which NIL supports, and their consequences, are discussed in section 12.2, page 104.

If *fill-pointer* is not null, then the array must be one-dimensional (a vector). It will be created with a fill pointer initialized to *fill-pointer*, which must be between zero and the size of the array (inclusive). Fill pointers are discussed in section 12.3, page 105.

Normally, the size of an array may not be changed (other than by modification of its fill pointer if it has one). This allows the implementation some leeway to provide for more efficient access and storage. However, if *adjustable* is specified and not **nil**, then the array will be created in such a way that its size (and its displacement attributes) can be modified later by **adjust-array**. Modification of array size and attributes is discussed in section 12.5, page 107.

The *displaced-to* and *displaced-index-offset* arguments control *array displacement*; that is where one array can "point into" another. This is discussed in section 12.4, page 107.

aref *array* &rest *indices*

Returns the element of *array* addressed by the *indices*. The number of indices must match the rank of the array, and each index must lie between zero (inclusive) and the size of the corresponding dimension of the array (exclusive).

If the array is one dimensional (i.e., a vector) and has a fill pointer, the fill pointer is *not* used to decide the range of the array which may be validly referenced; **aref** may be used to access *all* elements of such an array. In this, it differs from **elt** (page 49).

An array element may be set by using **setf** with **aref**.

array-rank *array*
Returns the rank of *array*.

array-dimension *array dimension-number*
Returns the size of the dimension *dimension-number* of *array*. The dimension number must be between zero (inclusive) and the rank of the array (exclusive).

array-dimensions *array*
Returns a list of the sizes of the dimensions of the array.

array-element-type *array*
Returns the element-type of *array*. This is not necessarily the same as what was given as the *element-type* argument to **make-array**; rather, it is the actual element-type used to *implement* the array, which will be a supertype of the originally specified element-type. This is discussed further below.

12.2 Array Element Types

Arrays may be restricted to contain only a certain type of element: this restriction is the *element type* of the array. Some element-types are distinguished in that the arrays will then be of a particular distinguishable type. For instance, arrays with element-type of **string-char** are **string-char-arrays**, and one-dimensional arrays of element-type **string-char** (which are therefore also vectors) are of type **string**. Similarly, the types **bit-array** and **bit-vector** are distinguished. There are other type restrictions (most of which result in special storage strategies for the data) which do not result in the array itself being of a particular type; nevertheless, the element-type of an array may be obtained with the **array-element-type** function (page 104).

When an array is created with a particular element type, the system chooses the most specific element type it offers which can satisfy the requirement. For instance, if an array is requested of element type **(double-float 0.0 1.0)** (double-floats between zero and one), a double-float array will be created. Similarly, for an array with element-type **symbol**, the element-type **t** will be used. **array-element-type** returns the type actually used; the requested element-type is forgotten.

There are several array types currently defined by NIL. Most of them are not particularly useful right now, because NIL does not yet have a smart enough compiler to cause declarations about array element types to do anything smart with the array accessing.

bit The array can only hold bits (the integers 0 and 1). A one-dimensional array of element-type **bit** is of the type **bit-vector**. If it not adjustable, not displaced, and has no fill pointer, then it will be a **simple-bit-vector**, and is specially implemented (less storage overhead, and faster access), and can be accessed efficiently with **sbit** (page 109). Many NIL complex datastructures, including the current implementation of bignums and more complex arrays which hold bits, small bytes, and binary data (such as double-floats), are built from simple bit vectors. Because of their utility, bit arrays are discussed further in section 12.8, page 109.

string-char
: The array can only hold characters which satisfy the predicate string-char-p (page 19). A one-dimensional array of this element-type is of type string. A string that is not adjustable, not displaced, and has no fill pointer is of the type simple-string; this is implemented more efficiently than a more general string, and can be efficiently accessed with the schar function (page 114). Chapter 13 is devoted to strings.

character
: The array can hold only characters (but they may be any type of characters). This provides no advantage over the element-type t in the current implementation; in a later NIL, vectors of element-type character will be acceptable to the string functions (chapter 13).

(unsigned-byte 8)
(signed-byte 8)
(unsigned-byte 16)
(signed-byte 16)
: Store those integers which are representable in the respective fields.

double-float
: The double-floats are stored packed in machine representation. Until the compiler has sufficient power to specially handle accesses to arrays of this type, there is no particular benefit to their use, because a generic array reference to a double-float array will have to cons the number to return it.

t
: An array of element-type t can hold any lisp object. If such an array is one-dimensional, not adjustable, not displaced, and has no fill pointer, then it is a simple vector (of the type simple-vector). Such a vector is especially efficient, and may be accessed with the svref function (page 109).

12.3 Fill Pointers

The :fill-pointer option to make-array allows one to create a vector of varying length. It is only applicable to one-dimensional arrays. The fill pointer of a vector is an integer which may range from 0 to the size of the vector. It is used as the length of the vector; the value of the fill pointer will be returned by length (page 49), and used as the length by all sequence and string functions; in fact, by everything except for aref (and its variants such as char and bit). The contents of the array at and beyond the fill pointer are still considered valid; they just are not considered when the array is viewed as a sequence.

A string with a fill pointer is reputed to be similar to a PL/I varying string, although such a comparison is beyond the realm of this author's knowledge.

To find the actual allocated length of a vector which has a fill pointer, use array-dimension with a dimension number of 0. array-dimension always returns the allocated length.

array-has-fill-pointer-p *array*

This returns t if *array* has a fill pointer (implying that it is a vector), nil otherwise. It is an error for *array* to not be an array.

fill-pointer *vector*

This returns the fill pointer of *vector*, which *must* be a vector with a fill pointer. This may be used with **setf**.

vector-push *vector object*

vector must be a vector with a fill pointer. If the fill pointer is a valid index into the vector (that is, its value is less than the allocated length of the vector), then **vector-push** stores *object* into that slot, increments the fill pointer, and returns the original (unincremented) fill pointer (which addresses where the object was stored). If the fill pointer is the same as the allocated length (the only other valid situation), then **vector-push** returns **nil**.

Note that the argument order to **vector-push** differs from that of the **push** macro (page 38). This will be incompatibly changed in the future, at which time **vector-push** will contrive to figure out which argument is which from their types.

vector-push-extend *vector object* &optional *extension*

This is like **vector-push**, but whereas **vector-push** will return **nil** if the vector is "full", **vector-push-extend** will instead call **adjust-array** to increase the size of *vector* in order that it might do the push. Thus, it never returns **nil**. If *extension* is supplied and not **nil**, then that is the amount by which the size of *vector* is incremented by **adjust-array**, if necessary; otherwise, some random guess based on the current size is used.

In order for **adjust-array** to succeed in increasing the size of *vector*, *vector* must have been created with the **:adjustable** option to **make-array**; see section 12.5, page 107.

The ordering of the arguments *vector* and *object* will be reversed in the future; see **vector-push**, above.

vector-pop *vector*

This is the inverse of *vector-push*. The fill pointer of *vector* is decremented, and the object addressed by that index is returned. The fill pointer must not already be zero.

reset-fill-pointer *vector* &optional (*index 0*)

Resets the fill pointer of *vector*, which must have one. If *index* is not specified, **0** is assumed.

This function is obsoleted by the use of **fill-pointer** with **setf**.

One common use of vectors with fill pointers is as buffers. For example, the NIL compiler uses a vector with a fill pointer for allocating a table of value cell indices to be referenced by the code it is compiling. It creates a vector with **make-array**, specifying that the array is adjustable, and giving it an initial fill pointer of 0. Then, it uses **vector-push-extend** to add a new entry, and the value that returns is the index into this table. **vector-push-extend** takes care of increasing the size of the vector if the initial guess as to its size was too small.

This same technique can be used for generating text, if the vector is a string (that is, make-array was given string-char as the :element-type keyword). This is how some string accumulating primitives work.

With vector-pop, vectors with fill pointers can also be used as stacks.

12.4 Displaced Arrays

Arrays may be created which do not have data of their own, but in fact "share" data with some other array. These are *displaced arrays*. The uses for displaced arrays vary. One might want to access the elements of a multi-dimensional array as if it were a vector; this could be done by

(make-array *size* :displaced-to *other-array*)

which returns a vector which will access the elements of *other-array* in row-major-order. With displacement, a *displaced index offset* may also be specified. Conceptually, when an array is accessed, a single index is computed from the indices and dimensions of the array; this is the index into the row-major-order data. This index then has the displaced-index-offset added to it, to get the index into the data for the array being displaced to.

Another potential use for displaced arrays is to reference some "substructure" of an array which implicitly has some "structure". This causes modification of the displaced array to modify the referenced subpart of the original array. In general, however, it is not appropriate to use array displacement as a substitute for a subseq operation (page 50): it is intended for cases where modification of one must implicitly modify the other, although if the subsequence is large it may be worthwhile.

Efficiency note: displaced arrays which are displaced to non-adjustable arrays access at just about the same speed as normal arrays (not counting those which are especially efficient, namely simple vectors, simple strings, and simple bit vectors). Arrays displaced to adjustable arrays are a touch slower. At this time, the NIL compiler does not know how to inline code any non-vector array access, however, so aref (or use of it with setf) will produce general function call unless there is exactly one index.

12.5 Modifying Array Sizes and Characteristics

Normally, an array may not have the size of its dimensions or other attributes changed once it is created (other than modification of its fill pointer; section 12.3, page 105). If a non-null :adjustable option is given to make-array, however, the array will be created such that this is possible.

adjust-array *array dimensions* &key *displaced-to displaced-index-offset element-type fill-pointer*

adjust-array interprets *dimensions* just as make-array does. The array is modified to have the new dimensions; however, its rank may *not* be changed.

If *fill-pointer* is specified, then *array* must have originally been created with a fill pointer; the value of *fill-pointer* is used as the new one.

The remaining options will not be detailed at this time. adjust-array currently only works on one-dimensional arrays, so although not generally useful yet, it has enough to keep vector-push-extend happy.

12.6 Special Vector Primitives

These functions are around in NIL for historical reasons. They should generally not be used in new code, for reasons which will become clear from reading the descriptions. They are documented because they are used all over the place in NIL itself still, and vref in particular is generated internal by the compiler from the use of aref.

vref *vector index*

This is *absolutely identical* to aref when aref is given a vector and a single index. aref is the preferred function to use in code; the NIL compiler compiles aref of a single index as a call to this internal vector-referencing subroutine.

Because this is the same as aref on a one-dimensional array, it is interesting to note that (for a vector with a fill pointer) vref does not use the vector-length to check if the index is in range.

vector-length *vector*

For historical reasons, this is the same as length but only accepts a vector as an argument. If the vector has a fill pointer, this is not the same as (array-dimension *vector* 0), which is the quantity used by vref for bounds checking.

12.7 Simple Vectors

Vectors of element-type t which are not adjustable and have no fill pointer are implemented as the primitive data type simple-vector. (This was called simple-general-vector in the previous NIL release. Do *not* misinterpret the name simple-vector to mean one encompassing vectors of other element types.) This type is extensively used in NIL as a building block for more complicated datastructures in NIL, including less simple arrays. There are special routines for creating and manipulating them, which are coded efficiently by the NIL compiler.

Vectors of this type may be checked for with typep, of course.

make-vector *size* &key *initial-contents initial-element*

Makes a vector of element-type t, *size* long. Because no complicated array options can be specified, this will always be a simple vector.

This may be called make-simple-vector in the future, but the name make-vector will be preserved indefinitely.

vector &rest *elements*
This makes a simple vector whose elements are initialized to *elements*. That is,

```
(vector 1 2 3)  =>  #(1 2 3)
```

svref *simple-vector index*
References a simple vector. This routine is entirely open-coded by the compiler, with no error checking; to retain runtime type and bounds checking, **aref** must be used.

svref may be used with **setf**.

This function used to be called **sgvref**; that name is still around.

simple-vector-length *vector*
Returns the length of the simple vector *vector*.

This used to be called **simple-general-vector-length** (yow!); that name is still around.

12.8 Bit Arrays

Arrays which contain only bits, which can be used to represent boolean *true* and *false*, are useful in various applications. There are several functions which perform boolean operations on arrays of this element type.

Bit arrays may be more or less appropriate for a particular application than integers used to represent a sequence of bits. Bit arrays (or bit-vectors) may be side-effected; integers may not. Integers however may be used to represent infinite sets, because they are virtually extended with their sign; see also section 10.8, page 81. Bit arrays, of course, may be multi-dimensional.

bit *bit-array* &rest *subscripts*
Just like **aref** (page 103), but only works on arrays of element-type **bit**.

setf may be used with **bit**.

sbit *bit-array* &rest *subscripts*
This is just like **bit**, but is only for use on *simple* bit arrays. This can result in much more efficient code, especially if the bit-array is one-dimensional.

setf may be used with **sbit**.

bit-and *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-ior *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-xor *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-eqv *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-nand *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-nor *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-andc1 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-andc2 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-orc1 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-orc2 *bit-array-1 bit-array-2* &optional *result-bit-array*

These crunch together *bit-array-1* and *bit-array-2*, performing the appropriate bitwise logical operation. *bit-array-1* and *bit-array-2* must have the same rank and dimensions. If *result-bit-array* is nil or not specified, then the result is a freshly created bit array of the same rank and dimensions. If it is t, then the results are stored in *bit-array-1*. Otherwise, it must be a bit array of the same rank and dimensions as the other two, and the results are stored into it.

bit-not *bit-array* &optional *result-bit-array*

Performs a bitwise logical negation on the contents of *bit-array*. If *result-bit-array* is nil or not specified, then the result is returned as a freshly created bit array of the same rank and dimensions as *bit-array*. If *result-bit-array* is t, then *bit-array* is side-effected with the results. Otherwise, *result-bit-array* should be the a bit array of the same rank and dimensions as *bit-array*, and will have the results stored into it.

12.8.1 Simple Bit Vectors

In NIL, a one-dimensional bit array which is not adjustable, not displaced, and has no fill pointer, is represented as the primitive type simple-bit-vector, which is represented more efficiently than a more general bit array. Simple bit vectors are used as building blocks for more complicated structures which contain binary data, such as the more complicated bit arrays, and even arrays of element-type double-float. There are primitives for accessing variable length fields from them as (possibly sign-extended) fixnums (but *not* general integers, yet), and primitives for treating them as if they were sequences of eight-bit bytes.

Once upon a time the name for this type was bits. This name still lingers in places, but is being replaced by simple-bit-vector.

make-bit-vector *size* &key *initial-element initial-contents*

Creates a simple-bit-vector *size* long. If neither *initial-element* nor *initial-contents* are specified, the contents of the created bit-vector are undefined. The current implementation of NIL storage allocation requires that all allocated datastructure be initialized, so in fact the bit-vector will always be initialized to zeros, but this could change in the future.

If *initial-element* is specified, it must be either 0 or 1, and the elements of the bit vector are initialized to that.

If *initial-contents* is specified, then it should be a bit vector, and the created bit vector is initialized to the contents of *initial-contents*. If *initial-contents* is longer than *size*, the extra elements are ignored; if it is shorter, then the contents of the remaining elements of the created bit vector are undefined (just as they are if neither :initial-element nor :initial-contents are specified). It is an error to specify both *initial-element* and *initial-contents*.

simple-bit-vector-length *simple-bit-vector*

Returns the length of *simple-bit-vector*.

nibble *simple-bit-vector skip take*
Returns the sequence of bits *take* long from *simple-bit-vector*, starting at *skip*, as a fixnum. *take* may range from zero to the number of bits representable in a fixnum (30); however, if the last, the result will include the sign bit so may be unacceptable for certain applications. The result is zero-extended.

setf may be used with nibble to replace the field.

nibble-2c *simple-bit-vector skip take*
Like nibble, but the result is sign-extended. That is, the result is interpreted as a signed binary value from the referenced field.

setf may be used with nibble-2c to replace the field.

get-a-byte *simple-bit-vector byte-index*
Interprets *simple-bit-vector* as a sequence of type (unsigned-byte 8), and returns the *byte-index*th byte.

setf may be used with get-a-byte.

get-a-byte-2c *simple-bit-vector byte-index*
Interprets *simple-bit-vector* as a sequence of type (signed-byte 8), and returns the *byte-index*th byte.

setf may be used with get-a-byte-2c.

13. Strings

Strings are vectors of characters which satisfy the predicate **string-char-p** (page 19). Although the generic sequence and array primitives operate on strings, there are two reasons for having additional functions for strings. For one, it is convenient for atomic symbols to be used in place of strings; symbols are not coerced to strings by sequence functions, but they *are* for most of the string functions. Additionally, many of the string functions which compare characters do so independent of the character case; the sequence functions are generally based on the **eql** predicate (page 20), so are case dependent.

Eventually, the string functions will be generalized to handle arguments which are general character sequences (that is, of type **(vector character)**, vectors which can hold any characters, not just those which are **string-char-p**). Until then, functions which can be given character arguments which contain non-zero bits and font attributes may not behave correctly if that is done.

13.1 String Coercion

string *frobozz*

This is a COMMON LISP function for doing coercion to a string. If *frobozz* is a string, it is returned; if it is a symbol, its name (**symbol-name**, page 66) is returned.

For backwards compatibility with earlier versions of NIL, and compatibility with LISP MACHINE LISP, **string** will also coerce characters and integers to strings. A character is converted to a string one character long, having that character as its sole element: the character must satisfy **string-char-p** (page 19) for this to succeed. An integer is converted to a string like a character, after first applying **int-char** (page 96), which of course must succeed in making a character for the string coercion to succeed.

to-string *frobozz*

This routine believes itself to be a coercer of *sequences* to strings. It is a superset of **string**; it additionally will accept any sequence, and interpret that as a sequence of characters; it thus may be used to convert lists or vectors of characters to strings; the contained characters will be coerced to characters using **to-character**. Bit vectors will be converted into strings of the characters "1" and "0".

This routine believes that **nil** is a sequence of no elements, so is not the appropriate routine for coercing symbols to strings; **string** (above) is.

For sequence to string coercion, one should probably be using **coerce** (page 9) anyway.

13.2 String Comparison

When routines deal with boundaries within strings, there are two different conventions applied. Many functions take range arguments as the lower inclusive bound and the upper exclusive bound (generally named *start* and *end*). These arguments conveniently default to 0 and the length of the string (typically). As a general rule, an upper exclusive bound may be explicitly specified as nil producing behavior as if it were not specified; this is often necessary in order for following optional arguments to be specified.

The other commonly used substring convention is for a starting index and a count (generally named *start* and *count*) to be specified. (This convention is used primarily by subprimitives, and old NIL functions, not COMMON LISP functions.) The substring in question is that starting at the index, and proceeding for count characters. Having a specified count run off the string is an error, and an explicit specification of a count of nil is not allowed. Just about all of the routines which still follow this subsequence convention are low-level NIL functions, in which the count is a required argument, so the question of how it defaults if not specified should not arise any more. Routines using this convention sometimes name these arguments *skip* and *take* rather than *start* and *count*.

string-equal *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
This returns t if the subsequence of *string1* from *start1* (inclusive) to *end1* (exclusive) is equal (using case-independent character comparison, ala **char-equal** (page 95)) to the subsequence of *string2* from *start2* (inclusive) to *end2* (exclusive). The subsequences are guaranteed to be unequal if the subsequence lengths differ.

As a special upwards-compatibility hack, **string-equal** will accept optional position arguments instead of keyword arguments.

string-not-equal *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
The opposite of **string-equal**. Similarly, the subsequences are guaranteed unequal if their lengths differ.

string-lessp *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
string-not-lessp *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
string-greaterp *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
string-not-greaterp *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
Each of these returns t if the specified subsequences satisfy the specified ordering predicate. Basically, the comparison is determined by the first character in the subsequences which differs (see **char-lessp** etc., page 95), unless the subsequences are equal, in which case a shorter subsequence is "less than".

As a special upwards-compatibility hack, **string-lessp** (but *not* the other functions) will accept its arguments positionally rather than by-keyword.

string= *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
string/= *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
string< *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
string<= *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
string>= *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*

string> *string1 string2* &key (*start1* 0) (*start2* 0) *end1 end2*
These functions are similar to those above, but do a case *dependent* character comparison; that is, they compare the characters like **char=** etc. (page 95), rather than **char-equal**.

13.3 Extracting Characters from Strings

char *string index*
Returns the *index*th character (zero-origined) of the string *string*, as a character object. *string* must be a string.

Because *string* must be a string, the result of **char** will always satisfy **string-char-p** (page 19), with all that that implies.

schar *simple-string index*
This is like **char** except that the string must be a simple string. This routine can be efficiently coded by the NIL compiler.

Like **char**, **schar** may be used with **setf** to store into the string.

13.4 String Creation

make-string *length* &key :initial-element
Makes a string *length* long. Since no complicated array options may be specified, this will always be a **simple-string**. *initial-element* must be a character which satisfies **string-char-p**; if it is specified, the returned string contains that character in every element.

If *initial-element* is *not* supplied, the contents of the returned string is *undefined* by COMMON LISP. NIL, which abhors uninitialized memory, will initialize it to zeros (which happen to be, but may not always be, the character #\Null). If NIL were to be transported to another operating system with different virtual memory facilities, then this could change incompatibly.

string-length *string*
Returns the length of the string *string*. *string* must be a string; it is not coerced.

For COMMON LISP, this function is superceded by the generic **length** function (page 49). There should be no noticeable efficiency difference between **length** and **string-length**. Of course, **string-length** will complain if its argument is not a string, whereas **length** will accept any sequence, including **nil**.

string-upcase *string* &key (*start* 0) *end*
string-downcase *string* &key (*start* 0) *end*
Returns a copy of *string* with all characters converted as by **char-upcase** (or **char-downcase**). If *start* and/or *end* are supplied, then only the specified subsequence of the result is affected: the result *always* is the same length as *string*. That is, **string-upcase** could have been defined as

```
(defun string-upcase (string &key (start 0) end)
  (nstring-upcase (copy-seq string) :start start :end end))
```

If no characters of the string are affected by the conversion, one may not depend on whether the result is *string* itself or a copy of it; to guarantee a copy, use copy-seq and nstring-upcase, as in the above example.

nstring-upcase *string* &key (*start 0*) *end*
nstring-downcase *string* &key (*start 0*) *end*
These routines destructively convert all of the characters in the specified subsequence of *string* to upper- or lower-case. *string* is returned.

string-trim *character-bag string*
string-left-trim *character-bag string*
string-right-trim *character-bag string*
These routines return a substring of *string* resulting from trimming the characters in the sequence *character-bag* from one or both ends of *string*. If no characters are trimmed, one may not depend on whether the result is *string* itself or a copy.

```
(string-trim '(#\space #\tab) "  This is a test.  ")
  => "This is a test."
(string-left-trim '(#\space #\tab) "  This is a test.  ")
  => "This is a test.  "
```

13.5 More String Functions

The functions in this section are not provided by COMMON LISP, but are typically inherited from LISP MACHINE LISP, and are heavily used in NIL. That is also why they have not yet been converted to take keyworded arguments (although some of them may be in the future).

string-append &rest *strings*
Returns a string which is the concatenation of all the *strings*. The arguments are coerced to strings using string; in this it differs from the generic sequence function concatenate (page 50), which will not accept symbols, but *will* accept sequences (of characters) other than strings.

substring *string start* &optional *end*
Returns the substring of *string* (coerced to a string using string) from the index *start* up to but not including *end*, which defaults to the length of *string* if nil or not specified. If the specified subsequence is the entire string, one may not depend on whether the returned result is *string* itself, or a copy; to guarantee a copy, subseq (page 50) may be used—however, subseq neither restricts its input to strings nor coerces symbols to strings.

string-reverse *string*
string-nreverse *string*
These are pretty much superceded by reverse (page 52) and nreverse (page 52). Note that for string-nreverse, *string* must be a real string because it is destructively reversed; string-reverse will accept a symbol and return a string.

string-search-char *char string* &optional (*from 0*) *to*
string-reverse-search-char *char string* &optional *from* (*to 0*)
string-search-not-char *char string* &optional (*from 0*) *to*
string-reverse-search-not-char *char string* &optional *from* (*to 0*)

string-search-set *char-set string* &optional (*from 0*) *to*
string-reverse-search-set *char-set string* &optional *from* (*to 0*)
string-search-not-set *char-set string* &optional (*from 0*) *to*
string-reverse-search-not-set *char-set string* &optional *from* (*to 0*)
char-set is coerced into a sequence of characters: it should be a string, or a list or vector of objects acceptable to character (page 96).

string-search *key string* &optional (*from 0*) *to*
string-reverse-search *key string* &optional *from* (*to 0*)

13.6 Implementation Subprimitives

The routines described in this section are very fast routines primitives which are oriented towards being open-compiled. As such, they perform very few niceties like argument defaulting. The versions available in the interpreter probably will do some error checking, but don't count on it. These are the stuff of which higher-level routines are made. Those which take string arguments only accept simple strings.

%string-cons *length fill-character*
Creates a primitive string *length* long, filled with the character *fill-character*. Calls to make-string will compile into calls to this, if possible, so one should not go out of the way to use this.

%string-posq *character string index count*
This searches through *string* starting at *index* and proceeding for *count* characters for the character *character*. If it is found, then the index at which it occurs is returned; otherwise nil is returned. This primitive only looks at the *code* attribute of *character*, ignoring the rest.

%string-eqv *string1 string2 index1 index2 count*
Returns t if the substrings of *string1* and *string2* defined by *index1*, *index2*, and *count* are string=—that is, the same, with case being significant.

%string-replace *destination source destination-index source-index count*
This transfers *count* characters from the simple-string *source* to the simple string *destination*, starting at the given indices.

%string-translate *destination source translation-table destination-index source-index count*
This is a slightly hairier version of %string-replace. Instead of the characters being transferred literally, the *code* of each character taken from the string *source* is used as an index in the string *translation-table* to obtain the code to store instead. For example, string-upcase could be defined

```
(defun string-upcase (string
                      &aux (len (string-length string)))
  (%string-translate
     (make-string len) string *:character-upcase-table
     0 0 len))
```

where *:character-upcase-table has as its value a string char-code-limit long whose *i*th character is the uppercase version of the character with code *i*. Note that this definition of string-upcase is not correct if the input *string* is not a simple-string, and note also how this relates to how string-char-p is defined.

String hashing in NIL is ultimately performed by the CRC instruction.

%string-hash *string crc-table start count*

This performs a hash computation on the substring of *string* starting at character *start* and proceeding for *count* characters. *crc-table* must be a simple bit vector 512 bits (16 VAX longwords) long; it should contain the hash polynomial for use by the CRC instruction. Several hash polynomial tables are provided (they are listed below). The hash computation is initially -1; the result is returned as a signed NIL fixnum—that is, a 32-bit word with the top two bits shifted off. Consult the VAX architecture manual, or some other DEC documentation, to set up other hash polynomials.

For this to be properly useful for incrementally generating CRC computations, this primitive will have to be changed to somehow input and output full 32-bit quantities; such a primitive should be available in some future NIL.

***:autodin-ii-hash-polynomial** *Variable*

***:ccitt-hash-polynomial** *Variable*

***:crc-16-hash-polynomial** *Variable*

This is the one NIL uses for doing intern and sxhash of strings.

14. Hashing

NIL supports a COMMON LISP compatible hash-table facility. This will eventually include the ability to have a hash-table from which associations can be garbage-collected.

14.1 Hash Tables

The following routines are COMMON LISP compatible:

make-hash-table &key :test :rehash-threshold :rehash-size :size
Creates a hash table. The *test* may be one of #'eq, #'eql, or #'equal. NIL additionally provides some others it is able to perform significant optimizations on as primitive (see below). Note that for NIL to use some predicate it must know how to compute a hash code compatible with that predicate's notion of equality; thus, not just any predicate is acceptable.

gethash *key hash-table* &optional *default*
This returns two values. If there is an entry for *key* in *hash-table*, then this returns as its values the value associated with *key*, and t. Otherwise, it returns *default*, and nil.

gethash may be used with setf to add an entry to (or replace an entry in) a hash table.

remhash *key hash-table*
Removes the entry associated with *key* from *hash-table*.

clrhash *hash-table*
Removes all entries from *hash-table*.

hash-table-count *hash-table*
Returns the number of entries in *hash-table*.

14.1.1 Additional Hash-Table Predicates

NIL additionally offers the following predicates for hash-tables:

string-equal

string=

associating hashing routines with predicates, making this list extensible?

14.2 Hash Functions

A hash function for equality predicate (*p x1 x2*) is a function which returns the same value for all *x* which are equivalent according to *p*. The standard LISP hashing function is **sxhash**, which is defined according to the **equal** predicate. **sxhash** is inherited from MACLISP, and is defined by COMMON LISP and LISP MACHINE LISP.

Other hash functions are defined by NIL, for use with other equality predicates. Usually they are not needed, because they are implied in the use of a hash table utilizing a particular equality predicate. By convention, all NIL hashing functions return a non-negative fixnum: it is a fixnum for ease of computation, and non-negative to make modulus operations more trivial.

sxhash *object*

This is the general LISP hashing function, based on the predicate **equal** (page 20). It returns a non-negative integer; in NIL, this will always be a fixnum. Two objects which are **equal** should always **sxhash** to the same thing. If this is not true, then any hash tables which use **sxhash** and **equal** will break. Note that because NIL will have a relocating garbage collector, the hash of an object should never be a function of the address of anything.

string-equal-hash *string*

This function returns a non-negative fixnum such that for all strings which are **string-equal**, their hashes computed by this function will be equal.

string=-hash *string*

Similar; defined by the predicate **string=** (page 113).

sys:sxhash-combine {*hash*}+

This macro might be useful to writers of :**sxhash** methods or special hashing functions. It is a canonical way to combine a fixed number of hash codes. For example, the **sxhash** of a cons does

```
(sys:sxhash-combine (sxhash (car x)) (sxhash (cdr x)))
```

The *hash*es are rotated some fixed amount determined by the number of arguments, and crunched together in some canonical fashion. (There may be a limitation on the number of arguments which are handled, but this will work for some moderate number of arguments.)

14.3 Symbol Tables

Although one can use packages to implement symbol tables, and this has been recommended in the past, it is now better to use a hash table based on the appropriate predicate, and storing an appropriate object as the value. For example, if one had been using a package as a symbol table and then using the symbol after interning it, a hash table could be used using **string-equal** or **string=** as the predicate (as appropriate) and putting a symbol in as the value. Depending on what the symbol is used for, it may be better to use a **defstruct**-created object instead; attributes can be accessed faster off of this than as properties on the plist of a symbol. Secondarily, there is a moderate space inefficiency to generating lots of value cells in NIL, so instead of generating symbols and using their value cells to store things is also better to use a specialized structure.

When the NIL package system is redone, the internal portion of that which does simple symbol-table hacking will be made available for applications where that is truly needed.

15. Packages

Sketchy. This is all going to break, either as a result of COMMON LISP or complete reimplementation and redesign or both.

understanding of simple obarrays/oblists and interning is assumed below

The basic idea of *packages* is that if all programs in a large messy environment like NIL use the same name-space for symbols (the traditional *oblist* or *obarray*), then either they will probably run into problems with naming conflicts, or every programmer is going to have to go out of his way to ensure that each program's names will be unique to that program. For example, by having naming conventions like reader-do-this and reader-frob-uncertainly or (heh heh) pkg-find-package and pkg-create-package. (I didn't name them, they came from LISP MACHINE LISP.)

Packages are an attempt to solve this by allowing each program (or "package") to have its own name-space, but allowing inheritance of symbols from other name-spaces. Each package may be considered to be a symbol table (or oblist or obarray), which has a "superior" package. The act of interning a string in a package (to find or create the symbol it should correspond to) involves looking in that package's symbol table. If there is a symbol with that print name there, then that symbol is returned. Otherwise, try the package's superior package, etc. If one gets to the "top of the tree" and no symbol has been found, then a symbol is created with the given print name, and inserted into the symbol table of the original package.

The NIL package hierarchy looks approximately like this:

```
keyword
global
  *
  sys
    system-internals
    compiler
    file-system
    gc
  format
  user
```

The GLOBAL package has in it all of the symbols which are intended to be used (shared) by everyone. They include function names like car and variable names like char-code-limit. The user package is the package which NIL starts out in, for users to randomly use. New isolated packages should be created under global, like user and format are.

The sys package is sort of a global package for the NIL system. It is initialized to contain those symbols which modules in system-internals, file-system, etc. should share.

The keyword package is for keywords: symbols like :which-operations. Note that it is *not* under global. The result of this is that typing in :open results in a different symbol from typing in open in any other package, resulting in the symbol :open being identified with the keyword package, and the printing functions then being able to print it as :open rather than open.

A little thought about the use of the sys and global packages in the above description will show that they should not be ordinary packages like the "terminal nodes" of the package-tree. Adding symbols to them results in significant behavior change. For this reason, it is supposed to be disallowed by normal interning, and only done by the globalize function (page 122). This check is not done currently. Anyway, it is likely that a different scheme will be concocted eventually.

pkg-find-package *name-or-package* &optional *losing-mode under-pkg*
If *name-or-package* is not a package, then the name is looked up and the package returned.

...

pkg-create-package *name superior-package*

pkg-goto &optional *name*
Setqs pakcage to the pkg-find-package of *name*; convenient for setting the toplevel value (for which it is intended).

package *Variable*

globalize *name* &optional *in-package*

intern *string-or-symbol* &optional *package*

intern-soft *string-or-symbol* &optional *package*
Non-side-effecting version: if no existing symbol is found, nothing is done and nil is returned.

mapatoms *function* &optional (*pkg package*) (*do-superiors? t*)
Calls *function* on all symbols in *pkg* (which is run through pkg-find-package first, so may be a package name). If *do-superiors?* is not nil, then the "superior" packages of *pkg* are examined also. More generally, if *do-superiors?* is nil, the "internal" symbols only of *pkg* are iterated over; otherwise, all symbols accessible from *pkg* are. *function* could conceivably be called more than once on the same symbol.

This is not open-compiled by the NIL compiler, so may suffer from lexical vs local variable problems.

15.1 Modules

COMMON LISP defines a fairly simple way in which one may name *modules*, declare that they have been loaded, and cause them to be loaded if they have not been. While this is intended to be used as a part of the COMMON LISP package system, it is fairly independent and can be used without it. Here it is.

In this sense, a module (not to be confused with the NIL data type **module** which will be renamed someday) is simply an independent subsystem which is treated as a unit. A module can come from one or more files; the number is irrelevant other than to the loading process. COMMON LISP modules are referenced by name; functions which take a module name may be given either a string or a symbol. In the module name, *case matters*, so, for instance, "LSB" is not the same module name as "lsb".

modules *Variable*

This variable has as its value a list of the names of all the modules which have been provided (see below). The implication is that these modules have been loaded.

provide *module-name*

This puts *module-name* on the list ***modules***. A file loaded as part of a module should contain a call to **provide** to tell NIL that that module has been provided.

require *module-name* &optional *pathname*

require is used to load a module if it has not already been loaded. If *module-name* is already provided, then **require** does nothing. If *pathname* is unspecified or **nil**, then the pathname (or pathnames) which need to be loaded in order to provide *module-name* are determined in some system-dependent manner; the method NIL uses is described below. Otherwise, *pathname* may be either a single pathname or a list of pathnames; those pathnames are loaded.

require signals an error if after loading the pathname(s), *module-name* has not been provided.

In NIL, there is an in-core "directory" of module names and the pathnames which must be loaded to provide those modules. This directory can be augmented by the **note-module-pathname** function (below). If the desired module name is not there, then **require** will check for some out-of-core files, and load them if it has not done so already; these files should contain calls to **note-module-pathname**. The files checked for are all named `MODULES.DAT`, and are searched for on the following directories, in order:

(1) The user's home directory (see **user-homedir-pathname**, page 199)
(2) The user's working directory (**user-workingdir-pathname**, page 200)
(3) and `NIL$DISK:[NIL.SITE]`, which is the directory where files specific to a particular NIL installation are kept.

It is allowable, of course, for any or all of these files to be missing.

note-module-pathname *module-name pathname*

This declares that in order to provide *module-name*, *pathname* must be loaded. *pathname* may also be a list of pathnames.

Eventually, higher-level ways of defining packages, systems, and modules will be defined, and use of note-module-pathname will be phased out.

The only modules known about in advance by the NIL system right now LSB and SIMP. Of course, as of this writing, the facilities documented in this section haven't even been installed yet, so there will probably be several more shortly.

16. Defstruct

16.1 Introduction

This chapter is a modification of the description of **defstruct** appearing in the Maclisp Extensions Manual [3]. There are three sorts of changes:

(1) Deletion of topics not applicable to NIL;
(2) Deletion of things which do not yet work in NIL;
(3) Modifications of the defaults, as the NIL **defstruct** is intended to be upwards-compatible with the COMMON LISP **defstruct**.

For these reasons, some of the wording may seem a bit strange, in that the original document is concerned with helping users write code compatible in differing Lisp implementations. **defstruct** is part of the COMMON LISP standard (but not all the parts of it), and the documentation on it will be fixed in the future. Any inaccuracies in this modification of it are purely the fault of GSB.

The keywords which are used in **defstruct** are all interned in the keyword package, just like other keywords in NIL. For compatibility with MACLISP programs, however, **defstruct** will accept those *not* in the keyword package. Conversely, the MACLISP defstruct will check for symbols which have a leading ":" in their names. In NIL, one should use the colons for stylistic consistency.

16.2 A Simple Example

defstruct

defstruct is a macro defining macro. The best way to explain how it works is to show a sample call to **defstruct**, and then to show what macros are defined and what each of them does.

Sample call to **defstruct**:

```
(defstruct (elephant (:type :list))
  color
  (size 17.)
  (name (gensym)))
```

This form expands into a whole rat's nest of stuff, but the effect is to define five macros: elephant-color, elephant-size, elephant-name, make-elephant and alter-elephant. Note that none of these symbols appeared in the original form, they were created by defstruct. The definitions of color, size and name are easy, they expand as follows:

```
(elephant-color x) ==> (car x)
(elephant-size x)  ==> (cadr x)
(elephant-name x)  ==> (caddr x)
```

You can see that **defstruct** has decided to implement an elephant as a list of three things; its color, its size and its name. The expansion of **make-elephant** is somewhat harder to explain, let's look at a few cases:

```
(make-elephant)                  ==>  (list nil 17. (gensym))
(make-elephant :color 'pink)     ==>  (list 'pink 17. (gensym))
(make-elephant :name 'fred :size 100)
                                 ==>  (list nil 100 'fred)
```

As you can see, make-elephant takes a "setq-style" list of part names and forms, and expands into a call to list that constructs such an elephant. Note that the unspecified parts get defaulted to pieces of code specified in the original call to defstruct. Note also that the order of the setq-style arguments is ignored in constructing the call to list. (In the example, 100 is evaluated before 'fred even though 'fred came first in the make-elephant form.) Care should thus be taken in using code with side effects within the scope of a make-elephant. (This particular behaviour will be "fixed" by COMMON LISP, but has not yet been worked into NIL.) Finally, take note of the fact that the (gensym) is evaluated *every time* a new elephant is created (unless you override it).

The explanation of what alter-elephant does is delayed until section 16.4.3, page 129.

So now you know how to construct a new elephant and how to examine the parts of an elephant, but how do you change the parts of an already existing elephant? The answer is to use the setf macro (section 5.9, page 38).

```
(setf (elephant-name x) 'bill)  ==>  (setf (caddr x) 'bill)
```

which is what you want.

And that is just about all there is to defstruct; you now know enough to use it in your code, but if you want to know about all its interesting features, then read on.

16.3 Syntax of defstruct

The general form of a defstruct form is:

```
(defstruct (name option-1 option-2 ... option-n)
  slot-description-1
  slot-description-2
  ...
  slot-description-m)
```

name must be a symbol, it is used in constructing names (such as "make-elephant") and it is given a defstruct-description property of a structure that describes the structure completely.

Each *option-i* is either the atomic name of an option, or a list of the form (*option-name arg . rest*). Some options have defaults for *arg*; some will complain if they are present without an argument; some options complain if they are present *with* an argument. The interpretation of *rest* is up to the option in question, but usually it is expected to be nil.

Each *slot-description-j* is either the atomic name of a slot in the structure, or a list of the form (*slot-name init-code*), or a list of byte field specifications. *init-code* is used by constructor macros (such as make-elephant) to initialize slots not specified in the call to the constructor. If the *init-code* is not specified, then the slot is initialized to whatever is most convenient. (In the elephant example, since the structure was a list, nil was used. If the structure had been a

fixnum array, such slots would be filled with zeros.)

A byte field specification looks like: (*field-name bytespec*) or (*field-name bytespec init-code*). Note that since a byte field specification is always a list, a list of byte field specifications can never be confused with the other cases of a slot description. The byte field feature of defstruct may be undergoing change in NIL due to the incompatible change of bytespec format (see section 10.9, page 84), so is discouraged for the present.

16.4 Options to defstruct

The following sections document each of the options defstruct understands in detail.

On the Lisp Machine and in NIL, all these defstruct options are interned on the keyword package.

16.4.1 :type

The :type option specifies what kind of lisp object defstruct is going to use to implement your structure, and how that implementation is going to be carried out. The :type option is illegal without an argument. If the :type option is not specified, then defstruct will choose an appropriate default: in NIL, defstruct will implement the structure as a vector-like object, which will be defined as a type whose name is the name given to defstruct. One can then check for objects of this type with typep. (This differs from the way defstruct *currently* operates in MACLISP and LISP MACHINE LISP.) It is possible for the user to teach defstruct new ways to implement structures, the interested reader is referred to section 16.6, page 138, for more information. Many useful types have already been defined for the user. A table of these "built in" types follows.

:list

Uses a list. This is the default in MULTICS MACLISP.

:named-list

Like :list, except the car of each instance of this structure will be the name symbol of the structure. This is the only "named" structure type defined on Multics and is the default named type there. (See the :named option documented in section 16.4.4, page 131.)

:tree

Creates a binary tree out of conses with the slots as leaves. The theory is to reduce car-cdring to a minimum. The :include option (section 16.4.9, page 133) does not work with structures of this type.

:list*

Similar to :list, but the last slot in the structure will be placed in the cdr of the final cons of the list. Some people call objects of this type "dotted lists". The :include option (section 16.4.9, page 133) does not work with structures of this type.

:array

Uses an array object (*not* a symbol with an array property). This is the default on Lisp Machines. Eventually, many of the same hairy array options which **defstruct** supports on the Lisp Machine will be supported in NIL; at this time, however, NIL users are advised to just use the default, or perhaps **:vector**.

:sfa

Uses an SFA. The constructor macros for this type accept the keywords **:sfa-function** and **:sfa-name**. Their arguments (evaluated, of course) are used, respectively, as the function and the printed representation of the SFA. See also the **:sfa-function** (section 16.4.12, page 134) and **:sfa-name** (section 16.4.13, page 135) options. (SFAs are available in NIL for compatibility with PDP-10 MACLISP. They should not normally be used, and are not documented in the NIL manual.)

:vector

Uses an vector. This will be a simple-vector.

:named-vector

Like vector, except element number 0 always contains the name symbol of the structure. Note that this is *not* the default *named* type in NIL, **:extend** is.

:extend

This is the default named type in NIL. Normally you don't need to know that it has this weird name, because this has been the default defstruct type if **:named** is specified for a while, and it is now the default type period. See also the **:class-symbol** option (section 16.4.11, page 134).

16.4.2 :constructor

The **:constructor** option specifies the name to be given to the constructor macro. Without an argument, or if the option is not present, the name defaults to the concatenation of "make-" with the name of the structure. If the option is given with an argument of **nil**, then no constructor is defined. Otherwise the argument is the name of the constructor to define. Normally the syntax of the constructor **defstruct** defines is:

(*constructor-name*
keyword-1 code-1
keyword-2 code-2
...
keyword-n code-n)

Each *keyword-i* must be the name of a slot in the structure, or one of the special keywords allowed for the particular type of structure being constructed. All of these keywords are symbols interned in the keyword package, although for upwards-compatibility (this is new behaviour) that is not required. For each keyword that is the name of a slot, the constructor expands into code to make an instance of the structure using *code-i* to initialize slot *keyword-i*. Unspecified slots default to the forms given in the original **defstruct** form, or, if none was given there, to some convenient value such as **nil** or **0**.

For keywords that are not names of slots, the use of the corresponding code varies. Usually it controls some aspect of the instance being constructed that is not otherwise constrained. The only one of these which is used in NIL is the :sfa-function option (section 16.4.12, page 134).

If the :constructor option is given as (:constructor *name arglist*), then instead of making a keyword driven constructor, defstruct defines a "function style" constructor. 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:

```
(:constructor make-foo
              (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines make-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. The arguments from the fourth one on are collected into a list and used to initialize the d slot. If there are three or less arguments, then nil is placed in the d slot. The e slot is *not initialized*. It's value will be something convenient like nil or 0. And finally the f slot is initialized to contain the symbol eff.

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.

Since there is so much freedom in defining constructors this way, it would be cruel to only allow the :constructor option to be given once. So, by special dispensation, you are allowed to give the :constructor option more than once, so that you can define several different constructors, each with a different syntax.

Note that even these "function style" constructors do not currently guarantee that their arguments will be evaluated in the order that you wrote them.

16.4.3 :alterant

The :alterant option defines a macro that can be used to change the value of several slots in a structure together. Without an argument, or if the option is not present, the name of the alterant macro defaults to the concatenation of "alter-" with the name of the structure. If the option is given with an argument of nil, then no alterant is defined. Otherwise the argument is the name of the alterant to define. The syntax of the alterant macro defstruct defines is:

(*alterant-name code*
slot-name-1 code-1
slot-name-2 code-2
...
slot-name-n code-n)

code should evaluate to an instance of the structure; each *code-i* is evaluated and the result is

made to be the value of slot *slot-name-i* of that structure. The slots are all altered in parallel after all code has been evaluated. (Thus you can use an alterant macro to exchange the contents to two slots.) As for the keyworded constructor macro, the *slot-name-i* should be symbols interned in the keyword package, although (again) that is not required.
Example:

```
(defstruct (lisp-hacker (:type :list)
                        :conc-name
                        :default-pointer
                        :alterant)
  (favorite-macro-package nil)
  (unhappy? t)
  (number-of-friends 0))

(setq lisp-hacker (make-lisp-hacker))
```

Now we can perform a transformation:

```
(alter-lisp-hacker lisp-hacker
        favorite-macro-package 'defstruct
        number-of-friends 23.
        unhappy? nil)

==> ((lambda (G0009)
       ((lambda (G0011 G0010)
          (setf (car G0009) 'defstruct)
          (setf (cddr G0009) G0011)
          (setf (cadr G0009) G0010))
        23.
        nil))
     lisp-hacker)
```

Although it appears from this example that your forms will be evaluated in the order in which you wrote them, this is not currently guaranteed.

Alterant macros are particularly good at simultaneously modifying several byte fields that are allocated from the same word. They produce better code than you can by simply writing consecutive setfs. They also produce better code when modifying several slots of a structure that uses the :but-first option (section 16.4.17, page 135).

For defstruct types whose accessors take more than one argument, all of those arguments must be supplied to the alterant macro in place of just the usual one. (See section 16.6.3.2, page 140 for how accessors with more than one argument can come to be, there are no built-in defstruct types with this property.)

16.4.4 :named

This option tells defstruct that you desire your structure to be a "named structure". In PDP-10 MACLISP this means you want your structure implemented with a :named-hunk, :named-list or :named-vector. On a Lisp Machine this indicates that you desire either a :named-array or a :named-array-leader or a :named-list. On Multics this indicates that you desire a :named-list. In NIL this indicates that you desire a :extend, a :named-vector or a :named-list. defstruct bases its decision as to what named type to use on whatever value you did or didn't give to the :type option; in NIL, the default named type is :extend, and :named is the default—the significance of this was explained in section 16.4.1, page 127. It is an error to use this option with an argument.

16.4.5 :predicate

The :predicate option causes defstruct to generate a predicate to recognize instances of the structure. Naturally it only works for some defstruct types. Currently it works for all the named types as well as the types :sfa (PDP-10 MACLISP and NIL only) and :extend (NIL only). The argument to the :predicate option is the name of the predicate. If it is present without an argument, then the name is formed by concatenating "-p" to the end of the name symbol of the structure. If the option is not present, then no predicate is generated. Example:

```
(defstruct (foo :named :predicate)
  foo-a
  foo-b)
```

defines a single argument function, foo-p, that is true only of instances of this structure.

16.4.6 :print

The :print option allows the user to control the printed representation of his structure in an implementation independent way:

```
(defstruct (pair :named
                 (:print "{~S . ~S}"
                         (pair-first pair)
                         (pair-second pair)))
  pair-first
  pair-second)
```

The arguments to the :print option are used as if they were arguments to the format function (page 187), except that the first argument (the stream) is omitted. They are evaluated in an environment where the name symbol of the structure (pair in this case) is bound to the instance of the structure to be printed.

This option presently only works on Lisp Machines and in NIL, using the defstruct types :named-array and :extend respectively. We hope to make it work in PDP-10 MACLISP for the :named-hunk type soon. In MULTICS MACLISP, this option is ignored. Notice that if you just specify the :named option without giving an explicit :type option, each defstruct implementation will default to a named type that can control printing if at all possible.

16.4.7 :default-pointer

Normally the accessors are defined to be macros of exactly one argument. (They check!) But if the :default-pointer option is present then they will accept zero or one argument. When used with one argument, they behave as before, but given no arguments, they expand as if they had been called on the argument to the :default-pointer option. An example is probably called for:

```
(defstruct (room (:type :tree)
                 (:default-pointer **current-room**))
  (room-name 'y2)
  (room-contents-list nil))
```

Now the accessors expand as follows:

```
(room-name x)          ==> (car x)
(room-name)            ==> (car **current-room**)
```

If no argument is given to the :default-pointer option, then the name of the structure is used as the "default pointer". :default-pointer is most often used in this fashion.

16.4.8 :conc-name

Frequently all the accessor macros of a structure will want to have names that begin the same way; usually with the name of the structure followed by a dash. The :conc-name option allows the user to specify this prefix. Its argument should be a symbol whose print name will be concatenated onto the front of the slot names when forming the accessor macro names. If the argument is not given, then the name of the structure followed by a dash is used, as it is if the :conc-name option is not present. (This is different than it used to be!) If it is desired that the slot names, as specified, be used as the accessor macros, then (:conc-name nil) may be used. An example illustrates a common use of the :conc-name option along with the :default-pointer option:

```
(defstruct (location :default-pointer
                     :conc-name)
  (x 0)
  (y 0)
  (z 0))
```

Now if you say

```
(setq location (make-location x 1 y 34 z 5))
```

it will be the case that

```
(location-y)
```

will return 34. Note well that the name of the slot ("y") and the name of the accessor macro for that slot ("location-y") are different.

16.4.9 :include

The :include option inserts the definition of its argument at the head of the new structure's definition. In other words, the first slots of the new structure are equivalent to (i.e. have the same names as, have the same inits as, etc.) the slots of the argument to the :include option. The argument to the :include option must be the name of a previously defined structure of the same type as the new one. If no type is specified in the new structure, then it is defaulted to that of the included one. It is an error for the :include option to be present without an argument. Note that :include does not work on certain types of structures (e.g. structures of type :tree or :list*). Note also that the :conc-name, :default-pointer, :but-first and :callable-accessors options only apply to the accessors defined in the current defstruct; no new accessors are defined for the included slots.

An example:

```
(defstruct (person (:type :list))
  name
  age
  sex)

(defstruct (spaceman (:include person)
                     (:conc-name nil)
                     :default-pointer)
  helmet-size
  (favorite-beverage 'tang))
```

Now we can make a spaceman like this:

```
(setq spaceman (make-spaceman :name 'buzz
                              :age 45.
                              :sex t
                              :helmet-size 17.5))
```

To find out interesting things about spacemen:

```
(helmet-size)           ==> (cadddr spaceman)
(person-name spaceman)  ==> (car spaceman)
(favorite-beverage x)   ==> (car (cddddr x))
```

As you can see the accessors defined for the person structure have names that start with "person-" and they only take one argument. The names of the accessors for the last two slots of the spaceman structure are the same as the slot names, but they allow their argument to be omitted. The accessors for the first three slots of the spaceman structure are the same as the accessors for the person structure.

Often, when one structure includes another, the default initial values supplied by the included structure will be undesirable. These default initial values can be modified at the time of inclusion by giving the :include option as:

(:include *name new-init-1 ... new-init-n*)

Each *new-init-i* is either the name of an included slot or of the form (*included-slot-name new-init*). If it is just a slot name, then in the new structure (the one doing the including) that slot will have no initial value. If a new initial value is given, then that code replaces the old initial value code for that slot in the new structure. The included structure is unmodified.

16.4.10 :copier

This option causes defstruct to generate a single argument function that will copy instances of this structure. The argument to the :copier option is the name of the copying function. If this option is present without an argument, then the name is formed by concatenating "copy-" with the name of the structure.

Example:

```
(defstruct (coat-hanger (:type :list) :copier)
  current-closet
  wire-p)
```

Generates a function approximately like:

```
(defun copy-coat-hanger (x)
  (list (car x) (cadr x)))
```

16.4.11 :class-symbol

For use with the :extend defstruct type available only in NIL (section 16.4.1, page 128), this option allows the user to control *how* the flavor definition is performed. This option *must* be given a variable name as an argument: the value of that variable is used as the flavor (class) object of the object which the defstruct-defined constructor will create. defstruct will not define the flavor.

This option was originally implemented for bootstrapping purposes, so that typed objects in NIL could be created before the flavor system was fully loaded. Eventually it will be fully outmoded by extensions to the flavor system, which already has the capability of defining accessor macros for instance variables.

16.4.12 :sfa-function

Available in PDP-10 MACLISP and in NIL, this option allows the user to specify the function that will be used in structures of type :sfa. Its argument should be a piece of code that evaluates to the desired function. Constructor macros for this type of structure will take :sfa-function as a keyword whose argument is also the code to evaluate to get the function, overriding any supplied in the original defstruct form.

If :sfa-function is not present anywhere, then the constructor will use the name-symbol of the structure as the function.

16.4.13 :sfa-name

Available only in PDP-10 MACLISP and NIL, this option allows the user to specify the object that will be used in the printed representation of structures of type :sfa. Its argument should be a piece of code that evaluates to that object. Constructor macros for this type of structure will take :sfa-name as a keyword whose argument is also the code to evaluate to get the object to use, overriding any supplied in the original defstruct form.

If :sfa-name is not present anywhere, then the constructor will use the name-symbol of the structure as the function.

16.4.14 :size-symbol

The :size-symbol option allows a user to specify a symbol whose value will be the "size" of the structure. The exact meaning of this 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 symbol 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 symbol.

16.4.15 :size-macro

Similar to :size-symbol. A macro of no arguments is defined that expands into the size of the structure. The name of this macro defaults as with :size-symbol.

16.4.16 :initial-offset

This option 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 fixnum, 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 you structure, otherwise you will be unable to make use of the slots that defstruct has left unused.

16.4.17 :but-first

This option is best explained by example:

```
(defstruct (head (:type :list)
                 (:conc-name nil)
                 (:default-pointer person)
                 (:but-first person-head))
  nose
  mouth
  eyes)
```

So now the accessors expand like this:

```
(nose x)        ==> (car (person-head x))
(nose)          ==> (car (person-head person))
```

The theory is that :but-first's argument will likely be an accessor from some other structure, and it is never expected that this structure will be found outside of that slot of that other structure. (In the example I had in mind that there was a person structure which had a slot accessed by person-head.) It is an error for the :but-first option to be used without an argument.

16.4.18 :callable-accessors

This option controls whether the accessors defined by defstruct will work as "functional arguments" (as the first argument to mapcar, for example). On the Lisp Machine and in NIL accessors are callable by default, but in PDP-10 MACLISP it is expensive to make this work, so they are only callable if you ask for it. (Currently on Multics the feature doesn't work at all...) The argument to this option is nil to indicate that the feature should be turned off, and t to turn the feature on. If the option is present with no argument, then the feature is turned on.

16.4.19 :eval-when

Normally the macros defined by defstruct are defined at eval-time, compile-time and at load-time. This option allows the user to control this behavior. (:eval-when (eval compile)), for example, will cause the macros to be defined only when the code is running interpreted and inside the compiler, no trace of defstruct will be found when running compiled code. (See eval-when, page 25.)

Using the :eval-when option is preferable to wrapping an eval-when around a defstruct form, since nested eval-whens can interact in unexpected ways.

16.4.20 :property

For each structure defined by defstruct, a property list is maintained for the recording of arbitrary properties about that structure.

The :property option can be used to give a defstruct an arbitrary property. (:property *property-name value*) gives the defstruct a *property-name* property of *value*. Neither argument is evaluated. To access the property list, the user will have to look inside the defstruct-description structure himself, he is referred to section 16.5, page 137, for more information.

16.4.21 A Type Used As An Option

In addition to the options listed above, any currently defined type (a legal argument to the :type option) can be used as a option. This is mostly for compatibility with the old Lisp Machine defstruct. It allows you to say just *type* when you should be saying (:type *type*). Use of this feature in new code is discouraged. It is an error to give an argument to a type used as an option in this manner.

16.4.22 Other Options

Finally, if an option isn't found among those listed above, defstruct checks the property list of the name of the option to see if it has a non-null :defstruct-option property. If is does have such a property, then if the option was of the form (*option-name value*), it is treated just like (:property *option-name value*). That is, the defstruct is given an *option-name* property of *value*. If such an option is used without an argument, it is treated just like (:property *option-name* t). That is, it is treated as if the argument was t.

This provides a primitive way for the user to define his own options to defstruct. Several of the options listed above are actually implemented using this mechanism.

16.5 The defstruct-description Structure

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 (section 16.6, page 138). Lisp Machine and NIL programmers will find that the symbols found only in this section are all interned in the "systems-internals" package ("SI" for short).

Whenever the user defines a new structure using defstruct, defstruct creates an instance of the defstruct-description structure. This structure can be found as the 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, etc.

The defstruct-description structure is defined something like this: (This is a bowdlerized version of the real thing, I have left out a lot of things you don't need to know unless you are actually reading the code.)

```
(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, something like spaceship or phone-book-entry.

The size slot contains the total number of slots 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 section 16.4.20, page 136, and section 16.4.22, page 137).

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: (another bowdlerized defstruct)

```
(defstruct (defstruct-slot-description
             (:default-pointer slot-description)
             (:conc-name defstruct-slot-description-))
  number
  ppss
  init-code
  ref-macro-name)
```

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 code associated with the type of the structure. See section 16.6, page 138.

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 nil.

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 %%defstruct-empty%%.

The ref-macro-name slot contains the symbol that is defined as an accessor that references this slot.

16.6 Extensions to defstruct

defstruct-define-type

The macro defstruct-define-type can be used to teach defstruct about new types it can use to implement structures.

16.6.1 A Simple Example

Let us start by examining a sample call to defstruct-define-type. This is how the :list type of structure might have been defined:

```
(defstruct-define-type :list
  (:cons (initialization-list description keyword-options)
         :list
         (cons 'list initialization-list))
  (:ref (slot-number description argument)
        (list 'nth slot-number argument)))
```

This is the minimal example. We have provided defstruct with two pieces of code, one for consing up forms to construct instances of the structure, the other to cons up forms to reference various elements of the structure.

From the example we can see that the constructor consing code is going to be run in an environment where the variable initialization-list is bound to a list which is the initializations to the slots of the structure arranged in order. The variable description will be bound to the defstruct-description structure for the structure we are consing a constructor for. (See section 16.5, page 137.) The binding of the variable keyword-options will be described later. Also the symbol :list appears after the argument list, this conveys some information to defstruct about how the constructor consing code wants to get called.

The reference consing code gets run with the variable slot-number bound to the number of the slot that is to be referenced and the variable argument bound to the code that appeared as the argument to the accessor macro. The variable description is again bound to the appropriate instance of the defstruct-description structure.

This simple example probably tells you enough to be able to go ahead and implement other structure types, but more details follow.

16.6.2 Syntax of defstruct-define-type

The syntax of defstruct-define-type is

```
(defstruct-define-type type
  option-1
  ...
  option-n)
```

where each *option-i* is either the symbolic name of an option or a list of the form (*option-i* . *rest*). (Actually *option-i* is the same as (*option-i*).) Different options interpret *rest* in different ways.

The symbol *type* is given a defstruct-type-description property of a structure that describes the type completely.

16.6.3 Options to defstruct-define-type

This section is a catalog of all the options currently known about by defstruct-define-type.

16.6.3.1 :cons

The :cons option to defstruct-define-type is how the user supplies defstruct with the necessary code that it needs to cons up a form that will construct an instance of a structure of this type.

The :cons option has the syntax:

```
(:cons (inits description keywords) kind
       body)
```

body is some code that should construct and return a piece of code that will construct, initialize and return an instance of a structure of this type.

The symbol *inits* will be bound to the code that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol *kind*. There are currently two kinds of initialization. There is the :list kind, where *inits* is bound to a list of initializations, in the correct order, with nils in uninitialized slots. And there is the :alist kind, where *inits* is bound to an alist with pairs of the form (*slot-number . init-code*).

The symbol *description* will be bound to the instance of the defstruct-description structure (section 16.5, page 137) that defstruct maintains for this particular structure. This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol *keywords* will be bound to a alist with pairs of the form (*keyword . value*), where each *keyword* was a keyword supplied to the constructor macro that wasn't the name of a slot, and *value* was the "code" that followed the keyword. (See section 16.6.3.6, page 142, and section 16.4.2, page 128.)

It is an error not to supply the :cons option to defstruct-define-type.

16.6.3.2 :ref

The :ref option to defstruct-define-type is how the user supplies defstruct with the necessary code that it needs to cons up a form that will reference an instance of a structure of this type.

The :ref option has the syntax:

```
(:ref (number description arg-1 ... arg-n)
      body)
```

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol *number* will be bound to the location of the slot that the is to be referenced. This is the same number that is found in the number slot of the defstruct-slot-description structure (section 16.5, page 137).

The symbol *description* will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure.

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The *last* argument is the one that will be defaulted by the :default-pointer option (section 16.4.7, page 132). defstruct will check that the user has supplied exactly n arguments to the accessor macro before calling the reference consing code.

It is an error not to supply the :ref option to defstruct-define-type.

16.6.3.3 :predicate

The :predicate option to defstruct-define-type is how defstruct is told how to produce predicates for a particular type when the :predicate option to defstruct is used (section 16.4.5, page 131). Its syntax is:

```
(:predicate (description name)
            body)
```

The variable *description* will be bound to the defstruct-description structure maintained for the structure we are to generate a predicate for. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code to evaluate to return the defining form for the predicate. A typical use of this option might look like:

```
(:predicate (description name)
            `(defun ,name (x)
               (and (frobbozp x)
                    (eq (frobbozref x 0)
                        ',(defstruct-description-name)))))
```

16.6.3.4 :overhead

The :overhead option to defstruct-define-type is how the user declares to defstruct that the implementation of this particular type of structure "uses up" some number of slots locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of :overhead is:

```
(:overhead n)
```

where *n* is a fixnum that says how many locations of overhead this type needs.

This number is only used by the :size-macro and :size-symbol options to defstruct. (See section 16.4.15, page 135, and section 16.4.14, page 135.)

16.6.3.5 :named

The :named option to defstruct-define-type controls the use of the :named option to defstruct. With no argument the :named option means that this type is an acceptable "named structure". With an argument, as in (:named *type-name*), the symbol *type-name* should be that name of some other structure type that defstruct should use if someone asks for the named version of this type. (For example, in the definition of the :list type the :named option is used like this: (:named :named-list).)

16.6.3.6 :keywords

The :keywords option to defstruct-define-type allows the user to define constructor keywords (section 16.4.2, page 128) for this type of structure. (For example the :make-array constructor keyword for structures of type :array on Lisp Machines.) The syntax is:

```
(:keywords keyword-1 ... keyword-n)
```

where each *keyword-i* is a symbol that the constructor conser expects to find in the *keywords* alist (section 16.6.3.1, page 139).

16.6.3.7 :defstruct-options

The :defstruct-options option to defstruct-define-type is similar to the :keywords option. It is used to define new options that may appear in the options part of a defstruct for a structure of this type. Its syntax is:

```
(:defstruct-options option-1 ... option-n)
```

This defines each *option-i* to be a option to defstruct that can be used with structures of this type. For example, the :array defstruct type for the Lisp Machine uses the :defstruct-options option as follows:

```
(:defstruct-options :make-array)
```

Currently this just works by giving each *option-i* a non-null :defstruct-option property (see section 16.4.22, page 137), but soon it will check to be sure that each *option-i* is *only* used as an option with structures of this type.

16.6.3.8 :defstruct

The :defstruct option to defstruct-define-type allows the user to run some code and return some forms as part of the expansion of the defstruct macro.

The :defstruct option has the syntax:

```
(:defstruct (description)
            body)
```

body is a piece of code that will be run whenever defstruct is expanding a defstruct form that defines a structure of this type. The symbol *description* will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure.

The value returned by the :defstruct option should be a *list* of forms to be included with those that the defstruct expands into. Thus, if you only want to run some code at defstruct expand time, and you don't want to actually output any additional code, then you should be careful to return nil from the code in this option.

16.6.3.9 :copier

The :copier option to defstruct-define-type allows the user to tell defstruct how to generate the copier functions required by the :copier option to defstruct (section 16.4.10, page 134). This option is entirely optional, because defstruct already has enough information to write an adequate copier function for any given type given the information supplied to the :ref and :cons options to defstruct-define-type. However, it is sometimes desirable to teach defstruct a *better* way to copy a particular type of structure.

The :copier option has the syntax:

```
(:copier (description name)
         body)
```

Similar to the :predicate option, *description* is bound to the instance of the defstruct-description structure maintained for this structure, *name* is bound to the symbol to be defined, and *body* is some code to evaluate to get the defining form. For example:

```
(:copier (description name)
         '(defmacro ,name (x)
            '(copy-frobboz ,x)))
```

16.6.3.10 :implementations

The :implementations option to defstruct-define-type is primarily useful to the maintainers of defstruct in keeping control of the variations in defstruct types available in different implementations. Its syntax is:

```
(:implementations arg-1 ... arg-n)
```

This makes the defstruct-define-type in which it appears only take effect in those implementations of LISP in which (status feature *arg-i*) is true for at least one of the *arg-i*.

17. The LOOP Iteration Macro

17.1 Introduction

loop is a LISP macro which provides a programmable iteration facility. The same loop module operates compatibly in LISP MACHINE LISP, MACLISP (PDP-10 and MULTICS), and NIL, and a moderately compatible package was developed for the MDL programming environment. loop was inspired by the "FOR" facility of CLISP in INTERLISP; however, it is not compatible and differs in several details.

The general approach is that a form introduced by the word loop generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body which may be executed several times, and some exit (*epilogue*) code. Variables may be declared local to the loop. The features are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The loop form consists of a series of clauses, each introduced by a keyword symbol. Forms appearing in or implied by the clauses of a loop form are classed as those to be executed as initialization code, body code, and/or exit code; within each part of the template that loop fills in, they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side-effects may be used, and one piece of code may depend on following another for its proper operation. This is the principal philosophy difference from INTERLISP's "FOR" facility.

Note that loop forms are intended to look like stylized English rather than LISP code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English. Some find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to stick to do.

Here are some examples to illustrate the use of loop.

```
(defun print-elements-of-list (list-of-elements)
    (loop for element in list-of-elements
          do (print element)))
```

The above function prints each element in its argument, which should be a list. It returns nil.

```
(defun gather-alist-entries (list-of-pairs)
    (loop for pair in list-of-pairs
          collect (car pair)))
```

gather-alist-entries takes an association list and returns a list of the "keys"; that is, (gather-alist-entries '((foo 1 2) (bar 259) (baz))) returns (foo bar baz).

```
(defun extract-interesting-numbers (start-value end-value)
   (loop for number from start-value to end-value
         when (interesting-p number) collect number))
```

The above function takes two arguments, which should be fixnums, and returns a list of all the numbers in that range (inclusive) which satisfy the predicate interesting-p.

```
(defun find-maximum-element (an-array)
   (loop for i from 0 below (array-dimension an-array 0)
         maximize (aref an-array i)))
```

find-maximum-element returns the maximum of the elements of its argument, a one-dimensional array.

```
(defun my-remove (object list)
   (loop for element in list
         unless (equal object element) collect element))
```

my-remove is like the COMMON LISP function remove, utilizing the equal predicate. (This is like MACLISP delete, but copies the list rather than destructively splicing out elements.)

```
(defun find-frob (list)
   (loop for element in list
         when (frobp element) return element
         finally (ferror nil "No frob was found in the list ~S"
                         list)))
```

This returns the first element of its list argument which satisfies the predicate frobp. If none is found, an error is generated.

17.2 Clauses

Internally, loop constructs a prog which includes variable bindings, pre-iteration (initialization) code, post-iteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after executing the body).

A *clause* consists of the keyword symbol and any Lisp forms and keywords which it deals with. For example,

`(loop for x in l do (print x))`,

contains two clauses, "for x in (car foo)" and "do (print x)". Certain of the parts of the clause will be described as being *expressions*, e.g. (car foo) in the above. An expression is a

single LISP form. Obviously, it must be followed immediately by either the end of the loop form, or by a loop keyword. Certain clauses take what is called a *multiple expression*; this may be a single LISP form, or a series of forms implicitly collected with progn. A *multiple expression* is terminated by the next following atom, which is taken to be a keyword. As a general rule, loop clauses which utilize LISP forms deal only with single expressions, with the exception of those for which the forms are evaluated for effect: specifically, only the do, initially, and finally clauses allow multiple expressions.

This syntax differs slightly from earlier versions of loop (which are probably still in use in LISP implementations other than NIL), in which *all* expressions were treated as multiple expressions. The reason for the change is twofold: first, a common syntactic error in using loop is to accidentally omit the do keyword (causing the expressions meant to be executed for effect to become part of the preceding clause). Second, it is anticipated that loop will support an "implied do" sometime in the future, making this omission in fact syntactically correct.

loop uses print-name equality to compare keywords so that loop forms may be written without package prefixes; in LISP implementations that do not have packages, eq is used for comparison.

Bindings and iteration variable steppings may be performed either sequentially or in parallel; this affects how the stepping of one iteration variable may depend on the value of another. The syntax for distinguishing the two will be described with the corresponding clauses. When a set of things is "in parallel", all of the bindings produced will be performed in parallel by a single lambda binding. Subsequent bindings will be performed inside of that binding environment.

17.2.1 Iteration-Driving Clauses

These clauses all create a *variable of iteration*, which is bound locally to the loop and takes on a new value on each successive iteration. Note that if more than one iteration-driving clause is used in the same loop, several variables are created which all step together through their values; when any of the iterations terminates, the entire loop terminates. Nested iterations are not generated; for those, you need a second loop form in the body of the loop. In order to not produce strange interactions, iteration driving clauses are required to precede any clauses which produce "body" code: that is, all except those which produce prologue or epilogue code (initially and finally), bindings (with), the named clause, and the iteration termination clauses (while and until).

Clauses which drive the iteration may be arranged to perform their testing and stepping either in series or in parallel. They are by default grouped in series, which allows the stepping computation of one clause to use the just-computed values of the iteration variables of previous clauses. They may be made to step "in parallel", as is the case with the do special form, by "joining" the iteration clauses with the keyword and. The form this typically takes is something like

```
(loop ... for x = (f) and for y = init then (g x) ...)
```

which sets x to (f) on every iteration, and binds y to the value of *init* for the first iteration, and on every iteration thereafter sets it to (g x), where x still has the value from the *previous* iteration. Thus, if the calls to f and g are not order-dependent, this would be best written as

```
(loop ... for y = init then (g x) for x = (f) ...)
```

because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, the example

```
(loop for sublist on some-list
      and for previous = 'undefined then sublist
      ...)
```

which is equivalent to the do construct

```
(do ((sublist some-list (cdr sublist))
     (previous 'undefined sublist))
    ((null sublist) ...)
  ...)
```

in terms of stepping, would be better written as

```
(loop for previous = 'undefined then sublist
      for sublist on some-list
      ...)
```

When iteration driving clauses are joined with **and**, if the token following the **and** is not a keyword which introduces an iteration driving clause, it is assumed to be the same as the keyword which introduced the most recent clause; thus, the above example showing parallel stepping could have been written as

```
(loop for sublist on some-list
      and previous = 'undefined then sublist
      ...)
```

because the keyword **for** is implied after the **and**.

The order of evaluation in iteration-driving clauses is that those expressions which are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions which are evaluated each time around the loop are evaluated in order in the body.

One common and simple iteration driving clause is **repeat**:

repeat *expression*

> This evaluates *expression* (during the variable binding phase), and causes the **loop** to iterate that many times. *expression* is expected to evaluate to a fixnum. If *expression* evaluates to a zero or negative result, the body code will not be executed.

All remaining iteration driving clauses are subdispatches of the keyword **for**, which is synonomous with **as**. In all of them a *variable of iteration* is specified. Note that, in general, if an iteration driving clause implicitly supplies an endtest, the value of this iteration variable as the loop is exited (i.e., when the epilogue code is run) is undefined. (This is discussed in more detail in section 17.6.)

Here are all of the varieties of **for** clauses. Optional parts are enclosed in curly brackets. The *data-types* as used here are discussed fully in section 17.4.

for *var* {*data-type*} **in** *expr1* {**by** *expr2*}

> This iterates over each of the elements in the list *expr1*. If the **by** subclause is present, *expr2* is evaluated once on entry to the loop to supply the function to be used to fetch successive sublists, instead of **cdr**.

for *var* {*data-type*} on *expr1* {by *expr2*}

This is like the previous for format, except that *var* is set to successive sublists of the list instead of successive elements. Note that since *var* will always be a list, it is not meaningful to specify a *data-type* unless *var* is a *destructuring pattern*, as described in the section on *destructuring*, page 158. Note also that loop uses a null rather than an atom test to implement both this and the preceding clause.

for *var* {*data-type*} = *expr*

On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* {*data-type*} = *expr1* then *expr2*

var is bound to *expr1* when the loop is entered, and set to *expr2* (re-evaluated) at all but the first iteration. Since *expr1* is evaluated during the binding phase, it cannot reference other iteration variables set before it; for that, use the following:

for *var* {*data-type*} first *expr1* then *expr2*

This sets *var* to *expr1* on the first iteration, and to *expr2* (re-evaluated) on each succeeding iteration. The evaluation of both expressions is performed *inside* of the loop binding environment, before the loop body. This allows the first value of *var* to come from the first value of some other iteration variable, allowing such constructs as

```
(loop for term in poly
      for ans first (car term)
              then (gcd ans (car term))
      finally (return ans))
```

for *var* {*data type*} from *expr1* {to *expr2*} {by *expr3*}

This performs numeric iteration. *var* is initialized to *expr1*, and on each succeeding iteration is incremented by *expr3* (default 1). If the to phrase is given, the iteration terminates when *var* becomes greater than *expr2*. Each of the expressions is evaluated only once, and the to and by phrases may be written in either order. downto may be used instead of to, in which case *var* is decremented by the step value, and the endtest is adjusted accordingly. If below is used instead of to, or above instead of downto, the iteration will be terminated before *expr2* is reached, rather than after. Note that the to variant appropriate for the direction of stepping must be used for the endtest to be formed correctly; i.e. the code will not work if *expr3* is negative or zero. If no limit-specifying clause is given, then the direction of the stepping may be specified as being decreasing by using downfrom instead of from. upfrom may also be used instead of from; it forces the stepping direction to be increasing. The *data-type* defaults to fixnum. Thus, the idiom for stepping through a typical COMMON LISP start/end range in which the start is inclusive and the end is exclusive, is

```
for var from start below end
```

for *var* {*data-type*} being *expr* and its *path* ...

for *var* {*data-type*} being {each|the} *path* ...

This provides a user-definable iteration facility. *path* names the manner in which the iteration is to be performed. The ellipsis indicates where various path dependent preposition/expression pairs may appear. See the section on Iteration Paths (page 161) for complete documentation.

17.2.2 Bindings

The with keyword may be used to establish initial bindings, that is, variables which are local to the loop but are only set once, rather than on each iteration. The with clause looks like:

with *var1* {*data-type*} { = *expr1*}
{and *var2* {*data-type*} { = *expr2*}}...

If no *expr* is given, the variable is initialized to the appropriate value for its data type, usually nil.

with bindings linked by and are performed in parallel; those not linked are performed sequentially. That is,

```
(loop with a = (foo) and b = (bar) and c
      ...)
```

binds the variables like

```
((lambda (a b c) ...)
 (foo) (bar) nil)
```

whereas

```
(loop with a = (foo) with b = (bar a) with c ...)
```

binds the variables like

```
((lambda (a)
    ((lambda (b)
        ((lambda (c) ...)
         nil))
     (bar a)))
 (foo))
```

All *expr*'s in with clauses are evaluated in the order they are written, in lambda expressions surrounding the generated prog. The loop expression

```
(loop with a = xa and b = xb
      with c = xc
      for d = xd then (f d)
        and e = xe then (g e d)
      for p in xp
      with q = xq
      ...)
```

produces the following binding contour, where t1 is a loop-generated temporary:

```
((lambda (a b)
    ((lambda (c)
        ((lambda (d e)
            ((lambda (p t1)
                ((lambda (q) ...)
                 xq))
             nil xp))
         xd xe))
     xc))
 xa xb)
```

Because all expressions in with clauses are evaluated during the variable binding phase, they are best placed near the front of the loop form for stylistic reasons.

For binding more than one variable with no particular initialization, one may use the construct

with *variable-list* {*data-type-list*} {and ...}

as in

```
with (i j k t1 t2) (fixnum fixnum fixnum) ...
```

A slightly shorter way of writing this is

```
with (i j k) fixnum and (t1 t2) ...
```

These are cases of *destructuring* which loop handles specially; destructuring and data type keywords are discussed in sections 17.5 and 17.4.

Occasionally there are various implementational reasons for a variable *not* to be given a local type declaration. If this is necessary, the nodeclare clause may be used:

nodeclare *variable-list*

The variables in *variable-list* are noted by loop as not requiring local type declarations. Consider the following:

```
(declare (special k) (fixnum k))
(defun foo (l)
    (loop for x in l as k fixnum = (f x) ...))
```

If k did not have the fixnum data-type keyword given for it, then loop would bind it to nil, and some compilers would complain. On the other hand, the fixnum keyword also produces a local fixnum declaration for k; since k is special, some compilers will complain (or error out). The solution is to do:

```
(defun foo (l)
    (loop nodeclare (k)
          for x in l as k fixnum = (f x) ...))
```

which tells loop not to make that local declaration. The nodeclare clause must come *before* any reference to the variables so noted. Positioning it incorrectly will cause this clause to not take effect, and may not be diagnosed. It happens that this clause was introduced due to some peculiar behavior of the MULTICS MACLISP compiler, and should not be needed in other implementations.

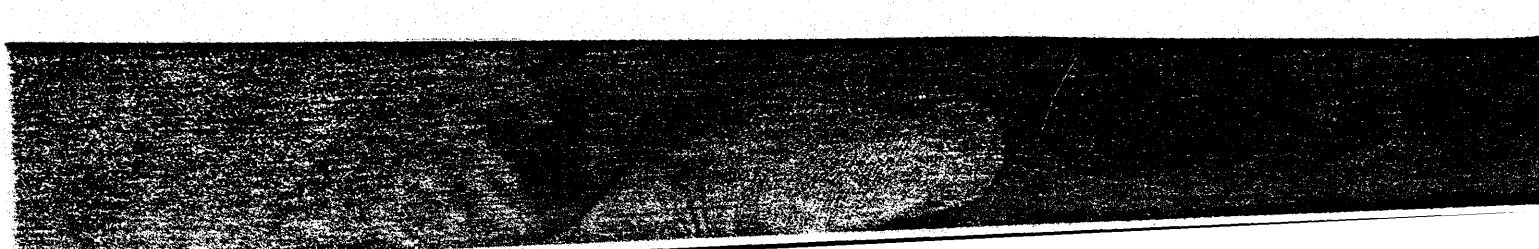
17.2.3 Entrance and Exit

initially *multiple-expression*

This puts *multiple-expression* into the *prologue* of the iteration. It will be evaluated before any other initialization code other than the initial bindings. For the sake of good style, the initially clause should therefore be placed after any with clauses but before the main body of the loop. initially is one of the few loop clauses which are allowed to be followed by multiple expressions; these expressions will be treated as an implicit progn.

finally *multiple-expression*

This puts *multiple-expression* into the *epilogue* of the loop, which is evaluated when the iteration terminates (other than by an explicit return). For stylistic reasons, then, this clause should appear last in the loop body. Note that certain clauses may generate code which terminates the iteration without running the epilogue code; this behavior is noted with those clauses. Most notable of these are those described in the section 17.2.7, Aggregated Boolean Tests. This clause may be used to cause the loop



to return values in a non-standard way:

```
(loop for n in l
      sum n into the-sum
      count t into the-count
      finally (return (quotient the-sum the-count)))
```

Like initially and do, finally may be followed by multiple expressions.

17.2.4 Side Effects

do *multiple-expression*
doing *multiple-expression*

multiple-expression is evaluated each time through the loop, as shown in the print-elements-of-list example on page 144. The do keyword may be followed by multiple expressions.

17.2.5 Values

The following clauses accumulate a return value for the iteration in some manner. The general form is

type-of-collection expr {*data-type*} {into *var*}

where *type-of-collection* is a loop keyword, and *expr* is the thing being "accumulated" somehow. If no into is specified, then the accumulation will be returned when the loop terminates. If there is an into, then when the epilogue of the loop is reached, *var* (a variable automatically bound locally in the loop) will have been set to the accumulated result and may be used by the epilogue code. In this way, a user may accumulate and somehow pass back multiple values from a single loop, or use them during the loop. It is safe to reference these variables during the loop, but they should not be modified until the epilogue code of the loop is reached. For example,

```
(loop for x in list
      collect (foo x) into foo-list
      collect (bar x) into bar-list
      collect (baz x) into baz-list
     finally (return (list foo-list bar-list baz-list)))
```

has the same effect as

```
(do ((g0001 list (cdr g0001))
     (x) (foo-list) (bar-list) (baz-list))
    ((null g0001)
     (list (nreverse foo-list)
           (nreverse bar-list)
           (nreverse baz-list)))
  (setq x (car g0001))
  (setq foo-list (cons (foo x) foo-list))
  (setq bar-list (cons (bar x) bar-list))
  (setq baz-list (cons (baz x) baz-list)))
```

except that loop arranges to form the lists in the correct order, obviating the nreverses at the end, and allowing the lists to be examined during the computation.

collect *expr* {into *var*}

collecting ...

This causes the values of *expr* on each iteration to be collected into a list.

nconc *expr* {into *var*}
nconcing ...
append ...
appending ...

These are like collect, but the results are spliced together.

```
(loop for i from 1 to 3
      nconc (list i (* i i)))
=> (1 1 2 4 3 9)
```

The difference is that, for nconc, the value of *expr* is not copied before being spliced, not that the entire list being accumulated is repeatedly copied during the iteration. This is significant in that one may get ahold of the intermediate result while the iteration is in progress by use of into, and *that* list *does* get destructively modified.

count *expr* {into *var*} {*data-type*}
counting ...

If *expr* evaluates non-nil, a counter is incremented. The *data-type* defaults to fixnum.

sum *expr* {*data-type*} {into *var*}
summing ...

Evaluates *expr* on each iteration, and accumulates the sum of all the values. *data-type* defaults to number, which for all practical purposes is notype. Note that specifying *data-type* implies that *both* the sum and the number being summed (the value of *expr*) will be of that type.

maximize *expr* {*data-type*} {into *var*}
minimize ...

Computes the maximum (or minimum) of *expr* over all iterations. *data-type* defaults to number. Note that if the loop iterates zero times, or if conditionalization prevents the code of this clause from being executed, the result will be meaningless. loop may choose to code the max or min operation itself by just using arithmetic comparison rather than calling max or min, if it deems this to be reasonable based on the particular LISP implementation and the declared type of the accumulation. As with the sum clause, specifying *data-type* implies that both the result of the max or min operation and the value being maximized or minimized will be of that type.

Not only may there be multiple *accumulations* in a loop, but a single *accumulation* may come from multiple places *within the same* loop *form*. Obviously, the types of the collection must be compatible. collect, nconc, and append may all be mixed, as may sum and count, and maximize and minimize. For example,

```
(loop for x in '(a b c) for y in '((1 2) (3 4) (5 6))
      collect x
      append y)
=> (a 1 2 b 3 4 c 5 6)
```

The following computes the average of the entries in the list *list-of-frobs*:

```
(loop for x in list-of-frobs
      count t into count-var
      sum x into sum-var
     finally (return (quotient sum-var count-var)))
```

17.2.6 Endtests

The following clauses may be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to **finally**) to be performed and possibly returning a value (e.g., from **collect**).

while *expr*
: If *expr* evaluates to **nil**, the loop is exited, performing exit code (if any), and returning any accumulated value. The test is placed in the body of the loop where it is written. It may appear between sequential **for** clauses.

until *expr*
: Identical to **while (not** *expr*).

This may be needed, for example, to step through a strange data structure, as in

```
(loop for concept = expr then (superior-concept concept)
      until (top-of-concept-tree? concept)
      ...)
```

The following may also be of use in terminating the iteration:

loop-finish
: (loop-finish) causes the iteration to terminate "normally", the same as implicit termination by an iteration driving clause, or by the use of **while** or **until**—the epilogue code (if any) will be run, and any implicitly collected result will be returned as the value of the **loop**. For example,

```
(loop for x in '(1 2 3 4 5 6)
      collect x
      do (cond ((= x 4) (loop-finish))))
 => (1 2 3 4)
```

This particular example would be better written as **until (= x 4)** in place of the **do** clause. Also, the readability of **loop** constructs suffers from the inclusion of a non-keyword construct which affects the behavior of the iteration because the reader of the code may not be expecting it hidden away in the code. (Stylistically it might be compared with the use of **go**.)

17.2.7 Aggregated Boolean Tests

All of these clauses perform some test, and may immediately terminate the iteration depending on the result of that test.

always *expr*
: Causes the loop to return t if *expr* always evaluates non-null. If *expr* evaluates to nil, the loop immediately returns nil, without running the epilogue code (if any, as specified with the finally clause); otherwise, t will be returned when the loop finishes, after the epilogue code has been run.

never *expr*
: Causes the loop to return t if *expr* never evaluates non-null. This is equivalent to always (not *expr*).

thereis *expr*
: If *expr* evaluates non-nil, then the iteration is terminated and that value is returned, without running the epilogue code.

17.2.8 Conditionalization

These clauses may be used to "conditionalize" the following clause. They may precede any of the side-effecting or value-producing clauses, such as do, collect, always, or return.

when *expr*
if *expr*
: If *expr* evaluates to nil, the following clause will be skipped, otherwise not.

unless *expr*
: This is equivalent to when (not *expr*)).

Multiple conditionalization clauses may appear in sequence. If one test fails, then any following tests in the immediate sequence, and the clause being conditionalized, are skipped. For instance,

```
(loop for x ...
      when (f x)
        unless (g x)
          do ...)
```

is like

```
(loop for x ...
      when (and (f x) (not (g x)))
        do ...)
```

Multiple clauses may be conditionalized under the same test by joining them with and, as in

```
(loop for i from a to b
      when (zerop (remainder i 3))
        collect i and do (print i))
```

which returns a list of all multiples of 3 from a to b (inclusive) and prints them as they are being collected.

If-then-else conditionals may be written using the else keyword, as in

```
(loop for i from a to b
      when (oddp i)
        collect i into odd-numbers
      else collect i into even-numbers)
```

Multiple clauses may appear in an else-phrase, using and to join them in the same way as above. *There is a bug in the handling of multiple* else *clauses. Beware.*

Conditionals may be nested. For example,

```
(loop for i from a to b
      when (zerop (remainder i 3))
        do (print i)
        and when (zerop (remainder i 2))
              collect i)
```

returns a list of all multiples of 6 from a to b, and prints all multiples of 3 from a to b.

When else is used with nested conditionals, the "dangling else" ambiguity is resolved by matching the else with the innermost when not already matched with an else. Here is a complicated example.

```
(loop for x in l
      when (atom x)
        when (memq x *distinguished-symbols*)
          do (process1 x)
        else do (process2 x)
      else when (memq (car x) *special-prefixes*)
             collect (process3 (car x) (cdr x))
             and do (memoize x)
           else do (process4 x))
```

Useful with the conditionalization clauses is the return clause, which causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is,

when *expr1* return *expr2*

is equivalent to

when *expr1* do (return *expr2*)

if the loop is not "named" (by use of the named clause), and to

when *expr1* do (return-from *name expr2*)

if the loop is named *name*. In other words, return arranges to always return from the loop it is a part of.

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test which would cause the iteration to terminate early not to be performed unless the condition succeeds. For example,

```
(loop for x in l
      when (significant-p x)
        do (print x) (princ "is significant.")
        and thereis (extra-special-significant-p x))
```

does not make the extra-special-significant-p check unless the significant-p check succeeds.

The format of a conditionalized clause is typically something like

when *expr1 keyword expr2*

If *expr2* is the keyword it, then a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the composition

when *expr* return it

is equivalent to the clause

thereis *expr*

and one may collect all non-null values in an iteration by saying

when *expression* collect it

If multiple clauses are joined with and, the it keyword may only be used in the first. If multiple whens, unlesses, and/or ifs occur in sequence, the value substituted for it will be that of the last test performed. The it keyword is not recognized in an else-phrase.

17.2.9 Miscellaneous Other Clauses

named *name*

This gives the block (prog) which loop generates a name of *name*, so that one may use the return-from form to return explicitly out of that particular loop:

```
(loop named sue
      ...
      do (loop ... do (return-from sue value) ...)
      ...)
```

The return-from form shown causes *value* to be immediately returned as the value of the outer loop. Only one name may be given to any particular loop construct. This feature does not exist in the MACLISP version of loop, since MACLISP does not support "named progs" or COMMON LISP blocks.

return *expression*

Immediately returns the value of *expression* as the value of the loop, without running the epilogue code. This is most useful with some sort of conditionalization, as discussed in the previous section. Unlike most of the other clauses, return is not considered to "generate body code", so it is allowed to occur between iteration clauses, as in

```
(loop for entry in list
      when (not (numberp entry))
        return (error ...)
      as frob = (times entry 2)
      ...)
```

If one instead desires the loop to have some return value when it finishes normally, one may place a call to return or return-from (as appropriate) in the epilogue (with the finally clause, page 150). return always returns from the loop it is a part of; that is, it turns into a call to return if the loop is not "named", return-from if it is.

17.3 Loop Synonyms

define-loop-macro *keyword*

May be used to make *keyword*, a **loop** keyword (such as **for**), into a Lisp macro which may introduce a **loop** form. For example, after evaluating

```
(define-loop-macro for),
```

one may now write an iteration as

```
(for i from 1 below n do ...)
```

This facility exists primarily for diehard users of a predecessor of **loop**. Its unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

17.4 Data Types

In many of the clause descriptions, an optional *data-type* is shown. A *data-type* in this sense is an atomic symbol, and is recognizable as such by **loop**. These are used for declaration and initialization purposes: for example, in

```
(loop for x in l
      maximize x flonum into the-max
      sum x flonum into the-sum
      ...)
```

the **flonum** data-type keyword for the **maximize** clause says that the result of the max operation, and its "argument" (x), will both be flonums; hence **loop** may choose to code this operation specially since it knows there can be no contagious arithmetic. The **flonum** data-type keyword for the **sum** clause behaves similarly, and in addition causes **the-sum** to be correctly initialized to 0.0 rather than 0. The **flonum** keywords will also cause the variables **the-max** and **the-sum** to be declared to be **flonum**, in implementations where such a declaration exists. In general, a numeric data-type more specific than **number**, whether explicitly specified or defaulted, is considered by **loop** to be license to generate code using type-specific arithmetic functions where reasonable. The following data-type keywords are recognized by **loop** (others may be defined; for that, consult the source code):

fixnum An implementation-dependent limited range integer.

flonum An implementation-dependent limited precision floating point number. Note that NIL interprets **flonum** to mean **double-float**.

small-flonum
This is recognized in the LISP MACHINE LISP implementation only, where its only significance is for initialization purposes, since no such declaration exists.

short-float
single-float
double-float
long-float These are only recognized in the NIL implementation of **loop** currently, although they will probably be recognized in future **loop** implementations in LISP MACHINE LISP which supports numbers of these types (these are the COMMON LISP types).

integer Any integer (no range restriction).

number Any number.

notype Unspecified type (i.e., anything else). This name is inherited from MACLISP.

Note that explicit specification of a non-numeric type for an operation which is numeric (such as the summing clause) may cause a variable to be initialized to nil when it should be 0.

If local data-type declarations must be inhibited, one can use the nodeclare clause, which is described on page 150.

17.5 Destructuring

Destructuring provides one with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure. For example,

```
(loop with (foo . bar) = '(a b c) ...)
```

has the effect of binding foo to a and bar to (b c).

loop's destructuring support is intended to parallel if not augment that provided by the host LISP implementation, with a goal of minimally providing destructuring over list structure patterns. Thus, in Lisp implementations with no system destructuring support at all, one may still use list-structure patterns as loop iteration variables, and in with bindings. In NIL, loop also supports destructuring over vectors (they should be general vectors, i.e., have element-type t).

One may specify the data types of the components of a pattern by using a corresponding pattern of the data type keywords in place of a single data type keyword. This syntax remains unambiguous because wherever a data type keyword is possible, a loop keyword is the only other possibility. Thus, if one wants to do

```
(loop for x in l
      as i fixnum = (car x)
      and j fixnum = (cadr x)
      and k fixnum = (cddr x)
      ...)
```

and no reference to x is needed, one may instead write

```
(loop for (i j . k) (fixnum fixnum . fixnum) in l ...)
```

To allow some abbreviation of the data type pattern, an atomic component of the data type pattern is considered to state that all components of the corresponding part of the variable pattern are of that type. That is, the previous form could be written as

```
(loop for (i j . k) fixnum in l ...)
```

This generality allows binding of multiple typed variables in a reasonably concise manner, as in

```
(loop with (a b c) and (i j k) fixnum ...)
```

which binds a, b, and c to nil and i, j, and k to 0 for use as temporaries during the iteration, and declares i, j, and k to be fixnums for the benefit of the compiler.

```
(defun print-with-commas (list)
   (loop for (item . more?) on list
         do (princ item)
         when more? do (princ ", ")))
```

will generate the output

```
Foo, Bar, Baz
```

when called on the list ("Foo" "Bar" "Baz").

In LISP implementations where loop performs its own destructuring, notably MULTICS MACLISP and LISP MACHINE LISP, one can cause loop to use already provided destructuring support instead:

si:loop-use-system-destructuring? *Variable*

This variable *only* exists in loop implementations in LISPs which do not provide destructuring support in the default environment. It is by default nil. If changed, then loop will behave as it does in LISPs which *do* provide destructuring support: destructuring binding will be performed using let, and destructuring assignment will be performed using desetq. Presumably if one's personalized environment supplies these macros, then one should set this variable to t; there is, however, little (if any) efficiency loss if this is not done.

17.6 The Iteration Framework

This section describes the way loop constructs iterations. It is necessary if you will be writing your own iteration paths, and may be useful in clarifying what loop does with its input.

loop considers the act of *stepping* to have four possible parts. Each iteration-driving clause has some or all of these four parts, which are executed in this order:

pre-step-endtest

This is an endtest which determines if it is safe to step to the next value of the iteration variable.

steps Variables which get "stepped". This is internally manipulated as a list of the form (*var1 val1 var2 val2* ...); all of those variables are stepped in parallel, meaning that all of the *val*s are evaluated before any of the *var*s are set.

post-step-endtest

Sometimes you can't see if you are done until you step to the next value; that is, the endtest is a function of the stepped-to value.

pseudo-steps

Other things which need to be stepped. This is typically used for internal variables which are more conveniently stepped here, or to set up iteration variables which are functions of some internal variable(s) which are actually driving the iteration. This is a list like *steps*, but the variables in it do not get stepped in parallel.

The above alone is actually insufficient in just about all iteration driving clauses which **loop** handles. What is missing is that in most cases the stepping and testing for the first time through the loop is different from that of all other times. So, what **loop** deals with is two four-tuples as above; one for the first iteration, and one for the rest. The first may be thought of as describing code which immediately precedes the loop in the **prog**, and the second as following the body code—in fact, **loop** does just this, but severely perturbs it in order to reduce code duplication. Two lists of forms are constructed in parallel: one is the first-iteration endtests and steps, the other the remaining-iterations endtests and steps. These lists have dummy entries in them so that identical expressions will appear in the same position in both. When **loop** is done parsing all of the clauses, these lists get merged back together such that corresponding identical expressions in both lists are not duplicated unless they are "simple" and it is worth doing.

Thus, one *may* get some duplicated code if one has multiple iterations. Alternatively, **loop** may decide to use and test a flag variable which indicates whether one iteration has been performed. In general, sequential iterations have less overhead than parallel iterations, both from the inherent overhead of stepping multiple variables in parallel, and from the standpoint of potential code duplication.

One other point which must be noted about parallel stepping is that although the user iteration variables are guaranteed to be stepped in parallel, the placement of the endtest for any particular iteration may be either before or after the stepping. A notable case of this is

```
(loop for i from 1 to 3 and dummy = (print 'foo)
      collect i)
  => (1 2 3)
```

but prints **foo** *four* times. Certain other constructs, such as **for** *var* **on**, may or may not do this depending on the particular construction.

This problem also means that it may not be safe to examine an iteration variable in the epilogue of the loop form. As a general rule, if an iteration driving clause implicitly supplies an endtest, then one cannot know the state of the iteration variable when the loop terminates. Although one can guess on the basis of whether the iteration variable itself holds the data upon which the endtest is based, that guess *may* be wrong. Thus,

```
(loop for subl on expr
      ...
      finally (f subl))
```

is incorrect, but

```
(loop as frob = expr while (g frob)
      ...
      finally (f frob))
```

is safe because the endtest is explicitly dissociated from the stepping.

17.7 Iteration Paths

Iteration paths provide a mechanism for user extension of iteration-driving clauses. The interface is constrained so that the definition of a path need not depend on much of the internals of loop. The typical form of an iteration path is

`for` *var* {*data-type*} `being {each|the}` *pathname* {*preposition1 expr1*}...

pathname is an atomic symbol which is defined as a loop path function. The usage and defaulting of *data-type* is up to the path function. Any number of preposition/expression pairs may be present: the prepositions allowable for any particular path are defined by that path. For example,

```
(loop for x being the elements of some-sequence from 1 to 10
      ...)
```

To enhance readability, iteration path names are usually defined in both the singular and plural forms; this particular example could have been written as

```
(loop for x being each element of some-sequence from 1 to 10
      ...)
```

Another format, which is not so generally applicable, is

`for` *var* {*data-type*} `being` *expr0* `and its` *path-name* {*preposition1 expr1*}...

In this format, *var* takes on the value of *expr0* the first time through the loop. Support for this format is usually limited to paths which step through some data structure, such as the "superiors" of something. Thus, we can hypothesize the cdrs path, such that

```
(loop for x being the cdrs of '(a b c . d) collect x)
 => ((b c . d) (c . d) d)
```

but

```
(loop for x being '(a b c . d) and its cdrs collect x)
 => ((a b c . d) (b c . d) (c . d) d)
```

To satisfy the anthropomorphic among you, his, her, or their may be substituted for the its keyword, as may each. Egocentricity is not condoned. Some example uses of iteration paths are shown in section 17.7.1.

Very often, iteration paths step internal variables which the user does not specify, such as an index into some data-structure. Although in most cases the user does not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. loop provides an additional syntax with which one may provide a variable name to be used as an "internal" variable by an iteration path, with the using "prepositional phrase". The using phrase is placed with the other phrases associated with the path, and contains any number of keyword/variable-name pairs:

```
(loop for x being the elements of seq using (index i)
      ...)
```

which says that the variable i should be used to hold the index of the sequence being stepped through. The particular keywords which may be used are defined by the iteration path; the index keyword is recognized by all loop sequence paths (section 17.7.1.2). Note that any individual using phrase applies to only one path; it is parsed along with the "prepositional phrases". It is an error if the path does not call for a variable using that keyword.

By special dispensation, if a *path-name* is not recognized, then the default-loop-path path will be invoked upon a syntactic transformation of the original input. Essentially, the loop fragment

```
for var being frob
```
is taken as if it were
```
for var being default-loop-path in frob
```
and
```
for var being expr and its frob ...
```
is taken as if it were
```
for var being expr and its default-loop-path in frob
```
Thus, this "undefined path-name hook" only works if the default-loop-path path is defined. Obviously, the use of this "hook" is competitive, since only one such hook may be in use, and the potential for syntactic ambiguity exists if *frob* is the name of a defined iteration path. This feature is not for casual use; it is intended for use by large systems which wish to use a special syntax for some feature they provide.

17.7.1 Pre-Defined Paths

loop comes with two pre-defined iteration path functions; one implements a mapatoms-like iteration path facility, and the other is used for defining iteration paths for stepping through sequences.

17.7.1.1 The Interned-Symbols Path

The interned-symbols iteration path is like a mapatoms (or COMMON LISP do-symbols) for loop.
```
(loop for sym being interned-symbols ...)
```
iterates over all of the symbols in the current package and its superiors (or, in Maclisp, the current obarray). This is the same set of symbols which mapatoms iterates over, although not necessarily in the same order. The particular package to look in may be specified as in
```
(loop for sym being the interned-symbols in package ...)
```
which is like giving a second argument to mapatoms.

In LISP implementations with some sort of hierarchical package structure such as LISP MACHINE LISP and NIL, one may restrict the iteration to be over just the package specified and not its superiors, by using the local-interned-symbols path:
```
(loop for sym being the local-interned-symbols  {in package}
      ...)
```

Example:
```
(defun my-apropos (sub-string &optional (pkg package))
    (loop for x being the interned-symbols in pkg
          when (string-search sub-string x)
            when (or (boundp x) (fboundp x) (plist x))
              do (print-interesting-info x)))
```
In the LISP MACHINE LISP and NIL implementations of loop, a package specified with the in preposition may be anything acceptable to the pkg-find-package function. The code generated by this path will contain calls to internal loop functions, with the effect that it will be transparent to changes to the implementation of packages. In the MACLISP implementation, the obarray *must* be an array pointer, *not* a symbol with an array property.

17.7.1.2 Sequence Iteration

One very common form of iteration is that over the elements of some object which is accessible by means of an integer index. loop defines an iteration path function for doing this in a general way, and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

define-loop-sequence-path

(define-loop-sequence-path *path-name-or-names*
fetch-fun size-fun
sequence-type default-var-type)

path-name-or-names is either an atomic path name or list of path names. *fetch-fun* is a function of two arguments: the sequence, and the index of the item to be fetched. (Indexing is assumed to be zero-origined.) *size-fun* is a function of one argument, the sequence; it should return the number of elements in the sequence. *sequence-type* is the name of the data-type of the sequence, and *default-var-type* the name of the data-type of the elements of the sequence. These last two items are optional.

The NIL implementation of loop utilizes the COMMON LISP sequence manipulation primitives to define both element and elements as iteration paths:

```
(define-loop-sequence-path (element elements)
    elt length)
```

Then, the loop clause

```
for var being the elements of sequence
```

will step *var* over the elements of *sequence*, starting from 0. The sequence path function also accepts in as a synonym for of.

The range and stepping of the iteration may be specified with the use of all of the same keywords which are accepted by the loop arithmetic stepper (for *var* from ...); they are by, to, downto, from, downfrom, below, and above, and are interpreted in the same manner. Thus,

```
(loop for var being the elements of sequence
          from 1 by 2
      ...)
```

steps *var* over all of the odd elements of *sequence*, and

```
(loop for var being the elements of sequence
          downto 0
      ...)
```

steps in "reverse" order.

The NIL implementation of loop defines the following additional sequence iteration paths. Basically, they are all special cases of the general elements path, but can generate better code because they may know how to access the sequence better. (If NIL's compiler were smarter, one would be able to get the same effect with type declarations, but that is not the case yet.) Each iteration path name is defined in both the singular and plural form.

vector-elements

This will iterate over any type of vector. There really isn't a good reason to use this over the elements iteration path anymore.

characters
string-elements
This iterates over the characters of a string. **char** is used to reference them, hence the iteration variable defaults to type **string-char**.

bits
bit-vector-elements
This iterates over the bits in a bit-vector; the iteration variable is declared to be **bit**, as it can take on only **0** or **1** as values.

The following are the special cases which may be of somewhat more interest, because they can generate much better code knowing that the sequence is of a particular simple type:

simple-vector-elements
The sequence must be a simple vector.

simple-string-elements
The sequence must be a simple string.

simple-bit-vector-elements
The sequence must be a simple bit vector.

All such sequence iteration paths allow one to specify the variable to be used as the index variable, by use of the **index** keyword with the **using** prepositional phrase, as described (with an example) on page 161.

17.7.2 Defining Paths

This section and the next may not be of interest to those not interested in defining their own iteration paths.

A **loop** iteration clause (e.g. a **for** or **as** clause) produces, in addition to the code which defines the iteration (section 17.6), variables which must be bound, and pre-iteration (*prologue*) code. This breakdown allows a user-interface to **loop** which does not have to depend on or know about the internals of **loop**. To complete this separation, the iteration path mechanism parses the clause before giving it to the user function which will return those items. A function to generate code for a path may be declared to **loop** with the **define-loop-path** function:

define-loop-path
(define-loop-path *path-name-or-names path-function*
list-of-allowable-prepositions
datum-1 datum-2 ...)
This defines *path-function* to be the handler for the path(s) *path-name-or-names*, which may be either a symbol or a list of symbols. Such a handler should follow the conventions described below. The *datum-i* are optional; they are passed in to *path-function* as a list.

The handler will be called with the following arguments:

path-name
The name of the path which caused the path function to be invoked.

variable
The "iteration variable".

data-type
The data type supplied with the iteration variable, or nil if none was supplied.

prepositional-phrases
This is a list with entries of the form *(preposition expression)*, in the order in which they were collected. This may also include some supplied implicitly (e.g. an of phrase when the iteration is inclusive, and an in phrase for the default-loop-path path); the ordering will show the order of evaluation which should be followed for the expressions.

inclusive?
This is t if *variable* should have the starting point of the path as its value on the first iteration (by virtue of being specified with syntax like for *var* being *expr* and its *pathname*), nil otherwise. When t, *expr* will appear in *prepositional-phrases* with the of preposition; for example, for x being foo and its cdrs gets *prepositional-phrases* of ((of foo)).

allowed-prepositions
This is the list of allowable prepositions declared for the pathname that caused the path function to be invoked. It and *data* (immediately below) may be used by the path function such that a single function may handle similar paths.

data This is the list of "data" declared for the pathname that caused the path function to be invoked. It may, for instance, contain a canonicalized pathname, or a set of functions or flags to aid the path function in determining what to do. In this way, the same path function may be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings
This is a list of variables which need to be bound. The entries in it may be of the form *variable*, (*variable expression*), or (*variable expression data-type*). Note that it is the responsibility of the handler to make sure the iteration variable gets bound. All of these variables will be bound in parallel; if initialization of one depends on others, it should be done with a setq in the *prologue-forms*. Returning only the variable without any initialization expression is not allowed if the variable is a destructuring pattern.

prologue-forms
This is a list of forms which should be included in the loop prologue.

the four items of the iteration specification
These are the four items described in section 17.6, page 159: *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*.

another four items of iteration specification
If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to *all* iterations.

Here are the routines which are used by loop to compare keywords for equality. In all cases, a *token* may be any LISP object, but a *keyword* is expected to be an atomic symbol. In certain implementations these functions may be implemented as macros.

si:loop-tequal *token keyword*
This is the loop token comparison function. *token* is any Lisp object; *keyword* is the keyword it is to be compared against. It returns t if they represent the same token, comparing in a manner appropriate for the implementation.

si:loop-tmember *token keyword-list*
The member variant of si:loop-tequal.

si:loop-tassoc *token keyword-alist*
The assoc variant of si:loop-tequal.

The following macro turns into the "proper" code for generating a temporary variable in a particular LISP implementation.

si:loop-gentemp &optional *prefix*
This expands into a call to the proper function for constructing a temporary variable. Depending on how it expands, the form *prefix* may not get used (so it should not have interesting side effects!). In NIL (and eventually in some other implementations), this utilizes the gentemp function so that the generated variable will be interned and somewhat mnemonic: in this case, *prefix* (which should be a symbol or a string) will be used as a prefix of the generated symbol. In other implementations, si:loop-gentemp will expand to just a call to gensym, and throw away *prefix*.

If gentemp does get used but *prefix* is not supplied, then the prefix loopvar- is used so that the variable is identifiable as originating with loop.

If an iteration path function desires to make an internal variable accessible to the user, it should call the following function instead of si:loop-gentemp:

si:loop-named-variable *keyword*
This should only be called from within an iteration path function. If *keyword* has been specified in a using phrase for this path, the corresponding variable is returned; otherwise, si:loop-gentemp is called and that new symbol returned. Within a given path function, this routine should only be called once for any given keyword.

If the user specifies a using preposition containing any keywords for which the path function does not call si:loop-named-variable, loop will inform the user of his error.

17.7.2.1 An Example Path Definition

Here is an example function which defines the string-characters iteration path. This path steps a variable through all of the characters of a string. It accepts the format

```
(loop for var being the string-characters of str ...)
```

The function is defined to handle the path by

```
(define-loop-path string-characters string-chars-path
      (of))
```

Here is the function:

```
(defun string-chars-path (path-name variable data-type
                          prep-phrases inclusive?
                          allowed-prepositions data
                          &aux (bindings nil)
                               (prologue nil)
                               string-var
                               index-var
                               size-var)
  (declare (ignore allowed-prepositions data))
  (setq string-var (si:loop-gentemp 'loop-string-)
        index-var (si:loop-gentemp 'loop-index-)
        size-var (si:loop-gentemp 'loop-size-))
  ; To iterate over the characters of a string, we need
  ; to save the string, save the size of the string,
  ; step an index variable through that range, setting
  ; the user's variable to the character at that index.
  ; Default the data-type of the user's variable:
  (cond ((null data-type) (setq data-type string-char)))
  ; We support exactly one "preposition", which is
  ; required, so this check suffices:
  (cond ((null prep-phrases)
         (ferror nil "OF missing in ~S iteration path of ~S"
                 path-name variable)))
  ; We do not support "inclusive" iteration:
  (cond ((not (null inclusive?))
         (ferror nil
           "Inclusive stepping not supported in ~S path ~
            of ~S (prep phrases = ~:S)"
           path-name variable prep-phrases)))
  ; Set up the bindings
  (setq bindings (list (list variable nil data-type)
                       (list string-var (cadar prep-phrases))
                       (list index-var 0 'fixnum)
                       (list size-var 0 'fixnum)))
  ; Now set the size variable
  (setq prologue (list '(setq ,size-var (string-length
                                          ,string-var))))
  ; and return the appropriate stuff, explained below.
  (list bindings
        prologue
        '(= ,index-var ,size-var)
        nil
        nil
        (list variable '(char ,string-var ,index-var)
              index-var '(1+ ,index-var))))
```

The first element of the returned list is the bindings. The second is a list of forms to be placed in the *prologue*. The remaining elements specify how the iteration is to be performed. This example is a particularly simple case, for two reasons: the actual "variable of iteration", index-var, is purely internal (being gensymmed), and the stepping of it (1+) is such that it may be performed safely without an endtest. Thus index-var may be stepped immediately after the setting of the user's variable, causing the iteration specification for the first iteration to be identical to the iteration specification for all remaining iterations. This is advantageous from the standpoint of the optimizations loop is able to perform, although it is frequently not possible due to the semantics of the iteration (e.g., for *var* first *expr1* then *expr2*) or to subtleties of the stepping. It is safe for this path to step the user's variable in the *pseudo-steps* (the fourth item of an iteration specification) rather than the "real" steps (the second), because the step value can have no dependencies on any other (user) iteration variables. Using the pseudo-steps generally results in some efficiency gains.

If one desired the index variable in the above definition to be user-accessible through the using phrase feature with the index keyword, the function would need to be changed in two ways. First, index-var should be set to (si:loop-named-variable 'index) instead of the call to si:loop-gentemp. Secondly, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be

```
(list bindings prologue
      nil
      (list index-var '(1+ ,index-var))
      '(= ,index-var ,size-var)
      (list variable '(char ,string-var ,index-var))
      nil
      nil
      '(= ,index-var ,size-var)
      (list variable '(char ,string-var ,index-var)))
```

Note that although the second '(= ,index-var ,size-var) could have been placed earlier (where the second nil is), it is best for it to match up with the equivalent test in the first iteration specification grouping.

18. The Flavor Facility

18.1 Introduction

Languages such as Smalltalk and Act-1 are designed to encourage a style of programming called *object-oriented* programming. LISP MACHINE LISP offers a facility for object-oriented programming as well; it is called the *Flavor System*, or just flavors. NIL offers a more primitive version of flavors than is available on the Lisp Machine, but unless you do quite complicated things with flavors, you will probably never notice the difference.

18.1.1 Object-oriented Programming

Suppose you were writing a file system. You might have several different types of files, including, for example, binary files and text files. If you wrote a program to print files on a user's terminal, and you wanted it to print ASCII characters when the user printed a text file, but octal numbers when the user printed a binary file, you might implement it as follows:

```
(typecase file
   (binary (octally-print file))
   (text (ascii-print file)))
```

That is, you might dispatch off the type of the file, calling the appropriate function to print the file. It might be nicer, however, to keep the information about how the file should be printed *with the file itself*. That is, the method used for printing itself could be part of the information contained in each file; we could simply decide that every type of file we will support in our operating system will know how to perform certain operations, and we could specify printing on a terminal to be one of them. Then we would implement the above as

```
(send file :print-contents)
```

where :print-contents is the name of a method that could be specified for each type of file.

On simply looking at the differences between the two samples of code, one might notice that the second expresses much more clearly and compactly what we are doing: printing the file. We trust whoever defined this type of file object to have defined a reasonable :print-contents method for it, and we don't worry any further about type-dispatching and the like. Thus object-oriented programming constructs can have the effect of freeing the programmer from an extra level of detail.

This should sound familiar even to users who have not used flavors, because it is similar to generic arithmetic in COMMON LISP. In fact, operations that work for more than one type of object (like the imaginary :print-contents above) are called *generic operations*.

Another thing we might notice about the object-oriented way of doing things is described by its name. We might say somewhat fancifully that the files in our example above have been raised from the realm of "inanimate data" to being objects that can *do* things. A file has become an object that knows how to print itself, and can be asked to do so.

Objects can know of things besides their methods for performing operations, and this brings up another advantage of object-oriented programming, which is useful even when one is not planning on implementing operations that will work on a large class of objects. Two objects of the same type will share the same methods for performing an operation. But two objects of the same type can still have distinct state. They can have instance variables: variables that are local to each instantiation, in much the same way that scoped variables are local to a function call. For example, the file objects we were discussing above could have variables :author and :write-date, and each object would have its own value for these variables.

18.1.2 Object-oriented Programming Using Flavors

When we use flavors to write object-oriented code, the objects themselves are not flavors. They are *instantiations* of flavors. The flavor of an object is actually its type. We define a flavor using defflavor. The definition of a flavor looks like

```
(defflavor flavor-name instance-variables
  component-flavors
  option1 option2... )
```

The *flavor-name* can be any symbol, the *instance-variables* a (possibly null) list of symbols (variables) and their initial values, the *component-flavors* a list of flavors, and the *option*s will be described further below. A more concrete example is:

```
(defflavor bicycle ((wheel-size nil) (gear-ratios nil)
                    (selected-gear nil)
                    (distance-travelled nil))
  ()
  :gettable-instance-variables
  :settable-instance-variables)
```

The option :gettable-instance-variables will cause a method that will return the value of that instance variable to be defined for each of the instance variables. :settable-instance-variables will cause a method that will allow us to set the value of that instance variable to be generated for each instance variable.

If we want to create an instantiation of the flavor bicycle, we use make-instance:

```
(setq my-bike (make-instance 'bicycle))
```

returns

```
#<BICYCLE 1287B8>
```

or something like it. This object can be described:

```
(describe my-bike)

The instance at address 1287B8 is of flavor BICYCLE and is 4
Q's long.  It directly or indirectly includes flavors (BICYCLE
VANILLA-FLAVOR), and is of the types (BICYCLE VANILLA-FLAVOR).
The 4 instance variables are:
  WHEEL-SIZE  NIL
  GEAR-RATIOS  NIL
  SELECTED-GEAR  NIL
  DISTANCE-TRAVELLED  NIL
NIL
```

We can cause an instantiation of a flavor to execute a method with send. The methods created upon definition of a flavor with the option :settable-instance-variables have the names of the instance variables appended to "set-", but are in the keyword package. Thus we could set the value of wheel-size like this:

```
(send my-bike :set-wheel-size 27)
```

The methods created on definition of a flavor with the option :settable-instance-variables are the same as the names of the variables, but are in the keyword package. So we could get the value of :wheel-size like this:

```
(send my-bike :wheel-size)
27
```

We can define methods for the flavor we've created with the function defmethod.

send *object message* &rest *args*

This is the basic message-passing primitive. It should be used instead of funcall, which has been used in the past in LISP MACHINE LISP.

lexpr-send *object message* &rest *args*

This is to send as lexpr-funcall (which is apply (page 31), as it is defined now) is to funcall.

There must be at least one *arg* given, and the last one must be either a list or a vector. The object is sent *message* with arguments of all of the other *args* followed by all the elements of the last *arg*.

send-forward *object message* {*arg*}*

This is only valid within the lexical scope of a defmethod definition.

Let the flavor which this method was defined on be called *flav*. send-forward then does a send, but starts searching for methods to handle *message* after *flav*.

n.b. send-forward is neither as efficient as it should be nor as efficient as one would like, yet. Note that it can be used to acheive many of the same effects as method combination.

lexpr-send-forward *object message* {*arg*}*
Like lexpr-send, but does send-forward.

make-instance *flavor-name* &rest *keyworded-arguments*
This is the primary instantiation function. The *keyworded-arguments* are alternating keywords and values. Typically, they specify initial values for the instance variables which are initable (as specified with the :initable-instance-variables option to defflavor). They may also be arbitrary keywords which are checked for validity against those specified with the :init-keywords option to defflavor, which (merged with the :init-plist specification to defflavor) will be passed as arguments to the :init method of the flavor.

defflavor *flavor-name instance-variables included-flavors options...*

flavor-name is the name of the flavor being defined. After it is defined, it is acceptable as a second argument to typep (page 18), which will return t if given a second argument of *flavor-name* and a first argument of an instantiation of *flavor-name*, or any other flavor which directly or indirectly includes *flavor-name*.

instance-variables is a list of instance variables or lists of instance variables and forms to eval to obtain their initial values. These are not necessarily all of the instance variables of the instance; some may be inherited from other flavors which *flavor-name* is being built from. However, compiled flavor methods for *flavor-name* may not know about those inherited instance variables, so if you "know" that a flavor is going to have certain variables and need to use them, you should include them here. (Note that in the current NIL instance variable inheritance is performed when the defflavor form is compiled, so one will not receive a diagnostic about this. The inheritance will be deferred in in some later release, however, to provide for other features, including the ability to not have the component flavors of *flavor-name* defined when the defflavor is being compiled or interpreted.)

If an atomic instance variable is specified in the instance variables list, then the instance variable is initialized "unbound" by make-instance (assuming no initialization was specified with make-instance). This will cause an unbound variable reference in the NIL interpreter; external ("outside accessible") references will just pick out the unbound pointer, as will compiled references to the instance variable, and probably cause some spastic behavior later.

Otherwise, the instance variable specification should be a list of the instance variable and an initialization form, which will be evaluated to determine the initial value. (NIL actually does not use eval, but stores the value either as a constant, if it self-evaluates, or as a function of no arguments which evaluates the initialization form; this function will be compiled when the defflavor form is compiled.)

The following defflavor options all deal with instance variables which must be listed in the *instance-variables* given for the defflavor. They may appear as atomic options, like :gettable-instance-variables and :settable-instance-variables in the bicycle example (page 171), in which case they refer to all of the *instance-variables* of the defflavor, or listed with those they pertain to, as in

```
(defflavor frob (var-1 var-2 var-3) ()
   :gettable-instance-variables
   (:settable-instance-variables var-1 var-2))
```

in which var-1, var-2, and var-3 are all :gettable, but only var-1 and var-2 are :settable.

:gettable-instance-variables
Causes automatic generation of methods which will fetch the values of the specified instance variables. Each method name is the name of the variable interned in the keyword package. Thus in the bicycle example, one may send a bicycle the :distance-travelled message to find how far the bicycle has traveled.

:settable-instance-variables
Causes automatic generation of methods which will replace the values of the specified instance variables. Each method takes exactly one argument, the new value. The method name will be the concatentation of "SET-" and the instance variable name, interned in the keyword package.

:initable-instance-variables
This specifies which instance variables may be trivially initialized by make-instance (and instantiate-flavor). For those which may be, it is done by specifying a keyword which is the instance variable name interned in the keyword package followed by the value. For example,

```
(make-instance 'bicycle :wheel-size 26)
```

:outside-accessible-instance-variables
This causes automatic generation of macros which access the specified instance variables without sending messages. In principle this is more efficient than sending the message; it, of course, requires that the instance have such instance variables. This is most useful when the instance variables are ordered (see below); otherwise, some lookup has to be performed.

:ordered-instance-variables
The instance variables will be ordered in the instance in exactly the order they are listed here, starting from slot 0. This can be done to allow super-fast external accessing, or simply because other low-level code (like VMS assembly language routines) needs to be able to understand the structure.

:special-instance-variables
don't work.

:functional-instance-variables
no workee.

Other options. Many of the instantiation-time checks are not performed, and some are sort of meaningless in the current implementation. This is because this implementation performs all inheritance computations at eval or compile time.

:required-flavors
The flavors listed are required to be included in any flavor which includes this one. make-instance is supposed to barf if that is not the case.

:required-instance-variables
The instance variables listed are required to be defined by any flavor which includes this

one, and make-instance is supposed to barf if that is not the case.

:required-methods
Any flavor including this flavor is required to support the listed methods. This is allegedly checked at instantiation time.

:no-vanilla-flavor
Do not include vanilla-flavor, as is done by default.

:included-flavors
Sort of like building the flavor from the named flavors, but they are made to come last always. *where is the inheritance-order and vanilla-flavor insertion and this explained?*

:flavor-not-instantiable
This flavor is not itself instantiable. This should be specified for things which are not complete in themselves, but *mixin flavors*—flavors which are meant to be mixed in to provide some aspect of other flavors.

:init-keywords
Allowable keywords which make-instance will pass along to the :init message when a flavor is instantiated.

:default-init-plist
Alternating keyword-values, which are supplied to the :init message when a flavor is instantiated, unless the keyword was supplied already to make-instance.

:documentation
ummmm

defmethod (*flavor-name message-name* [*message-type*]) *arglist body...*
Defines a method *message-name* for *flavor*. *message-type* is not supported, do not use it. *arglist* is any lambda-list acceptable to NIL. *self* will be bound (lexically) for the evaluation of *body*.

Lexical instance variables are correctly enclosed by the NIL interpreter in this version of NIL. The only time this can fail is if there is any funny stuff with how the definition is being performed, like evaluating a defmethod inside the lexical environment of another defmethod or a defun. This would not work compiled anyway.

defmethod-primitive (*flavor-name message-name*) *arglist body...*
This is used to define a method *without* interfacing to deal with the self variable or the instance variables. The arguments which the generated function receives will be the object, the map vector, the message, and the other arguments. This routine exists primarily for primitive low-level method-generation code, as that which might be used by defstruct.

18.2 System-Defined Messages

Here are some of the messages the system uses to deal with objects defined by **defflavor**, and what they mean.

:print-self *stream level slashify-p*

The object should print itself to the stream *stream*. *level* is the recursion level of printing, and should be compared against the dynamic value of **prinlevel**. *slashify-p* being non-null means that the output should maybe be re-readable; it is being done by **prin1** rather than **princ**.

If you use this in a non-trivial fashion (specifically, if the object will be printed in a non-atomic fashion), then it might be reasonable to define methods for the pretty-printer using the :pp-dispatch and :pp-anaphor-dispatch methods, and define the non-pretty-printing :print-self method in terms of how the pretty-printing is performed. This is described in [7].

:equal *other-object*

The method should return **t** if its object is **equal** to *other-object*, **nil** if it is not. *other-object* will be of the exact same type as the object receiving the message (a consequence of the formal definition of **equal**, page 20). This message is also used by **eql** (page 20) on numerical types which are not "primitive" NIL data types.

:sxhash

The object should return a hash encoding of itself, such that two objects which are **equal** have the same hash. See the description of **sxhash**, page 119, for the semantics which must be enforced, and note also the default **:sxhash** method, page 177.

:eval

Allows extending the evaluator in strange and wondrous ways to handle evaluation of non-list forms. Note that certain types which are defined to self-evaluate do so by special case checks in the interpreter, so one cannot change the evaluation behaviour of those types. The compiler can't handle these extensions however.

:funcall *argument-vector*

This is what happens by default when a **funcall** is performed on an instance. *argument-vector* is a stack vector (section 3.1, page 15) of the arguments.

:describe *?arguments?*

This is what is used by the **describe** function (page 222).

:exhibit-self *stream*
:select-nth *n*
:store-nth *n value*

These are used by the **exhibit** function (page 222) to define how exhibition is performed on objects of the given type. Basically, exhibition is initiated by sending the object a :exhibit-self message; it should respond by printing out the appropriate information, and returning the number of "slots" or "indices" which it includes. (Try exhibiting various NIL objects to see the format; do *not* include the clear-screen in the display. The indices

printed out in the initial display are printed by this method.) Then, the object will be sent (as the user interacts) :select-nth and :store-nth messages to select and store the corresponding components. Generally, there is no need to define such a method for ordinary flavors, as the method inherited from vanilla-flavor will show the instance variable names etc.

:pp-dispatch *format-description?*
:pp-anaphor-dispatch

These are used by the NIL pretty-printer [7]. :pp-dispatch is used to control formatting; to use this you will need to consult the pretty-printer documentation. :pp-anaphor-dispatch is used to detect circularities in the structure being printed; all that is normally needed is to call the function pp-anaphor-dispatch on each of the components which will be printed by the :pp-dispatch method. These methods should be defined in pairs, so that they refer to the same set of components.

18.3 Message Defaults

Here are some of the messages provided by vanilla-flavor, and what they do.

:print-self *stream level slashify-p*

Prints something vaguely informative.

:get-handler-for *message-name*

Returns the handler function for the message *message-name*, or nil. In NIL, this is necessarily of type compiled-function.

:operation-handled-p *message-name*

Returns a non-null value if the object supports a message *message-name*, nil otherwise.

:send-if-handles *message* &rest *args*

If the object supports *message*, then it is sent that message with arguments of whatever *args* were passed; otherwise, nil is returned.

:which-operations

A list of all of the messages which the object handles is returned. This is computed dynamically and cached on a per-flavor basis.

:equal

By default, two objects are equal only if they are eq. If the object has interesting criterial components, it must define an equal message to compare them.

:sxhash

The default :sxhash simply returns a hash computation on the name of the flavor. The reason for this is that if two objects are equal, their sxhashes *must* be equal. So, if the object does anything interesting for the :equal message, it should probably define a compatible :sxhash message so that different objects will hash differently.

:exhibit-self *stream*
:select-nth *n*
:store-nth *n value*

The default exhibition method displays the all of the instance variables of the instance, and their values. The select and store methods just allow one to fetch and modify the variables by index.

:pp-dispatch *format-description?*

The default :pp-dispatch method pretty-prints the object the way it prints (via :print-self), and treats it as atomic. If you define a :print-self method for something, the :pp-dispatch method may not function as desired, in that it will not do any formatting of the components.

:pp-anaphor-dispatch

The default :pp-anaphor-dispatch method does nothing, on the grounds that the :pp-dispatch method will not be printing any components.

19. Input, Output, and Streams

Input and output in NIL is performed by operations on *streams*. Some streams can operate in only one direction (input or output), and some can operate in both.

streamp *x*

Returns t if *x* is a stream, nil otherwise.

Most operations on streams are performed by functions which take the stream as one of its arguments, possibly defaulted. Although ultimately the stream operations turn into message-passing using the flavor system, these functions are the perferred way to do things, as they perform what mediation might be necessary between the desired effect and the stream's capabilities.

19.1 Standard Streams

The following variables have as their values streams used for various purposes. In the future, the names will be changed to have * characters at both ends; e.g., standard-input will become *standard-input*.

standard-input *Variable*

This is used as the default stream for various input functions, and for the toplevel and breaklevel loops.

standard-output *Variable*

This is used as the default stream for various output functions, and for the toplevel and breaklevel loops.

terminal-io *Variable*

The value of terminal-io is ordinarily the stream which connects to the user's console.

error-output *Variable*

This is the stream to which error messages should be sent. Normally, it directs output through the value of terminal-io (but see comments below), but it could be made to send them to a file, for instance. (This may not be used properly yet.)

query-io *Variable*

This stream is used to ask questions of the user. Normally it uses the terminal, but could be made to (for instance) log both the input and the output of the transactions.

trace-output *Variable*

This is the stream to which output from tracing (see the trace function, page 220) is sent.

All of the above streams, with the exception of terminal-io, are initially bound to synonym streams which pass all operations on to the stream which is the value of terminal-io.

The value of terminal-io should not normally be changed; to change where various input and output is sent, the appropriate other stream(s) should be modified. There are occasions when it might be reasonable to change the value of terminal-io, however, which is why the other streams are supposed to indirect through the value of it: fancy graphics or window hacking might necessitate making a completely new stream for it. This type of thing will be dealt with in some later version of this document.

NIL additionally defines the following streams, which should probably be flushed, or at least renamed with something more in line with the above variables.

msgfiles *Variable*

This is used for random kinds of message printout which will not require interaction on the part of the user. The compiler, for instance, prints its notifications here.

si:debug-input *Variable*
si:debug-output *Variable*

These are the input and output streams used by the current interactive debugger. When the next debugger is in use, it will be using some combination of error-output and query-io.

19.2 Stream Creation and Operations

open *what* &rest *keyworded-arguments*

The open function is the function used for creating streams which interface to I/O devices in NIL. It is likely that this will change in the future, such that each specific type of "opening" has its own specialized function (e.g., for "files", "terminals", possibly other devices), in which case open will be for "files".

First, *keyworded-arguments* is put into a canonicalized form. Essentially, open is considered to take alternating keyword/value arguments. However, for MACLISP compatibility, if open is given exactly two arguments, the second is interpreted as either a single keyword, or a list of single keywords, which are mapped specially into the standard open keyword arguments. Thus, in NIL,

```
(open pathname 'out)
```

opens *pathname* as a standard buffered ascii output file, and

```
(open pathname)
```

opens *pathname* as an ordinary buffered ascii input file.

open attempts to determine the way in which to actually perform the open by looking at the options. I am being very vague about this because it is going to change somewhat, but hopefully will remain upwards compatible. If there is a :type keyword, then the argument to that is used to tell open what type of open is being performed. The interesting ones right now are

:dsk

which says that *what* should be interpreted as a pathname, and the open will refer to a file in some filesystem. The specifics of this for VMS are discussed later in (page 202).

:tty
which says that *what* is the name of a terminal (it may be nil, meaning use the logical name TT), or a string with or without a : terminating it, or a pathname in which the device is used), and the open should behave accordingly. This tty may actually be quite useless, and you probably want instead

:display-tty
which is like :tty but sets things up so that cursorpos will work on it. This is discussed more thoroughly in section 19.9, page 209.

If no :type is specified and *what* is a stream, it is sent the :open message with arguments *keyworded-arguments*. The normal use of this is to re-open a stream which has been closed, and in this case no arguments are normally needed (and often are illegal). Not all streams necessarily support this, but all currently defined NIL streams which might be returned by open and to which close is meaningful, do. Many streams which might be this support the :set-pathname message, which is the primitive form of the MACLISP cnamef function; this allows changing the pathname which will be opened when a closed stream is reopened.

with-open-file (*var what* &rest *gubble*) *body*...
This binds *var* to the result of opening *what* with options *gubble*, and executes *body* in that environment. When the *body* is exited, the file is closed. (You cannot fool with-open-file by setqing *var*.) If the form is exited abnormally, by an error, quit, or •throw, the file is closed in :abort mode; for a freshly written output file, this means it is deleted.

The behaviour of with-open-file with respect to treatment of *what* and *gubble*, and to errors in opening, is identical to open otherwise.

with-open-file should be used wherever that scoping is reasonable, so that stray open files are not left around by buggy programs. There is also potential for it to be somewhat more storage efficient due to use of resources, since the extent of the created stream is known to be dynamic (it is not valid to pass it back outside of the with-open-file form). This is not done yet, but will be someday.

close *stream* &optional *abort-flag*
Closes *stream*.

You may close an already closed stream; close will return nil if the stream is already closed (or does not support closing), t otherwise.

If *abort-flag* is not nil, then (in principal) this is an error close, as perhaps performed by abnormal exit from a with-open-file form. For an output file, this *might* mean that the file gets deleted. This is done with ordinary "disk" type output streams.

abort-flag will probably be changed to be a keyworded argument in the future.

make-synonym-stream *symbol*

This makes a *synonym stream*. Such a stream directs (most) operations on it to the current dynamic binding of the variable *symbol*. In this way, the stream produced can always be indirecting to another stream, even when the value of *symbol* changes by its being bound or setqed.

make-string-output-stream &optional &key (:line-length 79) (:line-number 1) (:page-length 60) (:page-number 1) (:character-position 0)

This creates a stream which will accumulate all output given to it. This output may be obtained as a string by get-output-stream-string, below.

The *options* are used to initialize various parameters of the stream, so that formatting may be performed to it. By special dispensation to COMMON LISP, if make-string-output-stream is given exactly one argument, that is the line length.

get-output-stream-string *string-output-stream*

string-output-stream should be a stream created by make-string-output-stream. This returns all of the text accumulated since the last call to get-output-stream-string on this stream, or the stream's creation, as a string.

with-output-to-string (*var* . *options*) *body...*

This binds *var* to a stream which will accumulate all output sent to it as a string, which will be returned when with-output-to-string returns. The *options* which the stream may be created with are passed directly to make-string-output-stream, q.v. The stream so created has only *dynamic* extent; it is allocated as a resource, and deallocated on exit from with-output-to-string. As such, with-output-to-string can be more efficient than calling make-string-output-stream and get-string-output-stream-string yourself.

make-string-input-stream *string* &optional &key (:start 0) :end

This returns a stream which, when read from, will produce the characters of *string* from *start* to *end* (defaultly the end of the string). The behaviour of the stream is undefined if *string* is modified during the reading.

with-input-from-string (*var string* . *options*) *body...*

This evaluates *body* in an environment in which *var* is bound to a stream created by make-string-input-stream with a string of *string* and extra options *options*.

The stream so created, however, has only *dynamic* extent. The stream is allocated on entry and deallocated on exit for later reuse, so with-input-from-string can be more efficient than doing this yourself.

19.3 Input Functions

First some functions not specific to ascii input streams (necessarily). listen and clear-input could conceivably be meaningful on strange peripheral devices (dreamer, aren't i?).

listen &optional *input-stream*
This will return nil if there is no input immediately available from *input-sream*, non-null otherwise. On a terminal, the intent is that it tells whether the user has typed some input which has not been read yet. On non-interactive streams it should be true except at end-of-file; most streams probably don't support it yet.

clear-input &optional *input-stream*
Flushes buffered input from *input-stream*. This only works on the terminal right now. (It isn't really meaningful for non-interactive streams.)

19.3.1 Ascii Input

Most of the functions which read input take arguments *input-stream* and *eof-value*. In general, if *input-stream* is nil or not supplied, it defaults to the value of standard-input; if it is the atom t, the value of terminal-io will be used.

If no *eof-value* is specified, then an error will be signalled at end-of-file, otherwise the *eof-value* will be returned. Specifying an *eof-value* of nil is *not* equivalent to specifying no *eof-value*.

When input is read from an interactive stream, the characters typed will be echoed at the user. For those functions which do some significant amount of reading, such as readline or read, rubout processing will be provided. In this case, specifying an *eof-value* means that if the user attempts to "rub out" past the beginning of what he was typing, the function will return *eof-value*, instead of requiring him to type a complete expression (line, s-expression, whatever the function calls for).

What actually happens right now is that specifying an *eof-value* when reading from an interactive stream, dies.

read-char &optional *input-stream eof-value*
Reads one character from *input-stream*.

This doesn't seem to take *eof-value* yet?

peek-char &optional *input-stream eof-value*
This definition is wrong. The arguments should be *peek-type*, *input-stream*, *eof-value*. It will eventually be fixed.

Peek at a character in the input stream. Like read-char, but the next call to read-char will return the same character.

unread-char *character* &optional *input-stream*
Undoes a read-char. peek-char, in the simple case, could have been (sometimes, is) defined as being a read-char followed by an unread-char of the character just read.

Input streams are only required to support the ability to back up one character: multiple unread-chars without intervening read-chars are an error.

readline &optional *input-stream eof-value*
Reads a line of text from *input-stream* and returns it, as a string. A second value is returned, which is t if end-of-file was reached, nil otherwise.

read &optional *input-stream eof-value*
Reads one s-expression from *input-stream*, and returns it. Reading and reader syntax is discussed in section 16.3, page 126.

19.3.2 Binary Input

The semantics of binary input are stream specific. In general, integers of some significance are read, and NIL places no special interpretation on any particular values. The only sort of binary input NIL supports, however, only reads unsigned eight-bit bytes from disk files.

read-byte *input-stream* &optional *eof-value*
Reads one byte from *input-stream* and returns it as an integer, unless end of file is reached, in which case the normal end-of-file behaviour occurs.

19.4 Output Functions

Similar to the input functions, if an optional *output-stream* argument is not supplied to an output function, it defaults to the value of standard-output.

First some functions applicable to both ascii and binary streams.

force-output *output-stream*
The purpose of force-output is to ensure that no output which may have been produced is sitting around in anyone's buffers. If *output-stream* is buffered by NIL, the output should be sent to the operating system (or whatever), and if necessary, the operating system told to send the contents of its buffers off to their eventual destination.

In practice this doesn't do anything yet in NIL.

finish-output *output-stream*
This is like force-output, and additionally does not return until the output has actually reached its destination.

If a stream does not handle this, which no currently implemented NIL streams do, a force-output is done, q.v.

clear-output *output-stream*
The purpose of this is to cause as *little* as possible of any output already sent to *output-stream* to reach its destination; just as **force-output** attempts to get all buffers sent off, **clear-output** attempts to get all buffers flushed.

This is primarily intended for terminals, although it could be meaningful for random other devices (ascii and binary). It does not do anything, and is not really expected to, to a random disk file.

It doesn't do anything to anything in NIL.

19.4.1 Ascii Output

write-char *char* &optional *output-stream*
Writes *char* to *output-stream*.

terpri &optional *output-stream*
fresh-line &optional *output-stream*
terpri performs a newline on *output-stream*.

fresh-line does so, unless it can determine that the "cursor" is at the left margin.

fresh-line is supposed to return t if it performed a newline, nil otherwise. **terpri** always returns nil, for historical reasons.

oustr *string* &optional *output-stream* (*start 0*) *count*
Standard NIL string-output. Outputs the characters of *string*, starting at index *start* and proceeding for *count* characters, to *output-stream*. This is not defined by COMMON LISP, but has been in NIL for some time and is extremely useful for doing efficient output because it passes a pseudo-substring defined by *start* and *count* along to the stream. Most NIL streams do this more efficiently than single-character output, especially the terminal stream.

write-string *string* &optional *stream*
write-line *string* &optional *stream*
Writes the characters of *string* to *stream*. *write-line* follows them by a newline (terpri, page 185). In NIL this is almost always faster than using a loop of **write-chars**.

princ *object* &optional *output-stream*
prin1 *object* &optional *output-stream*
print *object* &optional *output-stream*
Standard Maclisp-style printing functions.

prin1 is the basic printing function, which attempts to output the printed representation of *object* to *output-stream* in such a way that it might be reconstructable with **read**. No newline or whitespace of any kind is output before or after, so delimiters of some sort might be needed between successive calls.

print adds those necessary delimiters: it does a terpri first, and writes a space character afterwards.

princ is pretty much the same as prin1 except it does not try to make the output readable with read, but rather outputs things "literally" insofar as that is possible with arbitrary Lisp objects. Strings, for example, are written as if by oustr—simply their contents. Symbols have their print-names written as for strings, etc. Numbers are generally printed the same as they are by prin1.

19.4.2 Binary Output

write-byte *integer binary-output-stream*
Writes the byte *integer* to *binary-output-stream*.

Note that the order of arguments here is the reverse of what the MACLISP **out** *function takes. Because of earlier confusion, the* write-byte *function accepts its arguments in either order right now.*

It is an error if *integer* does not fit in the byte size the stream deals with. *How is this defined? Probably by the stream, i.e. the bytes could be signed or not, the current ones are not and are 8-bits, so* integer *can range from 0 to 255.*

write-bits *binary-output-stream bits*
Writes the bit-vector *bits* to *binary-output-stream*. The intent of this is that *bits* is taken to be a concatenation of many bytes of data of whatever size the stream deals with.

It is an error for the size (in bits) of *bits* to not be an exact multiple of the byte size of the stream.

This function is provided primarily to help speed up the NIL compiler in creating VASL files. The semantics may change some as additional forms of binary streams are added to NIL.

This may in fact be flushed.

19.5 Formatted Output

See also format, which is sufficiently complex (and, in NIL, somewhat programmable) that it a separate section devoted to it (section 19.6, page 187).

pretty-prin1 *object* &optional *stream*
Similar to prin1, but outputs *object* in (what is hoped to be) a significantly more aesthetic format, with indentation showing nesting depth etc. The output starts wherever the cursor happens to be on *stream*; pretty-print may be used to do this on a new line.

pretty-prin1 assumes that *object* is actually LISP code, and bases its formatting behaviour on stylistic conventions used for indenting various program constructs. pretty-prin1-datum may be used if *object* should not have these heuristics applied.

In NIL, pretty-print1 attempts to determine the existence of circular structure, and show this somehow without blowing up.

The pretty-printer itself is described in much more detail in [7].

pretty-print *object* &optional *stream*
pretty-print1, with a terpri first and output of a space character after. This, pretty-print is to pretty-print1 as print is to prin1.

pretty-print1-datum *object* &optional *stream*
Like pretty-print1, but does *not* assume that *object* is LISP code.

pretty-print-datum *object* &optional *stream*
Similar.

19.6 Format

This section is a quick reworking of the chapter on format which appeared in [3]. It omits topics specific to implementations of format other than NIL's, and includes references to differences between what NIL currently provides, and the definition of format provided by COMMON LISP. It is not known at this time how the COMMON LISP definition of format will affect the MACLISP implementation, which utilizes the same source code as the NIL implementation right now. The NIL version of format will of course be made to conform to the COMMON LISP definition.

format *destination control-string* &rest *args*
format is used to produce formatted output. format outputs the characters of *control-string*, except that tilde ("~") introduces a directive. The character after the tilde, possibly preceded by arguments and modifiers, specifies what kind of formatting is desired. Some directives use an element of *args* to create their output.

The output is sent to *destination*. If *destination* is nil, a string is created which contains the output. If *destination* is t, the output is sent to the "default output destination", the value of standard-output (page 179). Otherwise, *destination* should be an output stream.

?format *destination control-string* &rest *args*
This is equivalent to format except that *destination* is interpreted just like the stream argument to print—nil means "the default" (the value of standard-output), and t means "the terminal" (the value of terminal-io). This only exists in MACLISP and NIL.

A directive consists of a tilde, optional decimal numeric 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 character is ignored. Examples of control strings:

```
"~S"        ; This is an S directive with no parameters.
"~3,4:@s"   ; This is an S directive with two parameters, 3 and 4,
            ;   and both the colon and atsign flags.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use format efficiently. 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 numeric 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 numeric argument. For example, you can use

```
"~5,'0d"
```

to print a decimal number in five columns with leading zeros (the first two parameters to ~D are the number of columns and the padding character).

In place of a numeric parameter to a directive, you can put the letter v, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but 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.

It is possible to have a directive name of more than one character. The name need simply be enclosed in backslashes ("\"); for example,

```
(format t "~\\now\\" (status daytime))
```

The backslashes above are doubled, because backslash is the quoting character in COMMON LISP. Because of this, the forward-slash character (/) will probably be made synonymous with backslash, but as of yet it has not been. As always, case is ignored here. There is no way to quote a backslash in such a construct. No multi-character operators come with format.

Once upon a time, various strange and wonderful interpretations were made on *control-string* when it was neither a string nor a symbol. Some of these are still supported for compatibility with existing code (if any) which uses them; new code, however, should only use a string or symbol for *control-string*.

19.6.1 The Operators

Here are the operators.

~A *arg*, any LISP object, is printed without slashification (like princ). ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. ~*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 *space* for *padchar*. The atsign modifier causes the output to be right-justified in the field instead of left-justified. (The same algorithm for calculating how many pad characters to output is used.) The colon modifier causes an *arg* of nil to be output as ().

~S This is identical to ~A except that it uses prin1 instead of princ.

~D Decimal output. *arg* is printed as a decimal integer. ~*n*,*m*,*o*D uses a column width of *n*, padding on the left with pad-character *m* (default of space), using the character *o* (default comma) to separate groups of three digits. These commas are only inserted if the : modifier is present. Additionally, if the @ modifier is present, then the sign character will be output unconditionally; normally it is only output if the integer is negative. If *arg* is not an integer, then it is output (using princ) right-justified in a field *n* wide, using a pad-character of *m*, with decimal output radix and trailing decimal point suppression.

If *arg* is not an integer, then it is output in the specified field (as by ~A), in decimal.

~O Octal output. Just like ~D; if *arg* is not an integer, it is output (as by ~A), in octal.

~B Similar, but binary (base 2).

~P If *arg* is not the integer 1, a lower-case "s" is printed. ("P" is for "plural".) ~:P does the same thing, after backing up an argument (like "~:*", below); it prints a lower-case s if the *last* argument was not 1. ~@P prints "y" if the argument is 1, or "ies" if it is not. ~:@P does the same thing, but backs up first.
Example:

```
(format nil "~D Kitt~:@P" 3)  =>  "3 Kitties"
```

~* ~* ignores one *arg*. ~*n** ignores the next *n* arguments. *n* may be negative. ~:* backs up one arg; ~*n*:* backs up *n* args.

~*n*@* is an "absolute goto"; it "goes to" the *n*th argument.

This directive only affects the "local" args, if control is within something like ~{.

~% Outputs a newline. ~*n*% outputs *n* newlines. No argument is used.

~& Performs a fresh-line on the output stream (page 185). ~*n*& outputs *n*-1 newlines after the fresh-line.

~X Outputs a space. ~*n*X outputs *n* spaces. No argument is used. This directive is changing in COMMON LISP to mean "hexidecimal output" (done like ~D); to get the effect of the old ~X, one can use ~T, or some construct utilizing ~< or ~{.

~~ Outputs a tilde. ~*n*~ outputs *n* tildes. No argument is used.

~*newline*

Tilde immediately followed by a newline ignores the newline and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the newline is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program:

```
(format the-output-stream "~&This is a reasonably ~
                           long string~%")
```

which is equivalent to formating the string

```
"~&This is a reasonably long string~%"
```

~| Outputs a formfeed. ~*n*| outputs *n* formfeeds. No argument is used.

In the current implementation, ~| will do something like try to clear the screen on a terminal. However, this will be changed: the intent of ~| is to output the page separator character, which might be inconvenient to type in or have sitting in the middle of a format string in ones source file. To get the old behavior, use ~:|—eventually ~| will be changed.

~T Spaces over to a given column. The full form is ~*destination*,*increment*T, which will output sufficient spaces to move the cursor to column *destination*. If the cursor is already past column *destination*, it will output spaces to move it to column *destination* + *increment* * *k*, for the smallest integer value *k* possible. *increment* defaults to 1. This is implemented by the format-tab-to function, page 196. On certain streams, this may not actually output spaces, but may use cursor positioning; thus, one should not depend on ~T "erasing" text by the typing of spaces.

This will be changed slightly to eliminate a common and unsightly fencepost. Currently ~T will do nothing if the "cursor" is exactly at column *destination*; however, it will be changed so that spacing will be done then too.

In the future, ~@T will do *relative* positioning.

~Q ~Q uses one argument, and applies it as a function to *params*. It could thus be used to, for example, get a specific printing function interfaced to format without defining a specific operator for that operation, as in

```
(format t "~&;  The frob ~vQ is not known.~%"
          frob 'frob-printer)
```

The printing function should obey the conventions described in section 19.6.2, page 194. Note that the function to ~Q follows the arguments it will get, because they are passed in as format parameters which get collected before the operator's argument. Not in COMMON LISP.

~[~[*str0*~;*str1*~;...~;*strn*~] is a set of alternative control strings. The alternatives (called *clauses*) are separated by ~; and the construct is terminated by ~]. For example, "~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~;Tiger ~;Yu-Hsiang ~]kitty". The *arg*th alternative is selected; 0 selects the first. If a numeric parameter is given (i.e. ~*n*[), then the parameter is used instead of an argument (this is useful only if the parameter is "#"). If *arg* is out of range no alternative 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 ~;Tiger ~;Yu-Hsiang ~:;Unknown ~] kitty".

~[~*tag00*,*tag01*,...;*str0*~*tag10*,...;*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*,...; 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 nil, and selects the *true* control string otherwise.

~@[*true*~] tests the argument. If it is not nil, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is nil, then the argument is used up, and the clause is not processed.

```
(setq prinlevel nil prinlength 5)
(format nil "~@[ PRINLEVEL=~D~]~@[ PRINLENGTH=~D]"
        prinlevel prinlength)
  => " PRINLENGTH=5"
```

~R If there is no parameter, then *arg* is printed as a cardinal English number, e.g. four. With the colon modifier, *arg* is printed as an ordinal number, e.g. fourth. With the atsign modifier, *arg* is printed as a Roman numeral, e.g. IV. With both atsign and colon, *arg* is printed as an old Roman numeral, e.g. IIII.

If there is a parameter, then it is the radix in which to print the number. The 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.

~C *arg* is coerced to a character code. With no modifiers, ~C simply outputs this character. ~@C outputs the character so it can be read in again using the # reader macro: if there is a named character for it, that will be used, for example "#\Return"; if not, it will be output like "#/A". ~:C outputs the character in human-readable form, as in "Return", "Meta-A". ~:@C is like ~:C, and additionally might (if warranted and if it is known how) parenthetically state how the character may be typed on the user's keyboard.

To find the name of a character, ~C looks in two places. The first is the value of the symbol which is the value of format:*/#-var, which is initialized to be the variable which the # reader macro uses. It is not necessary for the value of format:*/#-var to be bound. The second place is *format-chnames; this is used primarily to handle non-printing characters, in case the # reader macro is not loaded. Both of these are a-lists, of the form ((*name* . *code*) (*name* . *code*) ...).

The Maclisp/NIL format has no mechanism for telling how a particular character needs to be typed on a keyboard, but it does provide a hook for one. If the value of format:*top-char-printer is not nil, then it will be called as a function on two arguments: the character code, and the character name. If there were bucky-bits present, then they will have been stripped off unless there was a defined name for the character with the bits present. The function should do nothing in normal cases, but if it does it should output two spaces, and then the how-to-type-it-in description in parentheses. See section 19.6.2, page 194 for information on how to do output within format.

~< ~*mincol*,*colinc*,*minpad*,*padchar*<*text*~> justifies *text* within a field *mincol* wide. *text* may be divided up into segments with ~;—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 colon modifier causes spacing to be introduced before

the first text segment; the atsign modifier causes spacing to be added after the last. *minpad*, default 0, is the minimum number of *padchar* (default space) padding characters to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to 1. For example,

```
(format nil "~10<foo~;bar~>")     => "foo    bar"
(format nil "~10:<foo~;bar~>")    => "  foo  bar"
(format nil "~10:@<foo~;bar~>")   => "  foo bar "
(format nil "~10<foobar~>")       => "    foobar"
(format nil "~10:@<foobar~>")     => "  foobar  "
(format nil "$~10,,,'*<~3f~>" 2.59023)
                                  => "$******2.59"
```

If ~^ is used within a ~< construct, then only the clauses which were completely processed are used. For example,

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
        => "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
        => "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
        => "FOO   BAR   BAZ"
```

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 omitted 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 for the first clause is output before the padded text. The first clause ought to contain a carriage return. The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting piece of text, not whether to process the first clause. If the ~:; has a numeric 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 argument 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 numeric 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~>~^,~}.~%"
```

Note that the segments ~< breaks the output up into are computed "out of context" (that is, they are first recursively formatted into strings). Thus, it is not a good idea for any of the segments to contain relative-positioning commands (such as ~T and ~&), or any line breaks. If ~:; is used to produce a prefix string, it also should not use relative-positioning commands.

~{*str* ~}

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. If before any iteration step the list is empty, then the iteration is terminated. Also, if a numeric parameter *n* is given, then there will be at most *n* repetitions of processing of *str*.

~:{*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.

~@{*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.

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a list. On each iteration one argument is used as a list of arguments.

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 numeric parameter of zero).

If *str* is null, 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:

```
(apply (function format) (list* 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.

~} Terminates a ~{. It is undefined elsewhere.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. (In the latter case, the ~< formatting *is* performed, but no more clauses are processed before doing the justification. The ~^ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause. It may appear anywhere in a ~{ construct.) If there is no such enclosing construct, then the entire formatting operation is terminated.

If a numeric parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) 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.

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 ~:^.

~*n*G This is the old form of ~*n*@*, above. In COMMON LISP, ~G is a floating-point format directive derived from FORTRAN G format. New code should use ~@*.

~F Reserved for fixed-field floating-point format.

~E Reserved for exponential floating-point format.

~$ *This is not yet defined in NIL.*

~*rdig*,*ldig*,*field*,*padchar*$ prints *arg*, a flonum, with exactly *rdig* digits after the decimal point. The default for *rdig* is 2, which is convenient for printing amounts of money. At least *ldig* digits will be printed preceding the decimal point; leading zeros will be printed if there would be fewer than *ldig*. The default for *ldig* is 1. The number is right justified in a field *field* columns long, padded out with *padchar*. The colon modifier means that the sign character is to be at the beginning of the field, before the padding, rather than just to the left of the number. The atsign modifier says that the sign character should always be output.

In some implementations, if *arg* is unreasonably large, it will be printed in ~*field*,,,*padchar*@A format; i.e. it will be princ'ed right-justified in the specified field width. This will not happen in the Maclisp implementation, because the range provided by flonums is not extremely large.

~\ This is not really an operator. If one desires to use a multi-character format operator, it may be placed within backslashes, as in ~\now\ for the now operator. See page 188.

19.6.2 Defining your own

Everything in this section is defined in the MACLISP and NIL implementations of format only.

define-format-op

This may be used in two formats:

(define-format-op *operator varlist body-forms...*)

and

(define-format-op *operator character*)

The *operator* may be a character, string, symbol, or fixnum code for a character (in NIL, it is coerced using the string function, page 112). Whichever, it is canonicalized (into upper case) and will be interned into the same package which format resides in. For example, the format operator for *tilde* could be defined as

(define-format-op \~ #\~)

For the first format, the type of operator is determined by decoding *varlist*, which may have one of the following formats:

(*params-var*)
An operator of exactly zero arguments; *params-var* will get bound to the parameters list.

(*params-var arg-var*)
An operator of exactly one argument; *params-var* will get bound to the

parameters list, and *arg-var* to the argument.

(*params-var* . *args-var*)
An operator of a variable number of args; *params-var* will get bound to the parameters list, and *args-var* to the remaining arguments to format (or to the recursive ~{ arguments). The operator should return as its value some sublist of *args-var*, so that format knows how many were used.

A definition for the appropriate function is produced with a bvl derived from the variables in *varlist* and a body of *body-forms*. (The argument ordering in the function produced is compatible with that on the Lisp Machine, which is *arg-var* (if any) first, and then *params-var*.)

standard-output *Variable*
Output from format operators should be sent to the stream which is the value of standard-output.

format:colon-flag *Variable*
format:atsign-flag *Variable*
These tell whether or not we have seen a colon or atsign respectively while parsing the parameters to a format operator. They are only bound in the toplevel call to format, so are only really valid when the format operator is first called; if the operator does more parameter parsing (like ~[does) their values should be saved if they will be needed.

The *params* are passed in as a list. This list, however, may be temporary storage only; one should not allow it to be "passed back" from the call to the format operator without being copied first. Also, it is recommended that format *params* be referenced with elt, and their length obtained with length, in order that they may be reimplemented as some sort of sequence in the future (which will probably be a stack-allocated vector for NIL).

Conceptually, format operates by performing output to some stream. In practice, this is what occurs in most implementations; in Maclisp, there are a few special SFAs used by format. This may not be possible in all implementations, however. To get around this, format has a mechanism for allowing the output to go to a pseudo-stream, and supplies a set of functions which will interact with these when they are used.

format-tyo *character*
tyos *character* to the format output destination. *character* may be either an object of type character, or the fixnum code for a character.

format-princ *object*
princs *object* to the format output destination.

format-prin1 *object*
prin1s *frob* to the format output destination.

format-lcprinc *string capitalize?*

This outputs *string*, which must be a string or symbol, to the format output destination in lower-case. If *capitalize?* is not nil, then the first character is converted to upper case rather than lower.

format-terpri

Does a terpri to the format output destination.

format-charpos
format-linel

Return the charpos and linel of the format output destination. Since in the MACLISP implementation multiple output destinations may be implicitly in use (via outfiles, for instance), this attempts to choose a representative one. The terminal is preferred if it is involved.

format-fresh-line

This performs the fresh-line operation to the default format destination. The hair involved in this is mostly subsumed by the fresh-line function in NIL.

format-tab-to *destination* &optional *increment*

This implements ~T to the current format destination (q.v.). In NIL, it will utilize the :tab-to message if that is supported. Otherwise, if it can determine the "current position" of the format destination, it will output the proper number of spaces; all else failing, two spaces will be output.

format-formfeed

Performs a formfeed on the format output destination. In NIL, this will send the :formfeed message to the stream if that is supported, the :clear-screen message if that is supported, otherwise just output the page separator character. The :formfeed message is supported by a number of NIL streams, and is designed for just this use.

format-flatc

(**format-flatc** *form1 form2 ... formn*)

The *forms* are evaluated in an environment similar to that used inside of format: the various format output-performing routines such as format-tyo and format-princ may be used to "perform output". In all but the MULTICS MACLISP implementation, standard-output will be a stream which simply counts the characters output—it will only support the :write-char operation.

19.7 Querying the User

The following routines are built on the fquery function, which is modeled after that of LISP MACHINE LISP. fquery is complicated and subject to change, however, and is not itself documented here. Of the following routines, y-or-n-p and yes-or-no-p are defined by COMMON LISP; the others are not.

y-or-n-p &optional *message stream*
This prints *message* to *stream* (which defaults to the value of query-io), and then reads a character from *stream*. It returns t or nil depending on whether the character signified a positive or negative response: space and rubout are accepted in place of y and n. Because it is so easy to get a mistaken response from this routine, it should be used for anticipated questions only.

Because it is used for both input and output, *stream* must be bi-directional.

yes-or-no-p &optional *message stream*
This is similar to y-or-n-p, but requires a more complete answer. Typeahead to *stream* is flushed (with clear-input, page 183), and it feeps, before reading a complete "yes" or "no" followed by a newline.

format-y-or-n-p *format-string* &rest *format-args*
Most the time when y-or-n-p is used, people seem to want to use a format string with some arguments. This does that. Input and output is done to query-io. Otherwise, it behaves like y-or-n-p.

format-yes-or-no-p *format-string* &rest *fomrat-args*
Similar.

19.8 Filesystem Interface

The NIL filesystem interface is designed to allow it to refer to more than one filesystem. The names of files are not represented as just strings or lists of components, but are objects of type pathname. The pathname objects for different filesystems or *hosts* would be of different types, and operations on files in the filesystem are performed with respect to that type. For instance, we have under development facilities to allow use of the filesystems of TOPS-20 and ITS through CHAOSNET. At the moment, only the local VMS filesystem is supported.

19.8.1 Pathnames

A pathname has six criterial components.

host
This component always contains an object which describes the filesystem the pathname refers to. All pathnames have such a component; no pathname may be formed without such a component. Thus, pathname interpretation is always performed with respect to some host.

device
This is normally a string, naming a device.

directory
A string naming a directory, or a list of strings, if the directory is structured (that is, if the pathname is in a subdirectory).

name
A string, the "primary" or "root" name of the file.

type
A string, the "type" of the file. This is not necessarily as the *extension* which will be used to form the host-specific pathname string; e.g., for a VMS filesystem, a file type of LISP corresponds to the extension LSP.

version
This is the version of the pathname; usually it is an integer.

A pathname need not refer to an actual file in a filesystem, nor need all the components (other than the host) be present. An unspecified component is represented by **nil**. Such a component may be supplied by later defaulting operations. Components may also contain certain keywords which are interpreted specially:

:wild
A "wildcard" component.

:newest
:oldest
These are only applicable to the **version** component of a pathname. They cause the reference to the filesystem to refer to the newest or oldest version present. Only **:newest** is actually supported by the VMS filesystem interface.

:unspecific
If any component in a pathname has this as its value, then the pathname does not refer to a specific file in the filesystem, but rather to the group of files which match the other components. This is normally only used for the **type** or **version** components, so that one may refer to the entire group of files with the same device, directory, and name. This doesn't have any use in NIL yet; when NIL pathnames gain the ability to have arbitrary attributes (properties) associated with them, it will be significant.

:implied
If a pathname has this as a component, it means that the **device** component is a logical name which will supply the value for that component. This is used as a placeholder for "pathname merging". All components other than the host and device components may

take on this as their value.

:relative
This can only occur within a structured directory component. It is used in the representation of VMS rooted directories. For instance, the VMS pathname `__DBA0:[NIL259.]` parses into a pathname with the string `"__DBA0"` as its device component, and the list `("NIL259" . :relative)` as its directory component. NIL uses this in pathname merging.

:elipsis
This is used in representing things like `nil$disk:[nil...]`. Parsing that into a pathname would produce a directory component of `("NIL" . :elipsis)`. It will probably be flushed in favor of utilizing just :wild instead.

19.8.1.1 Pathname Functions

pathname *thing*
thing is coerced into a pathname.

If it is a pathname, it is returned.

If it is a list, then it is assumed to be a MACLISP namelist; interpretation of this, and MACLISP compatible pathname handling, is discussed in <not-yet-written>.

If it is a string (or symbol), then the text is examined for a prefix or suffix component, followed by a ":", which is a host string; if one is found, then that is the host used, otherwise a host is defaulted (the handling of this is pretty spastic right now, but hardly matters as there is only one host). The string is then parsed in the manner specific to that host, and the resultant pathname returned.

pathname-host *pathname*
pathname-device *pathname*
pathname-directory *pathname*
pathname-name *pathname*
pathname-type *pathname*
pathname-version *pathname*
These return the components of *pathname*, which is coerced to a pathname with the pathname function.

namestring *pathname*
pathname is coerced to a pathname with the pathname function, and its "standard printed representation" returned, as a string.

user-homedir-pathname &optional *host*
Returns the user's home directory, as a pathname: the name, type, and version components will be unspecified.

The home directory is where files specific to the user are looked for (or defaulted to). See, for instance, init-file-pathname, page 200.

Under VMS, this is obtained by translating the logical name SYS$LOGIN.

user-workingdir-pathname &optional *host*
Returns the user's working directory, as a pathname: the name, type, and version components will be unspecified.

For VMS, this is obtained from the RMS default directory string, and the SYS$DISK logical name. Note that the RMS default is copied from the command interpreter when the NIL process is created; temporarily exiting the NIL and changing the default will *not* change the value of this.

user-scratchdir-pathname &optional *host*
Returns, as a pathname, the directory of the directory which should be used by programs for writing "scratch" files.

The local-vms host uses the value of the logical name SYS$SCRATCH if that exists, otherwise the user's home directory. If for some reason the device field is absent, SYS$DISK is supplied. Note that the value of the logical name is copied from the command interpreter when the NIL is created. Temporarily exiting from the NIL and changing the logical name definition will have no effect.

init-file-pathname *program-name* &optional *host*
This returns the pathname of the user's init file for *program-name* on *host*. *program-name* should be a string.

For NIL under VMS, the init file is on the user's home directory, and has name NIL and extension INI (the file type is INIT). This same convention is used in general by this function: for an arbitrary program name, the init file is named, essentially,

SYS$LOGIN:*program-name*.INI

In the NIL programming environment, this is more for the use of LISP subsystems than a general facility (which could do things like determine the init file for logging into the vax). For example, if you had a system LSB which people loaded into their NIL, or which was dumped out in a NIL, it might load an LSB init file. Note that there is a problem here if *program-name* is not valid as a pathname name component for the particular host.

19.8.1.2 Merging and Defaulting

Merging and defaulting are the actions used to fill in components missing from a pathname specification, usually when the pathname is about to be used to reference something in the filesystem. For the most part, this involves supplying the components missing in one pathname from another. The algorithm used is slightly more complicated, and is described under merge-pathname-defaults, below.

In NIL, the pathname defaults for a specific application are maintained in a *pathname defaults* object (it will probably be of type fs:pathname-defaults). This enables modular handling of supplying of defaults for multiple hosts, pathname "stickiness" for sets of commands, etc. The defaults are often used to supply the host with respect to which some operation must be

performed, such as pathname parsing.

merge-pathname-defaults *pathname* &optional *defaults default-type default-version*

This is the main merger. *pathname* may be anything coercible to a pathname. *defaults* may be a pathname defaults object, a pathname, or a string or symbol (which will be coerced to a pathname first). *default-type* and *default-version* may be whatever is allowable for types and versions.

If it is necessary, *pathname* will be parsed with respect to a host determined from *defaults*. If the directory field of it is missing, then that will be supplied by *defaults*. There is some question as to what should happen if the device field of *pathname* is missing: currently, it is simply filled in from *defaults*. In the Lisp Machine implementation of this function, it is supplied as the default device for the host (perhaps inconsistently, for instance only when parsed from a string?); probably what should happen is that whether the device comes from *defaults* or not is determined by the host, so that it would if the devices were really structured (with directories in them etc.), and would not otherwise (which in Lisp Machine Lisp appears to be mainly for the sake of the ITS operating system).

If *pathname* has a name supplied, then if the type and version of the resulting pathname are defaulted from *default-type* and *default-version*, as necessary. Otherwise, the name, type, and version are defaulted from *defaults*. Thus:

```
(merge-pathname-defaults
   "[nil.vas]foo" "sys$disk:[gsb]zz.lsp;3" "vasl")
  => #<local-vms-pathname "node:sys$disk:[nil.vas]foo.vas">
(merge-pathname-defaults
   "[nil.vas]" "sys$disk:[gsb]zz.lsp;3" "vasl")
  => #<local-vms-pathname "node:sys$disk:[nil.vas]zz.lsp;3">
(merge-pathname-defaults
   "[nil.vas]=.inp" "sys$disk:[gsb]zz.lsp;3" "lisp")
  => #<local-vms-pathname "node:sys$disk:[nil.vas]zz.inp;3">
```

In the above, it is worth noting that "=" as a pathname component is used as a placeholder for an unspecified component, and that a file type of lisp is mapped (by the VMS pathname code) into an extension of lsp, and vasl to vas. The specifics of the syntax and file-type/extension mapping are described elsewhere.

The default value of *default-version* is :newest. The default value of *default-type* is "lisp"; however, it is highly recommended that this *not* be depended upon, as it may be changed to come from *defaults*.

Here are some of the pathname defaults in use in NIL.

load-pathname-defaults *Variable*

This is used to provide pathname defaults for load, compile-file, and any similar functions. It is initialized to the user's working directory with name FOO and type LISP (see user-workingdir-pathname, page 200).

default-pathname-defaults *Variable*

This provides super-defaults for anything that needs them, such as open, with-open-file, and parsing a pathname string out of context. It is initialized to the user's working directory with name FOO and type LISP (see user-workingdir-pathname, page 200).

scratch-pathname-defaults *Variable*

This is used by things which must write out "temporary" files. Things which use this should not modify it; it should be left to the user to set default pathnames for hosts (primarily for the sake of the device and directory) to say where such files should be written. See user-scratchdir-pathname, page 200.

For example, the current NIL compile function creates a file named aaafoo.vas, and supplies the device and directory from *scratch-pathname-defaults*.

19.8.2 Opening Files

Files are opened with the open function (page 180), or with-open-file (page 181), and may be closed with close (page 181). open by default assumes that the open is a reference to a file, so coerces its first argument to a pathname, and then creates a stream to the specified file in the filesystem. NIL currently employs only two modes of file opening. These are :ascii and :byte modes.

:ascii will cause the file to be written as variable-length records, with record-attribute of carriage-return. The existing disk I/O code does not have the intelligence to deal with records longer than 512 bytes, however, so is forced to terminate records when that limit is reached. To compensate, so that spurious newlines do not get inserted into LISP files, records exactly 512 long are assumed to not actually be terminated, but "continued" with the next record, when read as input.

:fixnum simply uses fixed-length 512-byte records; this is what vas1 files use.

fs:close-all-files

In case you *do* mess up and lose track of some files, this will close all open files (which have been really opened as streams, not just kludgey temporary opens). Every known host object is supposed to keep track of all streams which it has open, in such a way as to be secure against timing screws, so that this may at least be done.

Doing (exhibit fs:*host-instances*) should give one a handle on the open files, as the host instances should point to the files they have open. fs:*host-instances* is the list of all known hosts, and is used to (among other things) drive host-name lookup.

19.8.3 Other File Operations

probe-file *pathname*
If *pathname* (which is coerced to a pathname with the pathname function) can be opened, its truename is returned; otherwise, nil is returned. This may be used to see if *pathname* exists and is accessible. (If a file protection error occurs, probe-file returns nil, although that may change, as the intent is to see if the file exists.)

Note file-length and filepos are missing from NIL.

All of the remaining functions in this section deal with either a stream, or with a pathname. For the former, they perform the operation on the open stream; for the latter, on the file in the filesystem, which may involve opening the file temporarily. At present, none of them work on streams. A future edition of the filesystem code will contain more code written in LISP, and be much more versatile in this regard.

For the functions which are described as returning an error description, this is probably a string, but may change to be a more complicated object in the future. (That object should, however, have the property that it will print with princ as the error message.) Tests made on the return result should be made accordingly; that is, be based on null, or streamp, or listp or whatever. Signalled errors are typically signalled as proceedable :io-lossage errors; returning a value from the error should cause the function to return that as its value.

rename-file *file new-name* &optional (*error-p t*)
Renames *file*, a filename or a stream open to a file, to *new-name*, which must be coercible to a pathname. If *error-p* is not nil, then a file-system error will be signaled as a LISP error; if it is nil, then an error description will be returned. If everything goes fine, nil is returned.

delete-file *file* &optional (*error-p t*)
The file named by *file*, or the file open on the stream *file*, is deleted. If an error occurs, then a LISP error will be signaled if *error-p* is not nil, otherwise an error description will be returned. If all goes well, nil is returned.

file-creation-date *file*
This returns the creation date of the file as an integer in universal time format (see section 23.5, page 234), or nil if this cannot be determined.

file-author *file*
Returns the name of the author of the file as a string, or nil. For VMS files, the string is a UIC, e.g. "[200,007]", and the group and member numbers are guaranteed to be padded with leading zeros to at least three digits.

19.8.4 File Matching

allfiles *list-of-pathnames*

Returns a list of pathnames matching all of those in *list-of-pathnames*. This is done, of course, by appending together the lists of pathnames which match each of the pathnames in *list-of-pathnames*.

By convention, this matches over all those components not specified in each pathname. VMS does not allow matching all devices, however, so the device should be specified, or will be defaulted from somewhere (where? rms default but it shouldn't be). Newest versions can be matched also, by using the appropriate pathname syntax. Note that elipsis specifications in directories, and star specifications in names, all work (fortuitously, perhaps, but they work): e.g., (allfiles "[nil...]286*.*;") returns a list of pathnames of the files with highest version number of all the files in the NIL hierarchy with first three characters of their name being "286".

mapallfiles *function list-of-pathnames*

Calls *function* on each pathname which matches pathnames in *list-of-pathnames*. This is essentially equivalent to

(mapc *function* (allfiles *list-of-pathnames*))

but calls *function* on each as it is generated rather than consing up the list.

In fact, allfiles is implemented in terms of mapallfiles.

mapallfiles (and hence allfiles) accept a single symbol/string/pathname in place of a list. It is unclear what should be done about this; it is (currently at least) of no use to NIL to deal with multiple specifications at once, and in fact the original allfiles function in MULTICS MACLISP did not take a list, but only a single pathname.

There is also no kludgy testing in NIL such that if a namelist is specified it must be a fully-specified namelist (insofar as explicit "*" components are specified). Thus using a namelist as a single pathname will be interpreted as a list of pathnames, potentially resulting (incorrectly) in a match over all the files on the current device... (That was the reason for the kludgey check in MACLISP, you see...)

19.8.5 Loading Files

load *filename* &key :verbose :package :set-default-pathname :static :default-pathname :characters :binary :defaults :print

:verbose
Boolean: print out lots of gubbish about loading. Default is value of *load-verbose*.

:package
override the package specification (if any) obtained from the file.

:set-default-pathname
Boolean: set the default pathname of the pathname-defaults used (see :defaults and :default-pathname, below). Default is value of *load-set-default-

pathname*.

:defaults
Specified pathname-defaults to use in place of the value of *load-pathname-defaults*. (This one isn't in COMMON LISP. Not clear it should be heavily used, but i can see it has application.)

:default-pathname
Use this for defaulting, in preference to the pathname-defaults. The defaults are still set in the defaults specified by :defaults (or its absence).

:characters
:binary
Boolean. :binary t implies that the file is a VASL file; :CHARACTERS T implies that the file is LISP text. By default, the file is examined to determine which it is. And, if no type is specified in *filename*, first a file with type vasl and then a file with type lisp will be looked for.

:static
Boolean; says to load the file into the static heap. Default is value of *load-in-static-area*.

:print
Boolean; if not nil, says that the results of evaluation of forms in the file are to be printed. Default is nil, and it probably doesn't work anyway; certainly not for vasl files.

19.8.6 File Attribute Lists

Not too much detail yet... However, it's necessary to use it.

If, on the first line of a source file the characters "-*-" appear, then the text from the first "-*-" to the next is parsed as a *file attribute list*. (Funnyness with multiple lines? Well, anyway, it works easily on one line.) This text is logically a list of keyword/value pairs, with the values being either single values or lists of values. The entire construct is made invisible to the processing language by being placed within its commenting construct.

```
; -*- Mode:Lisp; Package:Compiler; Base:10; Readtable:NIL -*-
```

might occur as the first line in a NIL compiler source file. It says that the mode of the file is lisp (this being for the benefit of text editors), and that the file should be read and processed in the compiler package, decimal radix, and using the NIL readtable. "Lists" of values are provided for by separating the individual items by commas, as in

```
; -*- Mode:Lisp; LSB:ppdef,pretty-print-definition -*-
```

The parsing of tokens in such a construct is pretty rudimentary and crufty, but essentially things are symbols except for a series of digits (optionally followed by a decimal point) which is a decimal integer.

File attributes typically translate into some special binding environment needed for the processing of the file (in some context). The following are pre-defined in NIL:

package *package-name*
The file is processed in the package named *package-name*.

readtable *readtable-name*
: The file is read in using the readtable named *readtable-name*. Syntax, and readtable naming, is described in section 16.3, page 126.

base *radix*
: Binds both the input and output radices, no matter what those damned variables are named.

radix *radix*
: For those who are confused by **base** and will be even moreso when the variable is named ***base***.

patch-file *yes-or-no*
: If *yes-or-no* is **yes** (it shouldn't be specified otherwise), then a variable proclaiming the patch-file-ness of this file is bound, so that various things can see it and be clever, like the helpful function which warns you about redefining a function defined by someone else in another file (said function not existing yet).

lsb *module-name,system-name*
: This is defined by LSB [4], not NIL (q.v.).

When a file attribute list is parsed, the attribute names are keywords, and the values are either keywords or integers (or lists of keywords or integers).

:file-plist *pathname*
: If *pathname* can be opened, then this returns a disembodied property list with the file attributes in the **cdr**, and the truename of the file in the **car**. Otherwise, it returns **nil**.

  This probably should be renamed **:file-attribute-list**.

fs:process-in-load-environment *plist funct pathname* &rest *args*
: *plist* should be a parsed file attribute list (with an *even* number of elements; the **cdr** of what is returned by the **:file-plist** message). *pathname* should be the pathname which the file attributes were obtained from.

  The environment which is specified by that attribute list is established, and then *funct* called with arguments of *pathname* and whatever *args* were given.

  This is what is used by both **load** and the compiler. It enables a stable interface to how bindings and other environment modifications are obtained from the file attributes.

Examination of the source code (the file **[NIL.IO]PATHN.LSP**) will show the convention which is used for defining additional file attributes. It is basically an extension of that defined by LISP MACHINE LISP.

19.8.7 Internals for VMS Record Management Services

NIL contains some primitive routines and datastructure definitions with which I/O can be performed entirely from LISP code. In fact, recently a special-purpose stream type was written so that the editor could do very fast record I/O. This is the local-vms-editor-stream stream, which is in the source file `nil$disk:[nil.io]vlocal.lsp`, and can serve as an example of how some of these things are used. Some of the other primitives which interface to the filesystem are also written in LISP, but are hidden from the stream code by a functional interface; these are in the file `nil$disk:[nil.vmlisp]vmsfile.lsp`.

19.8.7.1 Data Structures

The datastructures used for RMS fab, rab, nam, and xab blocks are all just simple bit vectors of the appropriate size. There are LISP macros and constants defined to deal with them; these definitions were generated from the VMS MDL files. For instance, (si:fab$b_rfm *fab*) returns the record-format field of a fab. (setf (si:fab$b_rfm *fab*) si:fab$c_var) sets it to the code which says that the record format is variable-length records. Generally, fields of type byte and word are fetched/set as fixnums. longword and quadword fields are extracted into (freshly consed) simple-bit-vectors of the appropriate length; the setting operation replaces the bits of the structure (in the right position) with the bits of the object it is being set to, so the operation is invertible. See also the files `nil$disk:[nil.vmlisp]rmsstr.lsp` and `nil$disk:[nil.vmlisp]rmssub.lsp`.

The following routines create these structures:

si:make-fab

si:make-rab

si:make-nam

si:make-xab *code length*
Because xab structures vary, one must specify the xab code and its length. For instance,

```
(si:make-xab si:xab$c_fhc si:xab$c_fhclen)
```

creates a xab block used for hacking file header stuff.

Resources are defined for fab, rab, nam, and xab structures:

si:fab

si:rab

si:nam

si:xab *code length*

Because xab structures can vary in code and length, they must be supplied to the resource constructor. The appropriate constants are defined, of course. For instance,

```
(using-resource (si:xab xabdat si:xab$c_dat si:xab$c_datlen)
  . . .)
```

binds xabdat to a xab structure used for obtaining creation and revision dates.

19.8.7.2 RMS and Related Hacking

The following functions perform the obvious operation on their argument(s). A status code is returned.

si:rms$close *fab*

si:rms$create *fab*

si:rms$display *fab*

si:rms$erase *fab*

si:rms$extend *fab*

si:rms$open *fab*

si:rms$connect *rab*

si:rms$delete *rab*

si:rms$disconnect *rab*

si:rms$find *rab*

si:rms$flush *rab*

si:rms$free *rab*

si:rms$get *rab*

si:rms$nxtvol *rab*

si:rms$put *rab*

si:rms$release *rab*

si:rms$rewind *rab*

si:rms$trunc *rab*

si:rms$update *rab*

si:rms$wait *rab*

si:rms$read *rab*

si:rms$space *rab*

si:rms$write *rab*

si:rms$enter *fab*

si:rms$parse *fab*

si:rms$remove *fab*

si:rms$rename *fab1 fab2*

si:rms$search *fab*

si:rms$setddir *new-directory-specification*
Returns, and maybe updates, the RMS default directory. If *new-directory-specification* is nil, the RMS default is not modified; otherwise, it is set to *new-directory-specification*, which must be a simple string. This is a fairly direct interface to the SYS$SETDDIR system service.

If the return status is not normal (the value of si:rms$_normal), it is returned in place of the string.

si:trnlog *logical-name*
This performs the trnlog system service on the simple string *logical-name*. It returns a simple-string which is the translation, or nil.

19.9 Terminal I/O

The current NIL system contains a terminal stream which is translates general operations into terminal-specific display codes. Characters output to it (via the :write-char message or the write-char function) are interpreted as either display operations (e.g., carriage-return moves the cursor to the next line or wraps to the top of the screen, and clears the line it moves to), or as graphic characters (causing certain characters which are *not* graphic on typical ascii terminals to to be printed with certain conventions). Line wraparound is performed also.

The best way in which this can be accessed is with the cursorpos function, which is MACLISP compatible. In fact, the behaviour of the cursorposable tty stream emulates the behaviour of terminals under the ITS operating system, down to the terminal-width fencepost behaviour due to the use of the last column to hold the continuation character. (The "keyword character" argument to cursorpos, as defined by MACLISP, in fact derives from the second character in the escape

sequence used perform that cursor operation under ITS. Ah well.)

cursorpos &optional *arg1 arg2 arg3*

cursorpos is a MACLISP compatibility function, but it offers an interface to the display terminal code which may be safely and reliably used. Note that the arguments are interpreted in rather strange manners... As a general rule, cursorpos is supposed to return nil if it was not capable of performing that particular operation on the particular stream involved, t otherwise. It is *not* the case, however, that it may be used on non-terminal streams; that is an error.

(cursorpos)

returns the cursor position of terminal-io as a pair, (*vertical-position* . *horizontal-position*). Both positions are measured zero-origined, from the top-left corner of the screen. nil should be returned if the stream does not have a generally movable cursor.

(cursorpos *vpos hpos*)

Positions the cursor of the stream terminal-io at that position. Either *vpos* or *hpos* may be nil, in which case the current value is used.

Otherwise, the first arg to cursorpos should be a symbol (or character object), which may take an additional argument. The case is irrelevant.

C Clears the screen. The cursor moves to the "home" position (top left corner). On a non-display, this outputs a newline, so always succeeds.

A Fresh-line. In this instance, the fresh-line function (page 185) is preferred.

T "Top." The cursor is homed, moved to the top left corner.

Z Home down. Bottom left corner.

L Clear-to-end-of-line. From the current position to the right margin is cleared. The cursor does not move.

E Clear-to-end-of-screen. Current position to right margin, and all following lines, are cleared. The cursor does no move.

U Move Up a line. Wraps around the screen, does not scroll.

D Move Down a line. Wraps around the screen, does not scroll.

F Move Forward a character position.

B Move Backward a character position. If at the left margin, effectively does U then moves to the column to the left of the continuation-character column (i.e., it backs up the amount by which the cursor would have moved for a single-position printing character).

K Erase the character the cursor is over (this would not be the last one typed normally, see below:)

X B, then K. Simpleminded way to rubout the last character typed.

H *hpos*

Set the horizontal position to *hpos*.

V *vpos*
Set the vertical position to *vpos*.

] The insert-line operation

\ The delete-line operation

^ The insert-char operation

_ the delete-char operation

Additionally, one may specify a stream to **cursorpos** by giving that as the *last* argument. The atom **t** as a stream means, as with other printing functions, the terminal (the value of terminal-io). Note, however, that *no* stream *also* uses the value of terminal-io, rather than standard-output. Note also that the form

```
(cursorpos 't)
```

is interpreted as requesting the cursor position of terminal-io; to do a home-up, one must use some alternate form like

```
(cursorpos 'top)
(cursorpos #\t)
(cursorpos 't 't)
```

This strangeness is also MACLISP compatible...

19.9.1 Modifying the Terminal Characteristics

set-terminal-type *terminal-name*
Resets the terminal characteristics from the termcap entry found for *terminal-name*. See <not-yet-written>.

si:determine-and-set-terminal-type
This is the routine called on startup which either defaults or asks for your terminal type.

:init-with-termcap *termcap-struct*
termcap-struct is what would be returned by si:make-termcap.

19.9.2 Making More Terminal Streams

As noted elsewhere (page 180), **open** is what may be used to open terminal streams in NIL. The :type keyword specifies that a terminal stream is requested:

:display-tty
This produces a terminal stream just like that NIL starts up with. Additional options fed to open may be used to parameterize it; these are described below.

:cold-load
This produces a "raw" tty which has no display capability. It does perform some ascii-ification of non-display characters output, but performs no functions like line wraparound.

:tty
what does this do? is it left-over from something?

Interesting additional open keyword arguments which may be specified when opening a :display-tty:

:terminal-type *terminal-type-name*
The terminal capabilities description is obtained from the termcap entry for *terminal-type-name*. Since one of these is necessary, you might as well specify the right one rather than letting it default (the lookup in the database may still be necessary).

:cold-load-stream *cold-load-stream*
The first arg to open is ignored, and the innards of *cold-load-stream* (which *must* be a tty stream as created by the :cold-load open-type) is extracted to get to the real terminal, rather than opening a new one. (In the NIL loadup process, first the terminal streams are set to be a cold-load stream, and then later they are reset to be real display-tty streams using this. This is less important now than it used to be, but still comes in handy on occasion. It's not clear what use it might be to users.)

19.9.3 Display TTY Messages

In case you want to hack graphics on another terminal or something... See also the source code in `nil$disk:[nil.io]cursor.lsp`.

:write-char *char*
This is what implements write-char to a display terminal, with all the interpretation of *char* described earlier.

:oustr *string start count*
Note *start* and *count* are not optional. Using this results in a significant efficiency gain over individual :write-char messages, because the stream attempts to pass along as many block-mode operations as possible to VMS.

:write-raw-char *char*
This is not actually provided by si:display-cursorpos-mixin, but is *required* to be supported by flavors which mix that in. It is how si:display-cursorpos-mixin expects to get raw codes out to the terminal. Obviously, then, if you also wish to get raw codes to the terminal, you may use this message on display tty streams.

:raw-oustr *string start count*
Analogous.

20. Syntax

20.1 What the Reader Tolerates

I will defer detailed discussion of reader input and printed representation to the forthcoming COMMON LISP manual [1]. The LISP MACHINE LISP manual [12] also contains a good discussion of this. What will be presented here is basically a summary of what the current NIL reader accepts, utilizing COMMON LISP syntax.

Basically, the LISP reader reads characters from a stream and forms tokens out of them. Certain characters cause additional actions to take place; for instance, the (character will cause multiple (but possibly zero) expressions up to a matching) to be formed into a list. Some characters are significant only when they are the first non-whitespace character; the # dispatch macro character behaves like this: `#o403` is the integer 259 (`#o` meaning "read in octal"), but `foo#o403` is the symbol whose name is the string `"FOO#0403"`. Aside from these, the basic rule is that if a number can be formed from the characters of the token, it is; otherwise, it is a symbol. The sole exception is that the period character (.) is taken as a *cons dot*. If a character is preceded by a backslash (\), then *all* special significance is removed from it, including case translation, and it is treated as a token constituent.

Thus:

`foobar`	the atomic symbol `FOOBAR`
`foo\bar`	The atomic symbol whose print name is `FOObAR`
`259.259`	A floating point number. (Currently this is a double-float. But see the later discussion on floating-point syntax, section 2.1.2, page 4.)
`259\.259`	The atomic symbol with print name `259.259`.
`|FooBar|`	Vertical bars read a symbol with all characters (except for vertical bar and backslash) interpreted as constituent characters. So, this is the symbol whose print name is `FooBar`. Backslash may be used to include vertical bars and backslashes in the symbol.
`|Foo|Bar\\\|`	Similarly, the symbol whose print name is `Foo|Bar\`.
`259`	The decimal integer 259.
`259.`	The decimal integer 259. A trailing decimal point explicitly forces decimal notation, not floating-point.
`-259`	The negative decimal integer -259.
`+259`	The decimal integer 259.
`25\9`	The symbol 259.
`\259`	The symbol 259.
`|259|`	The symbol 259.

`1.0d-5` One times ten to the minus fifth power, as a double-float. See section 2.1.2, page 4.

`:foo` A colon in a token uses the characters on the left to name the *package* the following symbol is to be read into. The "null" package name means the package into which keywords are read.

`si:foo` The symbol `FOO`, read into the package named `SI` (the system-internals package).

`si:|Foo\|Bar\\|` The symbol `Foo|Bar\`, read into the `SI` package.

`foo#o403` The symbol `FOO#0403`.

If the token consists entirely of . characters (and none have been slashified), then it is illegal unless there is exactly one; that is a *cons dot*, which is only legal in list/cons formation. Thus, .foo. is the symbol whose print name is `.FOO.`, but ... is an error.

The primitive syntax for a cons is

(car . cdr)

In this notation, a list of items `a`, `b`, and `c` would be written as

```
(a . (b . (c . nil)))
```

List syntax allows us to "elide" a cons dot with a following cons; the dot is eliminated, as are the parentheses of the following cons:

```
(a b . (c . nil))
(a b c . nil)
```

and finally, because `nil` is the same as `()`,

```
(a b c)
```

The following characters terminate token formation, and do something special when encountered:

(Starts a list or cons, as described above.

) Terminates a list or cons, or some other construct which "matches" with parentheses, such as `#(`.

| Vertical bar terminates token formation.

" String syntax. The characters up to the matchine " form the string; " and \ may be included by preceding them with \.

' Reads the following expression, and "wraps" it with the function quote. Thus, 'foo reads as (quote foo).

` "Backquote". This is used for constructing expressions. Backquote is described later (section 20.1.1, page 216), and also in [3].

, Comma is used for performing substitution within backquoted expressions (q.v.).

The # character is a *dispatch macro character*. It reads (optional) digits as a numeric decimal argument, and then dispatches off of the following character. The following are defined:

#'expression
Wraps *expression* with function, similar to '. Thus, #'car reads as (function car).

#(x1 x2 ... xn)
Reads as a simple general vector *n* elements long (*n* may be zero), with those elements.

*#*bits*
Reads in as a simple bit vector whose elements are *bits* (the digits 0 or 1). Thus, #*100 is a simple-bit-vector of length 3; its element 0 is 1, and its elements numbered 1 and 2 are both 0.

#Brational
Reads *rational*, the syntax for a rational number (which, remember, may be an integer) in binary (base 2).

#Orational
Reads *rational* in octal.

#Xrational
Reads *rational* in hexidecimal.

#radixRrational
Reads *rational* in radix *radix*.

#font\character-or-character-name
Read an object of type character. The \ may be followed by either a single character (and then a delimiter), or by a token (read as described above), which is interpreted as the name of a character. The returned object will have a font attribute of *font*, which defaults to 0. #\a is lowercase a, and #\| is the character object for vertical-bar. A character name may have the names of character bits prepended to it. For instance, #\hyper-space is the character for space with the hyper bit. If the long form is used, the final character may need to be slashified to be interpreted correctly. For instance, #\control-a is uppercase a with the control bit, #\control-\a is lowercase a with the control bit, and #\control-(is an error (the left-paren delimits) which should have been typed as #\control-\(.

#C(real imag)
A complex number with real part of *real* and imaginary part of *imag*.

#nAcontents
Reads in as an array of rank *n*, with contents *contents* (see make-array, page 103). This does not work yet.

#S(name kwd1 val1 ... kwdn valn)
General structure syntax, for structures defined by defstruct. *name* is the name of the defstruct-defined structure. This does not work yet.

#+ conditional-expression expression-to-conditionalize
Read-time conditionalization. See the Maclisp Extensions Manual.

#- conditional-expression expression-to-conditionalize
Read-time conditionalization. See the Maclisp Extensions Manual.

#.expression
Reads in as the *evaluation* of *expression*.

#,expression

Load-time evaluation. If the expression is being read normally into NIL, this behaves like *#.*. However, if it is in a file being compiled, the compiler will arrange to have *expression* evaluated at load (i.e., vasload) time, when the containing expression is being constructed. This does not work yet in NIL.

20.1.1 Backquote

The *backquote* facility is used for constructing list structure from a template. Typically, most of the template is constant, and the most common use is when what is being created is LISP code, inside of macro definitions. In the simplest case, the backquote character (`) is used instead of quote ('), and, where substitution inside the template is desired, a form to be evaluated is preceded by a comma character. For instance,

```
(defmacro push (item place)
  `(setf ,place (cons ,item ,place)))
```

This definition reads in as code which is functionally equivalent to

```
(defmacro push (item place)
  (list 'setf place (list 'cons item place)))
```

Simple substitution is often not enough. If the comma in a backquoted expression is immediately followed by an atsign character (@), then, instead of simple substitution, the item being substituted is "spliced" into the containing list:

```
`(foo ,@bar mumble)
```

expands into code functionally like

```
(cons 'foo (append bar '(mumble)))
```

For instance, the macro

```
(defmacro always-returning-nil (&body body)
  `(progn ,@body nil))
```

produces expansions

```
(always-returning-nil)
  ==>  (progn nil)
(always-returning-nil (do-this))
  ==>  (progn (do-this) nil)
(always-returning-nil (do-this) (do-that))
  ==>  (progn (do-this) (do-that) nil)
```

The splicing is non-destructive on the expression being substituted, as if the splicing happens by use of append.

In NIL (but not COMMON LISP), a comma immediately followed by a dot instead of an atsign is treated similarly, *except* that NIL is free to use *destructive* operations for the splicing; that is, the splicing is performed as if by nconc (page 58) rather than append.

Backquoted expressions may be nested. They should be assumed to be expanded from the inside out. Consider the following macro definition:

```
(defmacro defhack (name fn &aux (place-var (gentemp)))
  `(defmacro ,name (,place-var)
     `(setf ,,place-var (,',fn ,,place-var))))
```

In this example, the expansion of the outer backquote expression causes the values of name, fn, andplace-var to be substituted in; this essentially means that we get (using italics now to show

the substitutions)

```
(defmacro name (place-var)
  '(setf ,place-var (,'fn ,place-var)))
```

Now, ",'*fn*" means that we substitute in the value of "'*fn*", which is of course just *fn*, which is what we want; on the other hand, we *do* want the value of *place-var* to be substituted in here, so it does not need that quote.

To see another way that this works, one can manually pseudo-expand the *inner* backquote in the original defhack definition

```
(defmacro defhack (name fn &aux (place-var (gentemp)))
  '(defmacro ,name (,place-var)
     (list 'setf place-var (list ',fn place-var))))
```

which indeed produces what we want. Generally, then, one uses ",." when one wants the substituted form to itself be evaluated and substituted with the expansion of the inner backquote expression, and ",',' when one wants to substitute in something which is "constant" with respect to the inner backquote expansion.

As one final extension to backquote, NIL allows its use with general vector syntax. This will probably be supported forever and ever, if for no other reason than the NIL compiler/code-generator emits instructions as vectors and constructs them that way.

```
'#(foo ,(+ 3 7) bar)  =>  #(foo 10 bar)
```

"Splicing" syntax (",@" and ",.") is not supported here. Neither NIL nor COMMON LISP support backquote use with array syntax (#A) or structure syntax (#S).

20.2 The Lisp Reader

20.2.1 Introduction

The NIL reader was designed to be incrementally extensible and to support the implementation of other languages in NIL. It also addresses some efficiency issues to to take advantage of, but to also hide, low-level considerations in disk and terminal I/O.

COMMON LISP and MACLISP compatible syntax extension functions are provided, along with readtables for the syntax of NIL, COMMON LISP,MACLISP, and CGOL. The definition of these is in the file [NIL.LISPIO]RTBSETUP.LSP.

Note that the default readtable has been set to one conforming to the COMMON LISP specification. The only significant difference between this and what MACLISP and LISP MACHINE LISP users have been using is that the syntax escape character is backslash, instead of slash. Some MACLISP programs we have seen are also using what is now the package prefix character ":" as a regular symbol-constituent character. If any of this presents a code porting problem, then set the readtable to one of the compatible readtables documented later, or specify a readtable in the modeline of the source files in question. For example:

```
;;-*-Mode:Lisp;Readtable:ML-*-
;; This code uses ":" and "/" as in maclisp.

;;-*-Mode:Lisp;Readtable:LM-*-
;; This code reads using the old lispmachine syntax.

C -*-Mode:Fortran;Readtable:Fortran-*-
C This would work if one defined a readtable for Fortran.

% -*-Mode:Lisp;Readtable:Cgol-*-
  This is lisp code in cgol syntax. Yow! %
define fib(x); if x<2 then 1 else fib(x-1)+fib(x-2)$

;; To get a maclisp readtable.
(setq readtable (si:lookup-readtable "ML"))
;; to get a lispmachine readtable.
(setq readtable (si:lookup-readtable "LM"))
```

20.2.2 Reader Extensions

For exotic or extensive reader extensions, see the documentation on the readtable, and how the various language readtables are set up, in [NIL.LISPIO]RTBSETUP.LSP and in [NIL.LISPIO]PARSER.LSP.

setsyntax *character type value*

This is a MACLISP compatibility feature, altering the syntax of the *character* in the current readtable. *type* may be macro, splicing, or single. If it is macro or splicing, then *value* is a function of no arguments which is invoked when the *character* is read.

setsyntax-sharp-macro *character type function* &optional *readtable*

This is also a MACLISP compatibility feature. *type* can be macro, peek-macro, splicing, or peek-splicing. *function* gets called with one argument, which is either null or the number between the # and the character.

20.2.3 Readtable

readtable *Variable*

The value of this variable is a datastructure that controls the behavior of the function read.

si:lookup-readtable *name*

Returns the readtable corresponding to the syntax named by the string *name*.

si:enter-readtable *name a-readtable*
Enters *a-readtable* giving the syntax for *name*. *name* may then appear as a readtable specification in the mode-line of a source-file.

create-readtable
Returns a naked readtable, with syntax for reading whitepace-delimited symbols.

si:add-number-syntax
Adds syntax for parsing numbers to the current readtable.

si:add-list-syntax &optional (*open #\(*) (*close #\)*)
Adds syntax for parsing lists to the current readtable.

si:add-package-syntax &optional (*char #\:*)
Adds syntax for specifing packages to the current readtable.

si:add-escape-char-syntax *char*
Makes *char* the syntax-escape-character in the current readtable.

si:add-prefix-op-macro *char operator*
Makes *char* a readmacro that returns a list of *operator* and the next thing read. For example,

```
(si:add-prefix-op-macro #/' 'quote)
```

20.2.4 Alternative Syntax

The CGOL syntax [11] is available by loading the file NIL$DISK:[NIL.LISPIO]CGOL.LOD. Further documentation is in the file NIL$DISK:[NIL.MANUAL]CGOL.TXT. The implementors do not recommend the extensive use of CGOL or any ALGOL-like syntax for LISP programming, especially in environments where program readability and editability are important long range considerations. However, some feel that syntactic variety taken in moderation is good for the soul.

cgolread
Reads a CGOL syntax expression.

cgolprint *expression*
Prints an s-expression lisp program in the CGOL syntax.

Another parser, for a language with syntax compatible with the symbolic algebra system MACSYMA [8], is available by loading the file NIL$DISK:[NIL.VAS]PARSER, which sets up a readtable named infix. The readmacro character "#$" has been set up to invoke this parser in the "NIL" readtable. One could then write the following:

```
(defun f (v a b x)
  #$(v[0,0]:COS((A-B)*X)/(2-2*(A-B)^2)+COS(V[1,1]*X),
     v[0,1]:COS((A+B)*X)/(2-2*(A+B)^2)-V[1,0]*V[0,0],
     v[1,0]:V[1,1]*V[0,0],
     v[1,1]:V[0,1]*V[1,0])$)
```

21. Debugging and Metering

21.1 Flow of Control

21.1.1 Tracing

trace *function*

trace enwraps the definition of *function* so that the arguments it is called with, and the values it returns, may be seen. *function* is not evaluated.

```
(defun f (x) (times x x)) => F

(trace f) => (F)
(untrace f) => (F)
(trace f) => (F)

(f 3)    ;printout:

#(1 :ENTER F #(3))
#(1 :EXIT F #(9))
```

The printout is a VECTOR. Its elements are:

[0] Recursion level for the given function

[1] :enter or :exit

[2] Name of the function.

[3] The *vector* of arguments, or the *vector* of return values.

Say that you wanted a breakpoint on entry to f. Then say

```
(defun f-bp (level direction name vector)
  (eq direction ':enter))

(trace (f (:break f-bp)))
```

Presently all trace options work this simple and functional way, the syntax of a trace option is (:keyword *predicate-function-to-call*), or simply :keyword which means the same thing as (:keyword t). Options are :noprint, :break, and :info.

One exception: (trace (f :menue)) enters a simple menue of various kinds of trace options.

21.1.2 Who does What, and Where

who-calls *symbol* &optional &key (*type :function*)
This searches all compiled-code modules to find those which reference the *type* value-cell of *symbol*. *type* may take on the values :function, :value, :local-function, or :local-value. It defaults to :function, thus finding all modules which call the function *symbol*. A *type* of :value would find all those modules which referenced *symbol* as a special variable. :local-function and :local-value (which should probably be called :lexical-anyway) aren't actually useful; they would only find uses where the references were not compiled away, and all local references are in the current compiler.

Someday this should be smart enough to do searching through all defined functions, including interpreted ones.

whereis *function*
function should be a symbol or a compiled-function. whereis returns the compiled-code module (the module-object) which defines *function*, or nil if that cannot be determined. It can only determine this for compiled functions.

Someday (i keep saying that don't i) there will be a more general mechanism, so that the source file can be determined for all "defined objects", such as those defined with defvar, defstruct, defmacro, etc. Until then, note the following function:

si:module-source-file *module*
This returns the name of the source file for the module *module*. The current implementation does this by looking at the vasl file from which *module* was loaded, so that file must exist on disk (with the same name).

apropos *string* &optional (*pkg package*)
This searches through *pkg* and all of its super-packages (see chapter 15, page 121), and returns a list of all of the symbols which contain *string* as a substring.

si:apropos-generate *fn arg* &optional (*pkg package*) (*superiors t*)
This function maps the function *fn* over all symbols which contain *arg* (a string or symbol) as a substring, in the package *pkg* (and its superiors, if *superiors* is not nil). si:apropos-generate uses mapatoms (page 122); it is possible that *fn* could be called on the same symbol more than once, although that will not happen very often in the current NIL implementation.

The apropos function is defined using this, by

```
(defvar *apropos-list*)
(defun apropos (arg &optional (pkg package) (superiors t))
  (let ((*apropos-list* ()))
    (si:apropos-generate
      #'(lambda (x) (push x *apropos-list*))
      arg pkg superiors)
    *apropos-list*))
```

One could write variants of this which test the symbol for specific properties, or with boundp or fboundp, and which print the results as they are computed rather than

accumulating them in a list.

21.2 Examining Objects

exhibit *object*

Invokes an interactive structure editor on the object. There is a "?" command to print out a command menu. The object is sent any of the following messages, :exhibit-self, :select-nth, :store-nth. See the definitions for built-in objects in "[NIL.SRC]EXHIBI.LSP".

describe *object*

Says a few things about the object.

21.3 Debug and Breakpoints

debug

Enters the debugger. Various commands, self documenting via the "?" command. Errors by default enter the debugger also. Note that in its current state, stack and argument information displayed requires an additional level of interpretation placed upon it for it to be correct. For example, local variable information currently shows simply the stack between call frames, including argument frames being computed and "dirty" (non-Lisp) data.

break *tag* &optional (*predicate-form t*)

break evaluates *predicate-form*, which defaults to t. If the result of this evaluation is not nil, then it enters a "break loop". ":bkpt *tag*" is printed out, and a recursive read-eval-print loop is entered. The prompt for reading says *n*>break>, where *n* is the number of nested break loops currently in force. Note that *tag* is not evaluated.

break is one of the older debugging tools around. It is not nearly as useful as it had once been, because in a LISP with lexically scoped variables, those values are not apparent from the break loop. In NIL what is probably more useful would be to insert explicit calls to (debug) in ones code, rather than to break. In the future, break will in fact do that, and its arguments will be a format-string and arguments for the format-string (see format, page 187).

***break** *value tag*

This is the internal version of break which evaluates both of its arguments normally. This is also how you can give a non-constant *tag* argument to the break loop. When break is changed, this function will go away.

21.4 Metering

21.4.1 Timing

timer *function* &optional (*loops 1*) *arguments*

Calls *function* with arguments *arguments* (a list or simple general vector), *loops* times, and prints out information on how much time was taken. Try, for example,

```
(timer #'cons 100 #(a b))
```

Needs some improvement to deal with function-calling and loop overhead; for that reason, this is not too useful with short fast functions.

COMMON-LISP defines two functions for obtaining compute and elapsed time. Both return integers in the same units, which can *not* be assumed to be fixnums. These units are units of *internal time*, which cannot be depended on to be the same in different COMMON LISP implementations. They may even differ from one NIL release to another.

internal-time-units-per-second *Constant*

This is an integer which is the number of "ties" of internal time per second. It happens that in NIL this is 100, because that is all the accuracy which VMS provides at the moment. It could change in the future, and its value should not be depended on.

get-internal-run-time

This returns the "run time" of the NIL process in internal-time-units-per-second units.

get-internal-real-time

This returns a measure of elapsed time in internal-time-units-per-second units. The time-base for this time measurement may not be depended on; only differences between the values of two successive calls should be used for anything.

runtime

This is the old name for get-internal-run-time. While it is the same name as the MACLISP function, it is incompatible—the NIL runtime function returns the run time in units of hundredths of a second.

elapsed-time
time

elapsed-time returns a measurement of elapsed time, in seconds, as a double-float. Two such quantities may be compared to determine elapsed time. The origin of this number may not be depended upon; in MACLISP it is the "system uptime"; in NIL it happens to be the double-float representation of the number of seconds since the origin of the Smithsonian time standard (local time). This quantity is only really accurate to hundredths of a second, even though it is potentially accurate to 100-nanosecond ties.

The synonym time is provided for MACLISP compatibility. This name should not be used in new programs, and should be changed in old programs, as the name time will be changed incompatibly by COMMON LISP. Also, there is a LISP MACHINE LISP time function which returns elapsed time, but as a fixnum in sixtieths of a second. The elapsed-time function will continue to exist as the functional equivalent to the MACLISP

time function, however.

si:pagefault-count

This returns the number of pagefaults taken by the process since process creation. Although this number is interesting to look at to see if the NIL is thrashing, it must be taken with several grains of salt due to the way VMS paging/swapping is performed. *[The following discussion should perhaps be somewhere else, under "performance considerations"?]*

The following points are especially of note. First, this number does not count the number of faults taken which involved fetching a page from the pagefile (or shared image file). Rather, it includes those "faults" for pages which still reside in physical memory, but are just not contained within the working set. Also, the overhead of doing this paging is charged to the process runtime.

Presumably, then, if one sets the working-set extent (the process parameter/user quota WSEXTENT) high, then the actual working set in use will approach the number of pages of the job which are resident in physical memory, and the count of pagefaults will better approximate the number of pagefaults for non-resident pages.

Note also the room function, which verbosely describes the virtual memory usage of the NIL, including the size of the living heap (i.e., how much consing has been going on).

The MACLISP-compatible status macro provides a gctime option which returns the runtime (in the same units as runtime does) which is the contribution to the process runtime by the garbage-collector. This is currently, of course, always zero. When the garbage-collector is available, there will be functions which parallel the above three, which will return the contributions to elapsed-time, runtime, and pagefaults by the garbage-collector. Note that the values returned by the above functions will always include the contributions by the garbage-collector.

21.4.2 Function Calling

The only type of function call metering which is available in NIL right now is a global database of how many function calls (and similar things) of various types have been performed since the NIL was first loaded up.

This number-of-function-calls metering is basically implemented by the NIL compiler. There are four tables 10 long; the four tables are for metering

function calls
Direct function calls. As in `(defun f (x) (g x))`.

funcalls
Simple funcalls.

sends
Calls to send. This does not (unfortunately) include lexpr-send.

applies
Compiled calls to apply (= lexpr-funcall).

The 10 entries in each table count the number of such occurences for zero through eight, and nine-or-more arguments. When the compiler compiles (say) a function call of two arguments, it

will sneak in an instruction like

```
incl w^c1$call_meter+2(s1p)
```

just before it does the actual function call. This sequence takes four bytes of code.

The intent of this type of metering is to measure how intensively various applications perform function calling, in order that we might be able to estimate how changes to the function calling sequence (such as modifications to the function entry code, function call-frame setup code, or even microcode support for either of those or the function call itself) might affect the performance of NIL programs. We have not yet actually done any measurements with these meters. However, in the event that they might be useful to people, the functions (which are somewhat dirty and kludgey) which read them are documented below.

si:get-call-meters

This returns an a-list of the values of the various calling meters. The a-list will be 4 long, and the first element of each of these lists is a keyword describing the type of call; the remaining 10 elements are the number of "calls" of that type for zero through 8, and (last) nine or more calls. The keywords are

:function
Direct function calls

:funcall
Compiled calls to **funcall**

:send
Compiled calls to **send**

:apply
Compiled calls to **apply (lexpr-funcall)**.

si:show-call-meters &optional (*meters (si:get-call-meters)*)

Prints out the meters.

si:subtract-call-meters *after-meters before-meters*

Returns a new "meters list" in which all of the numbers are the difference of the after and before values. All of the entries are assumed to be in the same order.

One could get a display of how much function calling (etc.) was going on by doing something like

```
(let ((before (si:get-call-meters)))
   (run-program)
   (si:show-call-meters (si:get-call-meters) before))
```

21.5 System Management

Included are some minimal utility functions for maintaining subsystems in NIL. These tools are not meant to be a comprehensive set, "addressing all the issues" as they say. Instead, they address some of the issues, have been found useful, and are used along with individual system specific procedures for maintaining systems including the editor and MACSYMA.

The practical working procedure on most programs goes something like the following: There are a set of source files that make up the program. One of these files defines a variable set to a list of these file names, and includes code for loading the files, creating needed package namespace(s), and peforming other functions as needed. Day-to-day works procedes in an incremental fasion, changes are made to the sources using the built-in editor, and these changes are tested and debugged using editor commands such as CONTROL-META-C (compile-defun, or <CONTROL-Z>-C), and META-Z (evaluate-defun, or <ESC>-Z) and other utilites in the system as needed. The editor, debugger, evaluator, and exhibitor are invoked many times during a days development cycle. From time to time during editing the changed files are saved of course, as a backup against environment crashes. At the end of the day, (or perhaps, during lunch hour, or after several days), a recompilation of the changed program files may be effected, using some of the functions documented in this section.

A somewhat parallel effort is the maintainence of a system that has "users." The same methodology as used in a development system is in effect: except that now the full-recompilation-cycle time may be months, and there is a definite target-environment which is to receive system changes in the form of "patch files." (See the documentation of the patch facility.)

Some additional functions documented here provide ways to find out something about how modules depend on one another.

21.5.1 An example

```
; This is an example "system-build" file.
(defparameter *my-files*
  '("USR:[ME.SYS]TOPLEVEL"
    "USR:[ME.SYS]UTILS"
    "USR:[ME.SYS]BASIC"))

(defvar *my-modules* ())

(defun load-my-system ()
  (setq *my-modules* (mapcar #'load *my-files*)))

(defun recompile-my-files ()
  (mapcar #'silent-comfile
          (mapcan #'(lambda (x)
                      (if (utils:source-need-compile? x)
                          (list x)))
                  *my-files*)))
```

```
(defun silent-comfile (x)
  (let ((compiler:*messages-to-terminal? ())
        (si:print-gc-messages ()))
    (comfile x)))

(defvar *my-undefined-functions-alist*
  ())

(defun find-my-undefined-functions ()
  (setq *my-undefined-functions-alist* ())
  (mapc
   #'(lambda (m)
       (if (typep m 'module)
           (utils:map-over-module-cells
            #'(lambda (module symbol key)
                (if (and (eq key :function)
                         (not (fboundp symbol)))
                    (let ((a (assq
                              symbol
                              *my-undefined-functions-alist*)))
                      (if a
                          (unless (memq module (cdr a))
                            (push module (cdr a)))
                          (push
                           (list symbol module)
                           *my-undefined-functions-alist*)))))
            m)))
   *my-modules*)
  *my-undefined-functions-alist*)
```

21.5.2 "Source (Re)Compilation"

utils:vas-source-file *filename*
Gets the exact name of source file from the object file, *filename*.

utils:source-need-compile? *source-filename* &optional *object-directory*
If there is no object file, or if the version of the number of the present source file is greater than the version number of the source from which the object was compiled then this function returns t.

utils:vas-source-needs-recompile? *filename*
Gets the source file name for the object *filename* and checks the version numbers.

21.5.3 Information in Modules

The exhibit function can be used to look at modules interactively.

21.5.4 Related Utilities

These are sometimes used to store information gathered during system programming, for example, "bug" cases, lists of undefined functions, sorted lists of special variables, etc.

utils:print-into-file *expression* &optional *filename*
Prints the *expression* into the *filename* which defaults to something generated in sys$scratch.

utils:pp-into-file *expression* &optional *filename*
As in utils:print-into-file above, but uses the pretty-print function.

21.6 Verification

verify *filename*
Expressions are read from the file named *filename* and fed into a normal read-eval-print loop. The *filename* is merged with a default specification of nil$disk:[nil.verify]. The input and results are printed both to the value of terminal-io and to an output file named *filename* with file type lis. (This is the closest thing to batch processing that we support.) There are various files in the [nil.verify] directory that are "verified" before a release of NIL is made. This function is also useful for making bug reports that are easy to deal with. For example, say that you found that multiplication of 2.2 and 3.3 did not work, you could then make a file multbug.lsp containing the following:

```
(si:print-herald)
;; multiplication bug
(errset (times 2.2 3.3))
```

Then run the verify function on this file and send it and the output in your bug note. An errset would be needed around any expression that would otherwise cause a fatal error.

22. Errors

Errors in NIL work by signalling *error conditions* using **signal**. The specifics of this are going to be changing in various ways; however, the basic interfaces for "creating" errors can hopefully be kept static, at least insofar as the functions can be made to accept and interpret arguments upwards-compatibly.

cerror *proceedable restartable condition-name format-string args*

This is what needs to be used to signal correctable errors. For an error to be correctable (in the current scheme), one uses **cerror** and gives a non-null *proceedable* argument. The *restartable* argument has to do with saying that the error can be "restarted" (i.e., something gets tried over again) by throwing to a tag with name **error-restart**: this is hardly, if ever, used, and will probably be obsolete quite soon.

condition-name is the name of the condition being signalled; it is typically, although not necessarily, a keyword.

format-string is a string suitable as an argument to **format** with extra arguments of *args*; that is how the error message is produced. There are, however, conventions on what particular arguments mean for particular conditions; some of them are described below.

At some point, the error system and how errors (and non-error conditions) are signalled will all change. It should be the case, however, that vanilla uses of **cerror**, especially those with the error conditions listed below, will continue to work.

check-type *place type-specifier* &optional *description*

check-type expands into code which verifies that the value of *place* (which must be a *place* suitable for use with **setf**—see page 38) is of the type *type-specifier*. If it is not, then an error is signalled and *place* is updated to the newly-supplied value. Note that *type-specifier* is not evaluated.

description is a string which describes the type; for instance, "an integer", "a prime number greater than 403". If it is not supplied, then **check-type** will attempt to construct such a phrase from *type-specifier*; however, the current implementation barely tries at all, so for now it is probably best to specify it.

For example,

```
(defun oddp (x)
  (check-type x gaussian-integer "a guassian integer")
  (if (typep x 'complex)
      (if (not (logbitp 0 (realpart x)))
          (logbitp 0 (imagpart x))
          (not (logbitp 0 (imagpart x))))
      (logbitp 0 x)))
```

check-arg *place predicate* &optional *string*

check-arg is an older form of check-type. *predicate*, if it is atomic, is a function of one argument to be used to test the value of *place*:

```
(check-arg x fixnump "a fixnum")
```

If it is not atomic, then it is a form to be evaluated to test to see if the value of *place* is acceptable:

```
(check-arg x (typep x '(signed-byte 8))
           "an eight-bit signed byte")
```

Even if it were to try, check-arg would have even more trouble constructing a descriptive string on its own than check-type, so it is recommended that *string* be supplied.

For example,

```
(defun fact (x &aux (ans 1))
  (check-arg x (typep x '(integer 0 *))
             "a non-negative integer")
  (do ((i 2 (1+ i))) ((>= i x) ans)
    (setq ans (* i ans))))
```

Here are some of the interesting and well-formed error conditions defined in NIL right now, and the arguments they expect. (Note that extra arguments may always be given.)

:wrong-type-argument *type-name losing-object*

This has to be the most common condition used in NIL. *type-name* is the name of the type of object which was expected, such as number, and *losing-object* is the object. (The *type-name* is not currently used for anything, and lots of code just puts a fairly random symbol there.) The value returned is used in place of the value of the wrong type. For example, a subroutine which arg-checks for fixnum:

```
(defun si:require-fixnum (x)
  (do () ((fixnump x) x)
     (setq x (cerror t nil :wrong-type-argument
                     "~*~S is not a fixnum"
                     'fixnum x))))
```

Wrong-type-argument checks are so common that it is better to subroutinize or macroify them rather than writing out loops. See, for instance, check-arg (page 230).

:unbound-variable *variable*

variable was not bound. Returning a value uses that as the variable instead.

:undefined-function *name*

name is not defined as a function. Returning a value uses that as the function name instead.

:wrong-number-of-arguments *random*

This is handled spastically right now, will probably be superceded by something else. At least when called from compiled code, returning a value causes that value to be returned as if from the losing call.

:invalid-form *form*

form was not meaningful to eval. The return value is evaluated in place of the bad form.

:io-lossage *description form*

Sort of a catch-all for random I/O errors. *description* is a string describing the error, and *form* is the form which produced it; for instance

```
(delete-file "[foo]bar.baz")
```

might signal a :io-lossage error with a *description* of the string

```
"%RMS-E-DNF, directory not found"
```

and a *form* like

```
(delete-file #<pathname "sys$sysdevice:[foo]bar.baz">)
```

:symbolic-constant-update *symbol* &optional *value old-value*

An attempt to update a value cell which is supposed to be constant was detected. In principle, this can happen to any type of value cell (i.e., special or lexical value or function cells), although in practice only special value cells are created in this manner. The text of the error message gives the context (i.e., makunbound, set, variable binding, etc.). The *value* and *old-value* are given when convenient for the code generating the error to do so.

Continuing from this error continues without having performed the set, binding, makunbound, or whatever. The revised error system will probably offer a menu of which this is one option, and doing the operation is another.

:symbolic-constant-redefinition *symbol old-value new-value*

This is somewhat similar to :symbolic-constant-update, but is signalled by (the primitive used by) defconstant (page 24) when the symbol being defined as a constant has a value already which differs from the one being assigned. Returning from the error ignores the value returned and proceeds to modify the value of the symbol.

23. Environment Enquiries

23.1 The Host Environment

lisp-implementation-type
This returns a string which is the name of the LISP implementation. In NIL, this string is "VAX NIL".

lisp-implementation-version
This function returns a string which describes the versions of the systems loaded in the NIL. In NIL, this is the same as calling (si:system-version-info t) (see page 282 for more details).

machine-type
This returns a string which names the type of machine the NIL is running on, such as "DEC VAX-11/780".

machine-instance
Returns a string which is a (supposedly) unique string naming the machine, such as "MIT-CORWIN". This name will normally be the same as that used as the "host" name by the pathname code.

host-software-type
The software type of the machine. In NIL, this is always "VMS".

short-site-name
This returns a short name of the site, for instance "MIT AI Lab". If this was not set up in the NIL site parameters file, it will default to what is returned by the machine-instance function.

long-site-name
This returns a long name for the site, such as "MIT Artificial Intelligence Laboratory". If this was not set up in the NIL site parameters file, it will default to the short site name (above), or the machine instance.

23.2 Maclisp-Compatible Status Enquiries

(status date)
returns a list of the year (modulo 100), month, and day of the month.
(status daytime)
returns a list of the current hour, minute, and second.
(status dow)
returns a symbol which is the full name of the current day of the week (in English, sorry). The package the symbol is interned in is probably system-internals; perhaps it should be the keyword package, or the value should be a string anyway.

(status gctime)
Returns the runtime contribution of the garbage-collector. This is, of course, always 0 in the current NIL.

The first four of these may all be obtained by use of the functions in section 23.5, page 234.

23.3 Privileges

get-privileges
This returns a list of keywords naming the privileges the NIL currently has enabled. The keywords are listed below.

set-privileges &rest *keywords-and-values*
This takes as arguments alternating privilege keywords and flags: if the flag is nil, the corresponding privilege is disabled, otherwise it is enabled (if that is possible). set-privileges returns a list of the privileges which were enabled when it finished, just as get-privileges does. For instance,

```
(set-privileges :sysprv t :bypass nil)
```

(attempts to) turn on the sysprv privilege, and turn off the bypass privilege.

The privilege keywords defined are
:cmkrnl
:cmexec
:sysnam
:grpnam
:allspool
:detach
:diagnose
:log_io
:group
:acnt
Note that for this privilege, the name used in VMS is noacnt; however, NIL uses acnt because that is the name used by DCL.
:prmceb
:prmmbx
:pswapm
:altpri
The internal name for this is setpri, however DCL uses altpri, so NIL does also.
:setprv
:tmpmbx
:world
:mount
:oper
:exquota
:netmbx
:volpro
:phy_io

:bugchk
:prmgbl
:sysgbl
:pfnmap
:shmem
:sysprv
:bypass
:syslck

23.4 Memory Usage

room &optional *stream*

room prints out a fairly verbose English description of the virtual memory usage of the NIL, and some related information. In particular, one of the things it tells is an estimation of how much space is left for expansion of the *living heap*, which is where all ordinary consing is performed.

Note also the si:pagefault-count function, page 224.

23.5 Time and Date

(See also section 23.2, page 232.)

NIL provides a set of functions for manipulating dates and times. A date and time is represented in one of two ways: by a *universal-time*, which is an integer number of seconds from 00:00 January 1, 1900 GMT, and as a *decoded time*, which consists of the following components:

seconds
minutes
hours
date
month
The one-origined month number: January is 1, etc.
year
The full year, e.g., 1983.
day-of-week
The zero-origined day of the week: 0 is Monday, 1 is Tuesday, and 6 is Sunday.
daylight-savings-time-p
t or nil, depending on whether the decoded time is daylight savings time.
timezone
The number of the timezone: 0 is Greenwich time, 5 is Eastern time, -2 is Eastern European time, and -4.5 is Indian Standard time.

This decoded time is not represented in any datastructure, but rather is returned as multiple values by functions which decode universal-time, or passed as arguments as other functions.

23.5.1 The Main Functions

The functions in this section are defined by COMMON-LISP, and all may be referred to without any package qualification.

get-universal-time
This returns the current time in universal-time format.

get-decoded-time
This returns the current time, decoded, as 9 values: the current second, minute, hour, date, month, year, day-of-week, daylight-savings-time-p, and timezone. It is effectively

```
(decode-universal-time (get-universal-time))
```

decode-universal-time *universal-time* &optional *timezone*
timezone defaults to the current timezone (see page 238). This decodes *universal-time* with respect to *timezone*, and returns 8 values: the second, minute, hour, date, month, year, day-of-week, daylight-savings-time-p, and timezone.

encode-universal-time *seconds minutes hours date month year* &optional *timezone*
timezone defaults to the current timezone. This encodes the given date and time for the timezone, and returns a universal-time.

23.5.2 Printing Dates and Times

The following functions are not globalized, but must be referred to with the time: package prefix. Most of them are the same as those provided by LISP MACHINE LISP. All of them take a *destination* argument, which may be a stream, t, or nil. t means use the value of standard-output; nil means return the result as a string. Note that this is compatible with the way the format function (page 187) interprets its *destination* argument, *not* the way the regular NIL printing functions do. The *destination* always defaults to t, meaning print the output on standard-output.

time:print-time *seconds minutes hours day month year* &optional (*destination t*)
This prints the specified date and time in brief format, to *destination*. The format used is "10/03/83 23:02:59"—the month number, the day in the month, the year (modulo 100), and the time.

time:print-current-time &optional (*destination t*)
This does time:print-time of the current time.

time:print-universal-time *universal-time* &optional (*destination t*) *timezone*
This effectively does decode-universal-time on *universal-time* and *timezone* (which defaults to the current timezone), and then time:print-time of the results to *destination*.

time:print-date *seconds minutes hours date month year day-of-the-week* &optional (*destination t*)
This prints the date in a format like
```
Friday the seventh of October, 1983; 11:02:59 pm
```

time:print-current-date &optional (*destination t*)
Prints the current date, in time:print-date format.

time:print-universal-date *universal-time* &optional (*destination t*) *timezone*
Uses time:print-date to print the decoding of *universal-time* and *timezone*.

time:print-brief-universal-time *universal-time* &optional (*destination t*) *reference-time*
This prints *universal-time* in a format like 10/07/83 23:02. However, portions of it will be omitted if they are the same as the corresponding components of the universal-time *reference-time* (which defaults to the current time). In particular, if the year is the same, it is omitted, and if the month and day are also the same, the entire date is omitted. So depending on the reference time, one could get 10/07 23:02 or even just 23:02.

time:print-universal-mail-format-date *universal-time* &optional (*destination t*)
Prints the date and time specified by *universal-time* in "mail format". This is a format which conforms to that specified in RFC822, "Standard for the format of ARPA Internet Text Messages" [6]. It looks like
```
Fri, 7 Oct 83 23:02:59 EDT
```

time:print-current-mail-format-date &optional (*destination t*)
Prints the current date and time, using time:print-universal-mail-format-date.

23.5.3 Namings

The time package contains a small database of names of months and days of the week, in various formats. The format is split along two dimensions: the *mode*, for instance :long, :medium, or :short, and the *language*, such as :english. There is a global default for each of these:

time:*default-language* *Variable*
The default value of this is :english.

time:*default-mode* *Variable*
The default value of this is :long.

The mode and language are combined (by lookup) into a composite symbol for which the lookup of the actual text format is performed. For the following functions, one may specify this composite symbol, such as :long-english, just the mode (:long), in which case the default language is used, or just the language, in which case the default mode is used. The three predefined modes are :long, :medium, and :short, and there are entries for the languages :english, :french, :german, :spanish, and :italian. Not all mode/language combinations are supported; in particular, :medium is treated as a special case, such that if it is not found, :short is tried instead. The :short mode typically implies the short standard abbreviation; this is usually three characters, however it is two for :german.

time:month-string *month* &optional *modespec*

Returns a string naming the month, in the format specified by *modespec*. Assuming the initial default values for time:*default-language* and time:*default-mode*,

```
(time:month-string 12)          =>  "December"
(time:month-string 12 :short)   =>  "Dec"
(time:month-string 12 :spanish) =>  "diciembre"
(time:month-string 9 :medium)   =>  "Sept"
(time:month-string 9 :short)    =>  "Sep"
```

time:day-of-the-week-string *day-of-the-week* &optional *modespec*

Similar:

```
(time:day-of-the-week-string 1)                 =>  "Tuesday"
(time:day-of-the-week-string 1 :short)          =>  "Tue"
(time:day-of-the-week-string 1 :medium)         =>  "Tues"
(time:day-of-the-week-string 1 :german)         =>  "Dienstag"
(time:day-of-the-week-string 1 :short-german)   =>  "Di"
```

time:mode-language-fetch *modespec hash-table what*

This is the primitive which canonicalizes the mode, language, or composite symbol into a composite symbol, and looks that up in *hash-table*. *what* is just a descriptive string for error reporting, e.g., "Month name". The two above functions use this function to retrieve vectors from the following hash tables, which are then indexed into to get the names.

time:*month-strings* *Variable*

This is a hash table associating the composite mode/language symbols (e.g., :long-english, :short-german) with vectors of the strings for the given months. The vectors are zero-origined, so 1 must be subtracted from the month number before the aref (or sgvref) is performed. The vectors are always simple general vectors.

time:*day-of-the-week-strings* *Variable*

The hash table for day-of-the-week names.

Thus, time:month-string is defined (modulo error checking) as

```
(defun month-string (month &optional modespec)
  (sgvref (mode-language-fetch
            (or modespec *default-mode*) *month-strings*
            "Month name string")
          (1- month)))
```

Fetching the vector and indexing into it can be useful if the vector is going to be used repeatedly, for instance for printing a directory listing.

23.5.4 Timezones

timezone *Variable*

The value of this is the current timezone. It should be initialized in the NIL site initialization file to the correct value; otherwise, it will be 5, which is Eastern time.

time:timezone-string &optional *timezone daylight-savings-p*

This returns the canonical three-character string for the timezone, or nil if the name is not known to NIL.

```
(time:timezone-string 5 nil)    => "EST"
(time:timezone-string 5 t)      => "EDT"
(time:timezone-string 0 nil)    => "GMT"
```

timezone defaults to the current timezone, and *daylight-savings-p* to the value of (time:daylight-savings-p).

23.5.5 Miscellaneous Other Functions

time:leap-year-p *year*

Returns t if *year* is a leap year, nil otherwise.

time:daylight-savings-p

Returns t if it is currently daylight savings time, nil otherwise.

time:month-length *month year*

Returns the length of the month month in *year*.

time:daylight-savings-time-p *hour day month year* &optional *timezone*

Returns t if the specified day/time is in daylight savings time in the specified timezone (which defaults to the current timezone), nil otherwise. (The specified hour is assumed to be standard time.) Note also section 23.5.6, page 239.

time:day-of-the-week (*day month year*)

Returns the number of the day of the week of the specified date.

time:verify-date *day month year day-of-week*

If *day* is a day within *month* which fell on *day-of-week*, this returns nil; otherwise, it returns a string describing the conflict.

time:moonphase &optional *universal-time*

Returns the phase of the moon decoded into five values: the (zero-origined) number of the quarter, the day, hour, minute, and second. The quarters are (from 0 to 3) new moon, first quarter, full moon, and last quarter. This is presumably the uncorrected mean longitude.

time:print-moonphase *quarter day hour minute second*
Prints the phase of the moon in a format like "FM+2D.3H.29M.57S.". The first field will be one of "NM", "FQ", "FM", or "LQ". Zero-suppression is performed on the other fields; for instance, "FM+2D.29M.".

time:print-current-moonphase &optional (*destination t*)
Similar.

Note also the get-internal-run-time and get-internal-real-time functions, on page 223.

23.5.6 Variations in Daylight Savings Time

The timezone objects NIL uses to store information on particular timezones include a function which may be used to tell if a particular hour, day, month, and year (when interpreted as standard time) is in daylight time. Most of the initial timezone entries have nil in this component, which is taken to mean that the zone never uses daylight time. It may be necessary to patch this up; some functions and hacks are presented here to aid in this.

time:zoneconv *timezone*
If *timezone* is a timezone object (an object of type time:timezone), it is returned. If it is a number, then an existing timezone entry for that offset in the list time:*timezones* is returned. (Otherwise an error is signalled.)

If the daylight-savings-time predicate NIL supplies is not appropriate for the site (or is just plain incorrect), it can be patched by

```
(setf (time:timezone-dst-rule
        (time:zoneconv timezone-offset))
      'beandorfian-daylight-savings-time-p)
```

where *timezone-offset* is the offset from GMT of the timezone in question, and beandorfian-daylight-savings-time-p the predicate which tells whether any particular hour in a particular year is in daylight savings time.

time:regular-american-daylight-savings-time-p *hour day month year*
This returns t if *hour*, *day*, *month*, and *year* is between 1 am of the last Sunday in April and 1 am of the last Sunday in October (see below).

time:last-sunday-in-april *year*
Returns the day of the month on which the last Sunday falls in April.

time:last-sunday-in-october *year*
Returns the day of the month on which the last Sunday falls in October.

Computations like "last Sunday in April" are fairly easy to perform. To find the last *dow* day in a month, find the day-of-the-week of the last day of the month, and subtract the number of days it takes to get back to the desired day of the week. For instance,

```
(defun last-sunday-in-april (year)
  (-& 30 (\\& (-& (+& 7 (day-of-the-week 30 4 year)) 6) 7)))
```

The 30 is the last day in April, 4 the month April; 6 is Sunday, and the 7s and arithmetic with them doing modulus stuff. Obviously the arithmetic can be simplified to

```
(-& 30 (\\& (+& 1 (day-of-the-week 30 4 year)) 7))
```

Because this actually finds the last day of a particular weekday on or before some particular day, it could be used to find the *first* Sunday in a month by substituting 7 for the last day of the month, and the *second* Sunday by substituting 14.

23.5.7 Internal Conversions

Internally, NIL time measurements convert the VMS 64-bit integer (number of 100-nanosecond tics) into a double-float number of seconds. The following subprimitives are provided for the conversion:

%convert-time-to-d_float *simple-bit-vector*

simple-bit-vector should be a simple bit vector 64 bits long. It is assumed to contain the binary representation of a VMS absolute time. This quantity is returned as a double-float in seconds, accurate to at least hundredths.

%convert-d_float-to-time *double-float simple-bit-vector*

The inverse of %convert-time-to-d_float; the VMS representation is stored into *simple-bit-vector*.

time:convert-vms-time-to-universal-time *vms-time*

vms-time is a double-float absolute time representation, as returned by %convert-time-to-d_float (above) or elapsed-time (page 223). This converts it to a universal-time. This is used not only by get-universal-time, but also for the conversion of dates in the filesystem (as by, for example, the file-creation-date and directory functions) into universal time format. See the following section for environmental factors which affect the interpretation of *vms-time*.

23.5.8 Brain Damage

The format of universal-time (which is defined by COMMON LISP) is the same as that specified by the ARPA Internet Time Protocol [10] as a "site-independent, machine-readable date and time." Because it is always an offset based on GMT, it is unaffected by local time zones; because this is always a standard time, it does not get confused by local variations (daylight savings time). Because it is absolute, the local time can be computed from it by knowing only the timezone and whatever rules and information are necessary to determine when daylight savings time takes effect. (The typical way this is done is a fairly simple perpetual-calender-like computation to see if the date and time falls between the last Sunday in April, and the last Sunday in October.)

The absolute time used by VMS is an offset (in 100-nanosecond units) from 00:00 November 17, 1858, in the current local time. No distinction is made between standard and daylight time. While this appears to have the benefit of simplifying life by eliminating daylight-savings-time computations, it has the side effect that to change to and from daylight time, the base time must be modified, *thus changing the only quantity useful for measuring absolute times*. There is also the fact that, in local time, there is an hour which occurs twice (when daylight time is reset to standard time), hence for that hour one cannot determine whether one is actually in daylight or standard time.

Because NIL must use the VMS time to determine the absolute time, get-universal-time (and all things which return results based on that quantity, which is just about all the time and date functions in this section) can return an incorrect value. Also, the quantities returned by elapsed-time (page 223) and get-internal-real-time (page 223) are based on this, meaning that if a shift in or out of daylight time occurs and the VMS base time is changed, elapsed time measured around the shift will be incorrect. There is a variable provided for telling NIL about two potential system-wide conventions for bypassing this sort of lossage.

time:*system-time-kludge* *Variable*

This may take on one of three values.

nil This is the default. It means that the VMS base time is always the local time (potentially daylight time), and hence subject to complete lossage.

:standard
This means that the VMS base time is always based on the standard time for the current timezone.

:gmt
This means that the VMS base time is GMT (implies never daylight time).

Either of the last two options eliminate the ambigous-hour and incorrect-elapsed-time problems which might be suffered otherwise. Of course, they mean that all the VMS utilities which show dates and times won't work quite right. Oh well.

The above variable is examined for conversion of all VMS internal times into universal time. Additionally, for interpretation of the *current* time, if the above variable is nil, then the SYS_STTC logical name is examined. (This is a logical name used/defined by Chaosnet software distributed by Symbolics, Inc. However, as with other similar logical names used by NIL, the system manager could put its definition in the system startup file.) If this logical name is defined, then if it translates to the string "ON", then daylight savings time is assumed to be in effect; translation into anything else is taken to mean that standard time is in effect. If the logical name is undefined, then the time is examined to see if it might be in daylight time (a computation which can be wrong near the times when the change is effected).

23.6 Job/Process and System Information

NIL provides a simple interface to the `$getjpi` and `$getsyi` system services. Both of these are simplified somewhat over what is provided by the system services themselves, in that they will obtain values for only one item at a time.

si:getjpi-value *item-code*

This returns a non-negative integer value for the JPI item designated by *item-code*. *item-code* is the integer code for the particular item in question (the same as used by VMS); the si:%jpi macro may be used to get the code for known items by name.

The value returned can have up to 64 bits; it will be a bignum if necessary, and will never be negative (the 64 bits are returned unsigned). If `$getjpi` is unhappy in some way, an error is signalled.

si:getjpi-string *item-code*

This returns a string containing the bytes returned by a **$getjpi** of *item-code*. An internal buffer is used which limits the length of the string to some moderate number of characters (256?).

si:%jpi *name*

name may be a symbol or string naming a JPI item. It is *not* evaluated; the name lookup is performed at macro-expansion (e.g., compile) time. The names are those used as the suffix names for the VMS symbols which designate the item codes; for instance, (si:%jpi :imagename) returns the item code to obtain the executing image name (this will be the NIL kernel image).

si:getsyi-value *item-code*

Similar to si:getjpi-value, but for the **$getsyi** system service.

si:getsyi-string *item-code*

Similar to si:getjpi-string, but for the **$getsyi** system service.

si:%syi *name*

Similar to si:%jpi.

The reason the item names for these things are accessed with these little macros is that these things are used fairly infrequently, so don't warrant having much space wasted in the NIL for assignments of constants to the names or anything. So, the names are stored as strings in tables. The names can be written as keywords in the calls to make them more stylistically consistent with other functions such as set-privileges.

The options available, and their meanings, can normally be obtained from online VMS help by doing

```
help sys $getjpi
```

and

```
help sys $getsyi
```

24. Compilation

Compilation is essentially the process of translating from one specification into another which is presumably more efficient, and probably more low-level in some respects. The NIL compiler translates LISP code into the VAX instructions necessary to execute that code; some of these instructions may perform the task directly, while others may call functions or NIL kernel subroutines to do it. In any event, the end result is intended to exclude the NIL interpreter from the running of the program. The NIL compiler does not output MACRO32 code or anything of the sort; rather, it represents the code and data itself, and assembles the code, LISP objects referenced by the code, and whatever other information is needed to help construct that data, into a *compiled code module*. This is what the **module** data type represents.

When the NIL compiler compiles a file, it first establishes the proper environment for the compilation, as specified by the file's attribute list (described in section 19.8.6, page 205). That done, it reads and processes each form in the file. What it does with each form depends on what the form is.

`(proclaim` *dcl-spec*`)`
: If the value of the form *dcl-spec* can be determined at compile-time, then the compiler will attempt to assert that proclamation for at least the duration of the compilation. See **proclaim**, page 46.

`(declare {`*dcl-spec*`}*)`
: The *dcl-specs* are processed, and take effect for at least the remainder of the compilation. **declare** in this context is sort of like **proclaim** with multiple unevaluated arguments; however, this usage (non-local declarations) of **declare** is being phased out and subsumed by **proclaim**. It will be probably be supported indefinitely, and will also be accepting certain MACLISP-style declarations which **proclaim** will not.

`(eval-when` *kwd-list forms...*`)`
: If **compile** is a member of *kwd-list*, all of the *forms* are evaluated then and there. Then, if **load** is a member of *kwd-list*, the *forms* are recursively processed.

`(progn` *forms...*`)`
: *forms* are recursively processed. Note that this is identical to **(eval-when (load)** *forms...*`)`, and upwards-compatible with the MACLISP **(progn 'compile** *forms...*`)` **hack.**

`(compiler-let` *bindings forms...*`)`
: Establishes the bindings specified by *bindings*, then recursively processes *forms* in that environment. See page 245.

`(defun` *name arglist etc...*`)`
`(defun (`*name property-name*`) arglist etc...)`
: The function is compiled, and the appropriate assignment (function cell or **putprop**) will be put in the compiler's output file.

`(defmacro` *name etc...*`)`
`(macro` *name lambda-list etc...*`)`
: The specified macro definition for *name* is added to the compilation environment. Then, the macro is compiled, so will be there when the compiled output file is loaded. It may not be depended on that *name* is defined in the LISP environment itself; only that code

being compiled will have the macro run to produce the expansion. If it is necessary for the macro to actually be defined (perhaps in order for it to be called from within other macros, as opposed to just expanded in code being compiled), then the **defmacro** or **macro** form should be enclosed within an **(eval-when (compile ...) ...)** form.

`(defun` *name* `macro` *lambda-list etc...*`)`
The compiler whines at you and turns this into **macro**. This is provided to catch old MACLISP code which should probably use **defmacro** (or at the very least **macro**) instead.

`(defun` *name* `fexpr` *lambda-list etc...*`)`
The compiler barfs at you and turns this into a special form definition. Calls to it from compiled code will not work. If *name* is only around for users to call interactively, however, it might just function properly. This is provided only to brute-force through some MACLISP programs. Generally, special forms or *fexprs* should be rewritten as macros.

`(defun` *name atom etc...*`)`
The function definition is assumed to be a MACLISP **lexpr**. It is transformed appropriately, and compiled, after the compiler gets through giving you a hard time.

`(defflavor` *etc*`)`
Code to perform the flavor definition at load time is generated. Additionally, declarative information is added to the compilation environment so that **defmethods** will compile correctly, and the routines defined for the **:outside-accessible-instance-variables** can be compiled correctly.

`(defmethod` *etc*`)`
Compiles the code for the defmethod.

anything-else
If *anything-else* is a macro call or a call to a function the compiler has special rewrite information about, the macro expansion or rewrite is performed, and the compiler tries again. Otherwise, *anything-else* will be evaluated at load time. Currently this is done by compiling the expression, which eliminates load-time dependencies upon macros.

There are various other forms which implicitly do compile-time processing by virtue of how they are defined, rather than by the compiler recognizing them specially. For instance, **defstruct** (currently) by default expands into the appropriate macro, function, and data definitions, with appropriate use of **eval-when**. For this reason, the above list cannot be taken as being all-inclusive.

compile-file *input-file* **&key** *package set-default-pathname output-file default-pathname defaults*
Compiles *input-file*, storing the vasl file in *output-file*.

If *package* is specified, then the file is read in in that package, in spite of what might be specified in the source file property list (see section 19.8.6, page 205).

The *input-file* is defaulted from the *default-pathname* if any, and then from *defaults*, which should be a pathname defaults. If *set-default-pathname* is not nil, then the default pathname of *defaults* (which defaults to the value of ***load-pathname-defaults***, page 201) is updated.

output-file defaults to *input-file* with a vasl file type (VMS extension of VAS).

By special dispensation (to those of me who cannot get out of the habit of using this feature), if exactly two arguments are given to compile-file, then the first is the input file and the second is the output file. This is typically used like

```
(compile-file "[nil.io]iofun" "[nil.vas]")
```

The COMMON LISP definition of compile-file accepts only the keyworded argument *output-file*.

comfile ...

Alternate name for compile-file. COMMON LISP defines the use of the name compile-file, but not comfile, for whatever that is worth.

compile *function*

Compiles the interpreted (and in-core) definition of *function*, in-core. That is,

```
(defun fact (x)
  (if (zerop x) 1 (times x (fact (sub1 x)))))
(fsymeval 'fact)
  => #<Interpreter-Closure FACT 0 123456>
(compile 'fact)
(fsymeval 'fact)
  => #<SUBR FACT>
```

In fact, what happens is that the *function* is compiled into a temporary file and that file loaded; this is somewhat of a kludge at the moment, but is done because of limitations of the NIL module assembler. Note in this regard user-scratchdir-pathname (page 200) and *scratch-pathname-defaults* (page 202).

The above described calling sequence is the intersection of what is provided now, and what COMMON LISP defines for compile. The NIL idiosyncracies will be removed eventually.

compiler-let ({(*var val*)}*) {*form*}*

When evaluated, this binds *dynamically* each *var* to the evaluation of each *val*, and then evaluates the *forms* in that environment. Syntactically, this is like let and let* (page 26). At this time, there is no guarantee as to whether the variables are bound in parallel or sequentially, however.

When compiled, however, the binding evaluation of the *vals* and binding of the *vars* is done at *compile* time, and then the *forms* (as a progn) compiled in that environment. This is one way to communicate information to the compiler and to macro functions, and is the only way to set certain compiler switches locally right now.

compiler-let is handled properly as a toplevel form in a file, and is properly tranparent to things like special predicate compilations.

Most of the variables which compiler-let used to be useful for have be excised from NIL, primarily because the primitives they controlled the compilation of have become more generic due to the implementation of COMMON LISP arrays. The only two such variables left are compiler:*open-compile-carcdr-switch and compiler:*open-compile-xref-

switch, and these are now obsoleted by use of the optimize declaration. compiler-let is still useful, of course, for communication between macros, and that is its intended use anyway.

24.1 Interaction Control

compiler:*messages-to-terminal? *Variable*

If this is not nil, then the compiler (compile-file and compile, and even the LSB-defined function lsbcl [4]) will print out verbosely on the terminal. If it is nil, nothing will be printed, unless errors occur, which is a separate can of worms. By default, this is t.

24.2 Efficiency, Optimization, and Benchmarking

The current NIL compiler is incredibly stupid in some ways. Basically, it is an overgrown one-pass code generator: it takes in a LISP program, and spits out VAX machine language, making liberal use of LISP function calls and special subroutines to handle things which are too bulky or too complex for it to code out itself. This one-pass nature is why it is not capable of handling completely general lexical variable references (among other things). On the other hand, it contains what amounts to an immense (procedural) database for how to compile things, and how to perform transformations from some general forms to more specific forms which can be handled more efficiently. For instance, the form

```
(member item list)
```

is actually coded as a call to si:member-eql, which does not take keyworded arguments like member does (see page 61), reducing the overhead of the call. Similarly,

```
(member item list :test #'eq)
```

is coded as a call to memq, which is itself implemented as a "minisubr", a quick (VAX JSB instruction) subroutine call to a hand-coded routine in the NIL kernel. Finally, there are many things the compiler *does* know how to "do out", but for various reasons may not choose to. The most prominent of these are the functions car, cdr, and their compositions and updates. While these can be trivially inline coded by the compiler (car and cdr are one instruction each if the cons is in a register), experience has shown that having error checking for them in compiled code greatly facilitates debugging, while generally providing only a minor efficiency penalty except in critical paths.

The general principle of the NIL compiler is, therefore, that the default settings of the optimization parameters (see the optimize declaration on page 45) should provide fairly "safe" code with error checking, as long as such safety does not carry with it too great an efficiency penalty. This is in keeping with the philosophy that casual or naive use of NIL should not result in undebuggable code, or code which tends to produce "hard" errors (for instance, reference to non-existent memory instead of a wrong-type-argument error from car).

Even with lack of "optimization" in the traditional compilation sense, the NIL compiler often has various choices to make about how to implement certain constructs which involve space, speed, and safety tradeoffs. One pervasive example of this is in how it sets up and performs function calls. This is probably the prime example, because it involves not only space, speed, and safety compromises, but also compilation-speed. I will not go into the fine details (which would necessitate explaining how function calling is performed at the VAX machine level), but try

to briefly describe it more generally. Basically, a function call is handled by the caller "creating" a call frame on the stack, filling it in, and then performing the actual call. When such a frame is created, it can be recognized as such by a debugger (the current one doesn't know beans about it, but that is beside the point), and, while the arguments are being computed, it is possible to tell not only what function is *about* to be called, but which arguments to it have been computed (and what their values are), and which have not. If the compiler decides that it is ok to not have *quite* this much debugability, it might decide to try to compute some of the arguments as it creates the function call argument frame—this can save both space and time, but means you cannot tell what is going on while it is happening. No matter which way it does this, it may still have to push varying length blocks of constants on the stack. Even here, there are space/time tradeoffs to be made: for instance, below some threshold, it takes longer to do a block-move than to do a series of moves/pushes. And, there is a decision as to what constant it is that gets pushed: a marker (recognizable by the debugger), or just zeroes (which can be done more efficiently).

Another area is that of certain functions which can be inline-coded, but which are normally too bulky to make it worthwhile under normal circumstances. **memq** is a fine example of this. Normally, **memq** is a minisubr which will give a correctable error if the list is not a proper list, and which will detect circularity. If speed is more important than space and safety, however, the compiler can inline-code the loop. The exact determination is not only a function of the qualities specified with **optimize**, but also the particular circumstances. If the list argument to **memq** is known at compile time, then the compiler can verify that it is a non-circular proper list, and thus no longer has to worry about the "safety" of the call. If in addition the **memq** is being used for "predicate value", that is, in a context like

(**if** (**memq** *item list*) *then-do-this* **else-do-this**)

where the specific value of the **memq** does not matter, then the compiler can just code out the proper series of **eq** checks. How many it will be willing to do this for, is once again a space/time tradeoff.

There is another class of optimization which can be performed: that is unfolding of common cases out of a more generic routine. For instance, the **1+** and **1-** functions (page 74) are coded as minisubrs. They are fairly generic, since they can accept arguments of any numeric type. But a common usage is with arguments which are fixnums. If space is not considered particularly important, but speed is, then the compiler can check to see if the argument is a fixnum, and if so do the addition or subtraction itself without getting into the generic routine. Because the code only does this on things it has verified as fixnums, and because it checks for overflow, this particular optimization does not involve "safety".

Another aspect of "safety" is how likely the operation being performed is to produce something "illegal"; in a LISP implementation like NIL, many objects like fixnums and characters have their data represented in the pointer, and operations on those objects just do operations on the pointers. For certain trivial operations, of which many of the character functions (chapter 11) are characteristic, the compiler might choose to produce code which is a bit more circumspect about what it does with pointers, what data-types of objects it might generate if given erroneous inputs, or possibly even whether it inline codes at all, depending on what the result of an erroneous input might imply. Take, for example, **char-upcase**: if this were inline-coded (it might not be in this particular version of NIL), it effectively would be replacing part of the "address field" of the pointer which is its argument, by something else. If that argument was not a character, but (say) a vector, it is quite likely that the resultant object would be unprintable,

give garbage when examined, and, assuming the existence of a garbage-collector, cause the garbage-collector to fail in the middle of its operation, leaving the LISP totally useless. Many operations of this nature, while having a moderate amount of overhead if done as function calls with error checking, can be made to at least not return such objects with only a trivial overhead. While in a sense this masks errors rather than detects them, having your lisp get blown away by an illegal pointer does not facilitate debugging either. In this regard, the things which heed the safety optimization quality will generally not inline code at all (depending on their triviality) if the safety quality is set to 3. (Also, in general, only high-level routines might offer this choice: the NIL fixnum-only routines, and character "subprimitives", will always inline-code, and not offer this safety "option".)

There are very few things which recognize the **compilation-speed** optimization strategy. These few include such trivialities as the optimization which would convert **(cons** *x* **nil)** into **(ncons** *x*), which would compile into trivially better code. The one non-trivial place which currently handles this is the compilation of function calls—if compilation-speed is turned up, then they are just done the dumb, safe, easy way, which requires no lookahead (at the arguments) on the part of the compiler.

Generally, then, the default in NIL is to produce fairly "safe" code, but not unreasonably so; **car**, **cdr**, etc., and structure references, will be done with error checking; function calling will be optimized as described above, however, because it is so pervasive and the optimization generally results in a space savings in addition to the time savings. Setting the speed higher will result in a loss of safety in varying degrees for different operations, and some increase in the space used; setting the space quality higher might cause in increase in runtime of the resultant code, but probably not much, at least in the current environment: there are very few things in the current compilation scheme which are affected by increasing the importance of only space.

25. Introduction to the STEVE editor

25.1 Introduction

STEVE is a general purpose screen oriented text editor based upon the EMACS editor. In many respects STEVE and EMACS are identical, with the primary difference being that STEVE is written in NIL for the DEC VAX-11 series computers and can be called directly from the NIL interpreter. Those who are familiar with EMACS will be able to use STEVE immediately, and should skip to the end of this chapter, as the first part is meant to be an introduction to STEVE.

25.2 Getting Started

There is one difference between the editor environment and the rest of NIL to be aware of. Because the editor and VMS have conflicting uses for many of the control keys, the editor must run in "passall" mode. This implies that the normal interrupt commands do not normally work in the editor. So the first command to learn is the editor command to return to whatever you were doing before you entered the editor. It is a two key command typed by holding down the "Control" key and pressing the "Z" key twice.

`Control-Z Control-Z` `Return-to-superior.`
Exit the editor and return to whoever called it. This is the normal way to exit from STEVE.

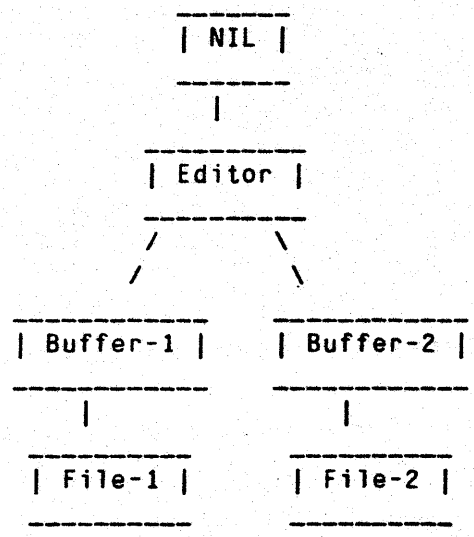
Now that you know how to exit the editor you may be curious how to enter it. Of course this is not an editor command, but rather a NIL function.

ed &optional *what-to-edit*
Enters the editor, returning to whatever you were working on before. If you have not run the editor since starting NIL it will be completely initialized with one empty buffer.

Normally one types (ed) to the NIL interpreter to get into STEVE. *what-to-edit* may be a pathname (or string naming a file), or the name of a function. If given, the editor will try to find the file or function definition and let you edit it; otherwise the argument is ignored. There are editor commmands to find files and function definitions anyway, so the argument is not really very important, except that it can be convenient, and can be used from programs.

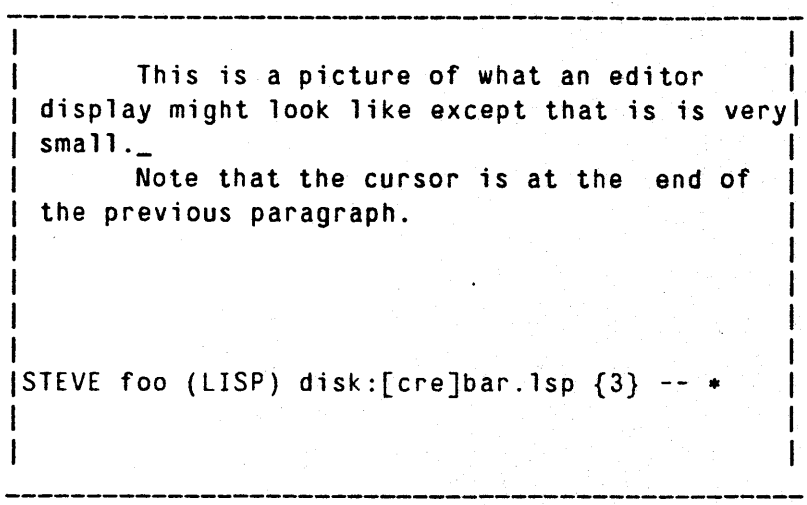
25.3 Editing Files

The principle purpose of an editor is to create or modify a file. In broad outline an editor is used by reading a file into a buffer, modifying it somehow and then writing it back to some long term storage device, generally a disk. Most of the editor commands are concerned with modifying a buffer, and will be expained later. In order to understand the commands for reading and writing files one should know about the general structure of STEVE and its buffers.



As the diagram shows, the editor runs inside NIL, and contains any number of buffers, each of which is associated with a file. This diagram can be modified by creating a new buffer or killing one, or by changing the file associated with any buffer. There are editor commands for all of these operations, and for some more complex combinations of them. The editor always selects one buffer as the current buffer, and displays a section of it around the cursor.

The format of this display is one of the features of an EMACS style editor like STEVE, and is the reason it is called a "screen editor".



The box in the diagram represents the edges of a terminal screen. The two paragraphs are the contents of a buffer. The single line below that is called the mode line. It contains as much information about the current state of the editor as is convenient. From this we see that the buffer is named "foo" and that it is associated with the file "disk:[cre]bar.lsp". The notation {3} after the file name indicates that the current version number is 3. If the file does not exist on disk the version number and the braces will be missing from the mode line. The star (*) on the right of the mode line indicates that the buffer has been changed so it is not the same as the file on disk. The current position of the cursor is at the end of the first paragraph. (On most terminals a cursor shows up as a blinking underscore or box, though this depends upon the exact type of terminal. In this chapter we show the cursor as a underline (_) since it is fairly difficult to print a blinking cursor.)

Under the mode line is a blank area of several lines. This is called the mode area and it is where most error messages and prompts are shown.

We are almost ready to start expaining the individual editor commands. The only other thing you should know first is how they are typed. Most STEVE commands are either one or two character commands. Since one adds alphabetic characters to the buffer simply by typing them (not that you know this yet) STEVE must not use alphabetic characters for its commands. Instead the control characters are used. (The control characters are typed by holding down the "control" key and pressing some other key, just as the capital letters are typed by holding down the shift key.) Since there are not enough control keys for all of STEVE's commands it also uses a *meta key*. A Meta key is similar to a shift key or a control key. Now we can have the characters "a", "A", "Control-A", "Meta-A", and "Control-Meta-A".

Unfortunately most terminals do not have a meta key. Not to worry, though, STEVE is designed to work without it, just as certain text justifiers are designed to work with terminals which have no lower case. Three commands are "bit-prefix" commands. Typing one of these will change the next character you type just as if you had been holding down the corresponding combination of control and meta keys.

`Altmode` `Prefix-Meta`
Pressing *Altmode* (marked SELECT or ESCAPE on some terminals) will make the next character a "meta" character. For example *Altmode F* (two characters) is the same as *Meta-F* (one character).

`Control-^` `Prefix-Control`
Pressing *Control-↑* (control-uparrow) will make the next character a "control" character. For example *Control-↑ F* (two characters) is the same as *Control-F* (one character). On some terminals, notably the VT100, *Control-↑* is typed as *Control-~* (control tilde); normally, the ^ character is a shifted 6, so one holds down *control*, *shift*, and *6*.

`Control-Z` `Prefix-Control-Meta`
Pressing *Control-Z* will make the next character be both a control and a meta character. For example *Control-Z F* (two characters) is the same as *Control-Meta-F* (one character).

All of these bit-prefix commands *add* the quality to the next character. There is no problem with doing it twice. The two character sequences *Control-Z Z* and *Control-Z Control-Z* both are read as *Control-Meta-Z*.

We are now ready start expaining the various editor commands. These are the commands you will use to create buffers and write files. All of these commands are safe to use since they will notice if you are about to destroy any of your work and ask you if you really want to do that.

`Control-X Control-F` `Find-File`
Find-File will prompt for a file name and you should type it from the keyboard. If there is a buffer for that file then it will be selected and be the new current buffer. Otherwise a buffer is created for the file and the file is read in from disk if it exists there. Find-File is the most common way to read a file from disk. It creates a new buffer for each file which is convenient. When Find-File creates a buffer it uses the file name without any extention as the buffer name. Since the name of each buffer must be unique this doesn't work when you are editing two files which have the same name but are on different directories or have different extensions (file types), so Find-File will notice if you are doing this and will ask you for a new buffer name to use.

`Control-X Control-S` `Save-File`
Save-File writes the current buffer to its associated file, and changes the mode line to indicate that the buffer and file are now identical. (This is not done until the output is complete, so if there is a disk error or some other error you will not think it has been saved when it hasn't been.)

`Control-X Control-V` `Visit-File`
`Control-X Control-R`
Visit-File is like Find-File except that it re-uses the current buffer, destroying its contents. It is still safe since it will offer to save it if any changes have been made to it.

`Control-X Control-W` `Write-File`
Write-File writes the current buffer to a file, but unlike save file it will prompt you for the file name.

As an example, suppose that the screen looks like the diagram above. If you type *Control-X Control-W* (Write-File) the editor will prompt you for a file name. Assume you want to save the file into `disk:[cre]baz.lsp`. You type "`disk:[cre]baz.lsp`". The screen will look like this:

```
|                                                |
|      This is a picture of what an editor       |
| display might look like except that is is very |
| small.                                         |
|      Note that the cursor is at the  end of    |
| the previous paragraph.                        |
|                                                |
|                                                |
|                                                |
|                                                |
|STEVE foo (LISP) disk:[cre]bar.lsp {3} -- *     |
|Write File:disk:[cre]baz.lsp_                   |
|                                                |
```

Notice that the cursor is temporarily placed in the mode area. After the command is complete it will return to the text in the buffer. The editor will fill in an incomplete file specification for you, using the file specification associated with the buffer. In this example the file name could have been typed as `[cre]baz.lsp` or `baz.lsp` or just `baz` since that is the only part that is changing.

After typing whatever file name you choose you must type *Return*. Most commands that prompt you in the mode area require a *Return* to end the command. Until you press *Return* you may change the file name using the delete key and retyping the parts that were wrong. Also the keys *Control-W* and *Control-U* usually delete a word or the whole command letting you start over. If you delete too far the command is aborted if that is legal, otherwise a bell will sound.

Suppose you hit the *Return* key now. The buffer will be written, the prompt will be deleted and the editor will tell you that it has finished. The screen will change showing you what is happening, and will look like this (though we cannot show you how it changes.)

```
|                                                |
|      This is a picture of what an editor       |
| display might look like except that is is very |
| small._                                        |
|      Note that the cursor is at the  end of    |
| the previous paragraph.                        |
|                                                |
|                                                |
|                                                |
|                                                |
|                                                |
|STEVE foo (LISP) disk:[cre]bar.lsp {1} --       |
|Writing File...                                 | This line...
|Written BAZ.LSP;1[CRE]DISK:                     | Then this line
```

Notice that the cursor has returned to the buffer text and that the star (*) has been removed from the mode line to indicate that the buffer and file are identical, and that the version number has been changed to 1. This is because the new file name did not exist on disk. Had the file been saved under its old name the version number would have been incremented by 1 from 3 to 4. Finally the file name in the mode area has been updated so that Save-File will use the new file name.

25.4 Modifying the buffer

25.4.1 The Simplest Commands

As I hinted before, typing any alphanumeric character will add it to the buffer. In fact almost any character that you can type without holding down the control key will act like this. Also, the delete (or rubout) key will delete the last character before the cursor. If you can place the cursor where you want it and delete and insert characters then you are already able to make any editing change you have to. Since it is so simple to change characters in the buffer, STEVE concentrates on commands to put the cursor where you want it quickly and easily. The first few such commands are:

`Control-F` `Forward-Character`
Control-F moves the cursor forward one character in the buffer. (The end of a line counts as one character.)

`Control-B` `Backward-Character`
Control-B moves the cursor backward one character in the buffer.

`Control-N` `Down-Real-Line`
Move straight down to the next line.

`Control-P` `Up-Real-Line`
Move straight up to the previous line.

These are the commands to move up down right and left.

Now you know how to edit a file! If you can you should probably try to use STEVE to create a simple file and save it. Print it if you can and compare it to what you see on the screen. See what happens if you try to back up before the beginning of the buffer using *Control-B* or *Control-P*. Type enough lines to fill up the screen (use *Return* to end each line) then a few more. What happens when the cursor is about to move onto the mode line? Now use *Control-P* to move back.

25.4.2 Now that you know the Simplest Commands

Now that you know the simplest commands there are many others that you should learn. There are some general facts about the editor which will help you get more out of each command which I will expain first.

25.4.2.1 Numeric Arguments

It is possible to give any command a numeric argument. The command or may not use it, but you can always supply it. In fact, if you don't supply an argument an argument of one (1) is implied. There are several ways to specify an argument. In all cases the numeric argument is typed before the command. The most general way to specify an argument is:

`Control-U Universal-Argument`

Control-U followed by a positive or negative integer specifies that integer as the argument for the following command. *Control-U* with no number specifies an argument of four (4). *Control-U Minus* with no number is treated specially as an argument of minus 1 (-1). Some commands treat *Control-U* with no number differently than *Control-U 4*.

For terminals with a meta key it may be easy to use the meta-digit keys.

`Meta-0, Meta-1, ..., Meta-9, Meta-Minus`
`Control-Meta-0,..., Control-Meta-9, Control-Meta-Minus`
`Auto-Argument`

Any of the Metafied numeric digits begin a numeric argument. It is just like *Control-U* followed by the digit. Notice that repeated meta digits are multiplied together.

`Control-0,..., Control-9, Control-Minus`
`Auto-Argument-Digit`

The control-digits end any previous digit and act as digits in an argument. Thus *Control-2 Control-3* is the argument twenty-three (23). Any arguments before or after a sequence of control-digits will be multiplied by the final control-digit argument. Because most terminals do not send control-digits these must be specified using the uparrow bit-prefix (for instance, by typing *Control-↑ 2*), so in practice they are not used much. Note that the *Control-Minus* must be specified first.

If several arguments are specified they are multiplied together. The primary use of multiple arguments is to type *Control-U* several times in a row. Each *Control-U* multiplies the argument by four (4). So *Control-U Control-U* is sixteen (16) and *Control-U Control-U Control-U* is sixty-four (64). The cursor movement commands treat the argument as a repeat count (as do most

commands where that is meaningful). Some useful combinations are *Control-U Control-U Control-F* which moves forward about a quarter of a line, and *Control-U Control-N* which moves down four lines. You will find many other "Cliches" or combinations of editor commands which you use automatically to do one thing.

25.4.2.2 Control-X

As I said before there are not enough keys on a keyboard for all of the commands defined in STEVE. The *Meta* key is one way of getting more characters so STEVE can have a large number of single character commands. But it is not enough. To get even more commands STEVE uses the key *Control-X* as a prefix character. There are many two character commands which begin with *Control-X*. What actually happens is that the editor normally looks up the command for each key in a table. The *Control-X* key says that the editor should use a different table for the next key. This greatly expands the number of commands that can be typed.

25.4.2.3 Meta-X and Control-Meta-X

With *Meta* and *Control-X* it is possible to define enough editor commands, but there is another problem. Eventually there are so many commands that it becomes difficult to remember them all. For this reason there is a command, *Meta-X*, which reads a command name from the keyboard and executes the command. It is easier to remember the name of an unusual command that to remember which key invokes it. In fact there are many commands which we don't bother to define keys for.

You type *Meta-X* either by holding the *Meta* key and pressing *X*, or by typing the *Escape* key followed by *X*. When you type it the cursor is moved to the echo area and a colon (:) is printed as a prompt. You type the name of the command and then type *Return* to execute it. Some Meta-X commands take "string" arguments. These can be typed in several different ways. The simplest way is to type the command name, then to type an *Escape* before each argument. (An extra *Escape* after the last argument will be ignored.) When the command has been typed with all of its arguments, press *Return* to execute it.

There are a number of special features which make it easier to type a *Meta-X* command. The *Delete* (or *Rubout*) key will delete the last character you have typed. (If you delete too many characters the *Meta-X* command is aborted.) The *Control-G* key will abort the command at any time. (*Control-G* will abort a partially typed command almost anywhere in the editor.) *Control-W* will delete a word, and *Control-U* will delete the entire *Meta-X* command, letting you start over.

The command does not have to be completely typed, only enough to make it unique. At any time you may find out if a command is unique by typing *Escape* (or *Altmode* on some terminals). The editor will finish as much of the command as it can and type that part of it for you. If it is not unique the bell will ring. If it is unique the *Escape* will be typed after the command (it appears as a dollar sign ($)). You may delete these characters just as if you had typed them if this is not the command you wanted.

The *Space* key is another special character. It is like *Escape* except that it only completes one word of the command. If the command is finished it will add an escape after the last word.

If you type a question mark (?) while typing a *Meta-X* command you will see a list of all possible ways to finish the command. This is typed in the upper part of the screen, over the text. (As soon as the *Meta-X* command is finished, the text will be re-displayed.) If the list is longer than one screenful the word "*more*" will appear on the last line above the mode line. Type *Space* to see the next screenful of commands. Type *Control-G* to abort the entire *Meta-X* command. (There are several other commands which use the upper part of the screen temporarily. All of these will print "*more*" in the bottom line and expect either a *Space* to continue, or a *Control-G* to abort. Any other character causes an abort, and is then used as a command.)

A summary of special *Meta-X* characters.

Delete	Rubout the last character showing in the command.
Escape	Completes the command and separates arguments.
Space	Completes a word.
Control-G	Abort everything.
Control-W	Rubout a word. Works while typing arguments also.
Control-U	Start over. Rubout the entire Meta-X command. (Doesn't abort.)
?	Help.

Most commands which are normally executed using *Meta-X* are smart about their arguments. They can determine how many you have typed and will prompt you for any that are required. Often it is easier to use *Meta-X* commands this way since the prompt will tell you what kind of argument to type. Some commands can do completion for you or otherwise help you type the arguments. The *Control-Meta-X* command is a variant of *Meta-X* which is designed to take advantage of this. The difference is that the command is executed as soon as it is completed, either by *Escape* or *Space*. Otherwise it is exactly the same as *Meta-X*.

25.4.2.4 Marks and Regions

Associated with each buffer is a ring which may store up to eight (8) marks. These are buffer pointers created by certain commands for future reference. There is a command to create a mark where the cursor is and a command to go to the last mark, and some other commands. The text between the cursor and the last mark is called the region. Many commands operate on this region.

25.4.2.5 Killing and Un-killing

Whenever more than one character is deleted it is stored in a place called a *kill-ring*. Should you decide that it was a mistake to delete it then you may retrieve it with the un-kill command (*Control-Y*). This also lets you copy text from one place to another, by killing it, moving the cursor and then un-killing it. To make several copies type *Control-Y* several times. The command un-kill-pop (*Meta-Y*) will retrieve the next to last peice of killed text. If *Meta-Y* is used right after *Control-Y* or *Meta-Y* the previous un-kill is deleted first. (Unlike ITS EMACS, *Meta-Y* can be used at any time.)

25.4.2.6 List Oriented Commands

A number of commands operate on "lists". These are normally defined as LISP lists with balanced parentheses. This definition is controlled by a syntax table and may vary in different major modes (see below). For example, in LSB mode the characters { and } are a type a parenthesis and will define a list. The editor knows about doublequote syntax for strings and vertical-bar syntax for symbols.

25.4.2.7 *more*

A number of commands will overwrite the text on the screen. There is no need to worry, the text has not changed and will redisplayed when the current command is finished. If this overwrite fills the top part of the screen then the word "*more*" will be printed on the line above the mode-line. The editor will wait for you to read the screen and type a space. The space will not be put into the buffer, it just indicates that you are ready to see the next screenful of information. If you type *Control-G* it will abort (see below).

25.4.2.8 Aborts

When the editor is reading from the terminal it usually will abort if you type *Control-G*. The word "aborted" will appear in the mode area. This is a good thing to try if you are losing, though it doesn't work in some places it should.

25.5 Major Modes

When editing different kinds of documents it is often convenient for some editor commands to behave slightly differently. For example, when editing a program it seems most useful to have the *Tab* key indent the current line so it lines up with the corresponding syntactic unit above it, but when editing a paper you want the tab key to indent for a paragraph. STEVE has a number of *major modes* which are designed for special kinds of editing. Most of the major modes are very similar, so there is no need to relearn much when you change modes.

`Bolio mode`
: A mode built on Text mode (see below) indended for sources to the text justifier Bolio. Knows about Bolio comments. Also assumes that Bolio is being used to document a Lisp program, so the paren echo hack is turned on and *Meta-.* tries to find a function definition. The Control-Meta digits are used to change to that number font. *Control-Meta-** inserts a "pop font" command.

`Fundamental mode`
: The basic mode upon which most other modes are built. Not used for much editing, since usually there is a better and more specialized mode for any particular job.

`Lisp mode`
: For editing LISP programs. The principle features are that parentheses are matched as they are typed (try it, it is hard to explain) and that the *Tab* key knows how to indent for LISP code.

`LL mode`
: Lisp Listener mode is not really for editing documents. It simulates the LISP (or NIL) top level loop by evaluating each top level form as soon as it is typed, and printing the result into the buffer. There are several reasons to use this mode for interactive testing. Because you are typing at the editor you have its full power to modify a form as you type it in. You are not limited to deleteing the last characters typed as you would be normally. Even after a form is executed you may modify it and re-use it by backing up (with *Control-P*), editing it, and then re-executing the form with *Meta-Z* or by erasing and re-typing the last close paren. Finally, there is a record of what you have done, and the results. You may save the buffer and print it. You may add comments as you work.

`LSB mode`
: For editing LSB programs. The primary difference from Lisp mode is that the characters { and } are also treated as Parentheses.

`Test mode`
: This should be dyked out. It is not useful except for debugging the editor itself.

`Text mode`
: For editing english (or german or french...) text. *Tab* is normal and *Meta-.* only searches the loaded buffers without trying to find the source file through the function definition. (This may be wrong... comments?) Paragraph commands search for lines which begin with a white space character rather than for blank lines (as they do in program modes.)

25.6 Help and Self Documentation

STEVE has a several commands designed to help you when you don't know how do something. The principle commands are *Meta-?* and *Control-Meta-?*, which is the more general of the two. When you type *Control-Meta-?* the editor will prompt you in the mode area with:

```
Help (type ? for options):
```

You respond with a single character. The choices are

- A Apropos. (You type a Word to search for.)
- C Document a Character. (You type the character.)
- D Describe a command. (You type the command name.)
- K Document a Key. Identical to C.
- S Syntax. (You type a character.)

A (Apropos) prints all paragraphs in the help file which contain a string. It is useful for finding documentation on some concept. Also available through *Meta-X Apropos*.

C (Character) finds the name of the command that a key is bound to and then treats that just like D (Describe) would. Also available through *Meta-X Describe-Key*. (Type the full name. *Meta-X Describe* confuses completion.)

D (Describe) searches for a paragraph in the help file which contains the string in the first line of the paragraph. The help file is structured so that paragraph will be the documentation for that command when it is fully typed. If this is losing because you don't know the full name of the command try Apropos instead. Also available through *Meta-X Describe*.

K (key) is another name for C (Character) and *Meta-X Describe-Key*.

S (Syntax) documents the editor syntax of characters. The character is read using the NIL function read, so many characters can be typed as themselves. Most others can be typed by using the quote prefix "\". The possible syntax types are *Word-Alphanumeric*, *Lisp-Alphanumeric*, *White-Space*, *Paren-Open*, *Paren-close*, *String-Quote*, *Character-quote*, and *Prefix*. Also available through *Meta-X Describe-Char-Syntax*. (Note that *Meta-X Describe* interferes with completion of this name.)

25.7 Glossary of Commands

So far you know about how to insert characters into the buffer, give commands arguments and these commands:

Control-F	`Forward-Character`
Control-B	`Backward-Character`
Control-N	`Down-Real-Line`
Control-P	`Up-Real-Line`
Control-X Control-F	`Find-File`
Control-X Control-S	`Save-File`
Control-X Control-V	`Visit-File`
Control-X Control-R	`Visit-File`
Control-X Control-W	`Write-File`
Delete	`Backward-Delete-Character`

Starting on the next page is a complete list of commands, including these and all others.

Glossary Of STEVE commands

25.7.1 Special Character Commands

`Backspace` `Backward-Character`
Move the cursor backward one character or more if given an argument.

`Tab` `Insert-tab (In non-LISP modes)`
Insert a tab.

`Tab` `Indent-For-Lisp (In LISP modes)`
Indent the current line according to the nesting structure.

`Linefeed` `linefeed`
Break the current line and indent the next line. Equivalent to *Return* followed by *Tab*.

`Return` `Crlf`
Insert a line separator or just move to the next line if before two blank lines. Skips comment ender if there is one.

`Altmode` `Bit-Prefix Meta`
Make the next character be a *Meta* character.

`Rubout` `Backward-Delete-Character (in non LISP modes)`
Deletes one character before point. If given an argument kills that many characters before point.

25.7.2 Control Character Commands

`Control-Altmode` `Exit-Editor`
Return to whoever called the editor, generally the NIL interpreter.

`Control-Space` `Set-or-pop-mark`
With no argument places a mark at point. With an argument pops the last mark and goes to it.

`Control-;` `Indent-for-comment?`
Inserts a comment on the current line or adjusts the placement of an existing comment.

`Control-<` `Mark-Beginning`
Place a mark at the beginning of the buffer.

`Control-=` `What-Cursor-Position`
Prints the *X* and *Y* coordinates of the cursor on the screen, the current character and the number of characters before point and the percentage of the file which that is. Line separators count as two characters since that is how many they occupy in a file. See Count-Lines-Region

`Control->` `Mark-End`
Place a mark at the end of the buffer.

`Control-@` `Set-or-pop-mark`
With no argument places a mark at point. With an argument pops the last mark and goes to it.

`Control-A` `Beginning-Of-Line`
Move the cursor to the beginning of the current line.

`Control-B` `Backward-Character`
Move the cursor back one character or more if given an argument.

`Control-C` `Exit-Editor`
Return to whoever called the editor, generally the NIL interpreter. *Control-C* should interrupt the editor as it does in the rest of NIL but because the editor must be in Passall mode that is not possible.

`Control-D` `Delete-Character`
Delete the character that the cursor is on.

`Control-E` `End-Of-Line`
Move the cursor to the end of the current line.

`Control-F` `Forward-Character`
Move the cursor forward one character or more if given an argument.

`Control-G`
Control-G will abort the editor if it is reading from the terminal.

`Control-H` `Backward-Character`
Just like *Control-B*. *Control-H* is *Backspace* in seven-bit ASCII.

`Control-I` `Tab`
Control-I does whatever *Tab* would do. In Lisp Mode and its derivatives (see major modes, below) this indents according to the syntax of text as a LISP program. In non-Lisp modes this is a normal *Tab*.

`Control-J` `Indent-New-Line`
Equivalent to *Return* followed by *Tab*. Ends the current line and indents the next line.

`Control-K` `Kill-Line`
Kill to the end of the current line. If the cursor is at the end of a line it kills the line separator. With an argument kills that many lines.

`Control-L` `New-Window`
Clear the screen and redisplay everything. Useful if the screen is garbaged somehow (for example if someone sends you mail). The window is moved to put the cursor in the middle of the screen. With an argument puts the cursor that many lines from the top of the screen. With a negative argument counts from the bottom of the screen.

`Control-M` `CRLF`
Insert a line separator or just move to the next line if before two blank lines. Skips comment ender if there is one.

`Control-N` `Down-Real-Line`
Move the cursor straight down one line or more if given an argument.

`Control-O` `Open-Line`
Puts a *Return* right after the cursor. With an argument creates that many blank lines.

`Control-P` `Up-Real-Line`
Move the cursor up one line or more if given an argument.

`Control-Q` `Quoted-Insert`
The next character is treated as an alphanumeric character regardless of what it is. This is how to put control characters into the buffer. *Meta* characters cannot be put in the buffer, because they cannot be in NIL strings.

`Control-R` `Reverse-I-Search`
Incrementally search backward through the buffer for a string.

`Control-S` `I-Search`
Incrementally search the buffer for a string.

`Control-T` `Transpose-Characters`
Exchange the character before the cursor with the character at the cursor.

`Control-U` `Universal-Argument`
Read an argument for the next command.

`Control-V` `Next-Screen`
Move the window and the cursor forward almost one screenful. The last two lines of the window are now the top two lines. With a numeric argument moves the window and cursor that many lines.

`Control-W` `Kill-Region`
Kill the region between point and mark and save it in the kill ring.

`Control-X` `Prefix-Character`
Control-X is a prefix character. Type any character after it for a two character command.

`Control-Y` `Un-Kill`
Get the most recent kill out of the kill ring and insert it in the buffer. With an argument *N* gets the *N*th kill. With just *Control-U* as an argument, it leaves the cursor before the un-killed text.

`Control-Z` `Bit-Prefix Control-Meta`
Read the next character as a *Control-Meta* character.

`Control-\` `Prefix-Meta`
Read the next character as a *Meta* character.

`Control-]` `Abort-Recursive-Edit`
Return from a recursive edit without doing anything more.

`Control-^` `Bit-Prefix Control`
Read the next character as a *Control* character.

`Control-Rubout` `Backward-Delete-Hacking-Tabs`
Like *Rubout* except that a *Tab* is first expanded into spaces. This is useful for indenting things. In Lisp modes *Rubout* and *Control-Rubout* are interchanged.

25.7.3 Meta Key commands

Meta-Linefeed **Indent-New-Comment-Line**
Equivalent to *Control-N Meta-;*

Meta-Return **Back-To-Indentation**
Put the cursor on the first non white-space character in the current line. (Tabs and spaces are white-space.)

Meta-Altmode **Minibuffer**
Start a minibuffer.

Meta-# **Change-Font-Word**
Change the font of the previous word.

Meta-(**Make-parens**
Enclose the next LISP expression in parens. With an argument enclose that many LISP expressions.

Meta-) **Move-Over-Right-Paren**
Move past the next close parenthesis, then do a *Linefeed*.

Meta-. **Defun-Search-All-Buffers**
Find a defun. In some modes this will look at the subr object to find the module a grovel around to find and load the file where the function is defined. In most text modes (other than bolio) it just searches the loaded buffer.

Meta-; **Indent-for-comment?**
Inserts a comment on the current line or adjusts the placement of an existing comment.

Meta-< **Goto-Beginning**
Put the cursor at the beginning of the buffer.

Meta-= **Count-Lines-Region**
Prints the number of lines between point and mark in the mode area. Also prints the number of buffer characters between point and mark (counting the line separator as one character. See What-Cursor-Position.)

Meta-> **Goto-End**
Put the cursor at the end of the buffer.

Meta-? **Describe-Key**
Reads a key from the keyboard and prints its documentation.

Meta-A **Backward-Sentence**
Move to the end of the previous sentace.

Meta-B **Backward-Word**
Backup one word. (With an argument backs up that many words.)

Meta-C **Uppercase-initial**
Capitalize a word.

Meta-D **Kill-word**
Kill the next word.

`Meta-E` `Forward-Sentance`
Move the cursor to the end of the current sentance.

`Meta-F` `Forward-Word`
Move over one word. With an argument moves over that many words.

`Meta-H` `Mark-Paragraph`
Put point at the beginning of a paragraph and mark at the end.

`Meta-I` `Insert-Tab`
Puts a tab into the buffer. *Meta-I* does not change in Lisp modes.

`Meta-J` `Indent-New-Comment-Line`
Equivalent to *Control-N Meta-;*.

`Meta-K` `Kill-Sentence`
Kill the sentence after the cursor.

`Meta-L` `Lowercase-Word`
Convert the next word to all lowercase characters.

`Meta-M` `Back-To-Indentation`
Move the cursor to the first non white-space character in the current line.

`Meta-N` `Down-Comment-Line`
If the current line has a blank comment delete it. Then move to the next line and add or adjust the comment start in the correct column.

`Meta-P` `Up-Comment-Line`
If the current line has a blank comment delete it. Then move to the previous line and add or adjust the comment start in the correct column.

`Meta-R` `Move-To-Screen-Edge`
With an argument move to the beginning of that line on the screen. With a negative argument count from the bottom. With no argument move one third from the top.

`Meta-S` `Center-Line`
Centers the non white-space characters in the current line.

`Meta-T` `Transpose-Words`
Exchange the words before and after the cursor.

`Meta-U` `Uppercase-Word`
Convert the next word to all upper case characters.

`Meta-V` `Previous-Screen`
Move point and the window back so the two top lines become the two bottom lines. With an argument move that many lines.

`Meta-W` `Copy-Region`
Put the text between point and mark in the kill ring but do not delete it from the buffer.

`Meta-[` `Backward-Paragraph`
Move to the beginning of a paragraph. In Lisp modes a paragraph begins with a blank line. Otherwise a paragraph begins with a line that starts with a white-space character.

`Meta-\` `Delete-Horizontal-Space`
Delete any spaces or tabs around the cursor.

`Meta-]` `Forward-Paragraph`
Move to the end of a paragraph.

`Meta-^` `Delete-Indentation`
Join the current line to the previous line and delete white space as appropriate. Leaves the cursor where the line separator was, so a *Linefeed* undoes the effect of *Meta-^*.

`Meta-~` `Buffer-Not-Modified`
Clears the flag which says the current buffer has been changed. The star (*) in the mode line will be erased. Be careful with this command: use it only when you are sure there have not been any changes to the buffer that you want saved.

`Meta-Rubout` `Backward-Kill-Word`
Kill the word befor the cursor.

25.7.4 Control-Meta Commands

`Control-Meta-Backspace` `Mark-Defun`
Put point at the beginning of a defun and mark at the end.

`Control-Meta-Linefeed` `Indent-New-Comment-Line`
Equivalent to *Control-N Meta-;*.

`Control-Meta-Return` `Back-To-Indentation`
Move the cursor to the first non white-space character in the current line.

`Control-Meta-(` `Backward-Up-List`
Move backward to next enclosing open parenthesis.

`Control-Meta-)` `Forward-Up-List`
Move forward to next enclosing close parenthesis.

`Control-Meta-;` `Kill-Comment`
Kill the entire comment field on the current line.

`Control-Meta-?` `Editor-Help`
Self documentation function. Type a single character (one of `A`, `C`, `D`, `K`, `S`, or `?`) to select which type of help you want.

`Control-Meta-@` `Mark-Sexp`
Put the mark at the end of the next LISP expression.

`Control-Meta-A` `Beginning-Of-Defun`
Backup to the beginning of the current or previous defun. Does not require matched parentheses or a complete defun.

`Control-Meta-B` `Backward-Sexp`
Move backward over one LISP expression.

`Control-Meta-C` `Compile-Sexp`
Compile the current defun. Only works for NIL code. The compiled function is loaded into the current NIL.

`Control-Meta-D` `Down-List`
Move to the inside of the next list in the buffer.

`Control-Meta-E` `End-Of-Defun`
Move to the end of the current or next defun. Does not require matched parentheses or a complete defun.

`Control-Meta-F` `Forward-Sexp`
Move forward over one LISP expression.

`Control-Meta-H` `Mark-Defun`
Put point at the beginning and mark at the end of the current defun.

`Control-Meta-J` `Indent-New-Comment-Line`
Equivalent to *Control-N Meta-;* .

`Control-Meta-K` `Kill-Sexp`
Kill the next LISP expression.

`Control-Meta-M` `Back-To-Indentation`
Move the cursor to the first non white-space character in the current line.

`Control-Meta-N` `Forward-List`
Move forward over one list.

`Control-Meta-O` `Split-Line`
Break a line at the cursor and indent the second half so it starts in the same column.

`Control-Meta-P` `Backward-List`
Move backward over one list.

`Control-Meta-Q` `Indent-Sexp`
Apply tab to every line in the LISP expression following the cursor except for the first line.

`Control-Meta-R` `Reposition-Window`
Try to place the beginning of the current defun at the top of the window without moving the cursor. Does not require balanced parentheses.

`Control-Meta-T` `Transpose-Sexps`
Exchange the previous and next LISP expressions.

`Control-Meta-U` `Backward-Up-List`
Move backward to the previous enclosing open parenthesis.

`Control-Meta-V` `Scroll-Other-Window`
In two window mode scrolls the other window forward. With an argument scrolls by lines.

`Control-Meta-W` `Append-Next-Kill`
If the next command is a kill command the previous kill will be appended to it, even if it would not otherwise be. Has no effect if the next command is not a kill command.

`Control-Meta-X` `Instant-Extended-Command`
Read an extended (named) command from the keyboard and execute it. If completion finishes the command name it will be executed instantly, without waiting for a *Return*.

`Control-Meta-[` `Beginning-Of-Defun`
Move to the beginning of the current or previous defun.

`Control-Meta-]` `End-Of-Defun`
Move to the end of the current or next defun.

`Control-Meta-^` `Delete-Indentation`
Join the current line to the previous line and delete white space as appropriate. Leaves the cursor where the line separator was, so a *Linefeed* undoes the effect of *Control-Meta-^*.

`Control-Meta-Rubout` `Backward-Kill-Sexp`
Kill the LISP expression before the cursor.

25.7.5 Control-X Commands

`Control-X Control-A` `Toggle-Auto-Fill-Mode`
With no arg, toggles auto fill mode. With a negative arg, turns it off. With a positive arg, turns it on and sets Fill Column to that number.

`Control-X Control-B` `List-Buffers`
Lists all buffers and their major modes.

`Control-X Control-Z` `Exit-Editor`
Return to whoever called the editor, generally the NIL interpreter.

`Control-X Control-D` `Directory-Display`
List all versions and types of the current file. With an argument reads a pathname and lists all files which match it.

`Control-X Control-F` `Find-File`
Find-File will prompt for a file name and you should type it from the keyboard. If there is a buffer for that file then it will be selected and be the new current buffer. Otherwise a buffer is created for the file and the file is read in from disk if it exists there. Find-File is the most common way to read a file from disk. It creates a new buffer for each file which is convenient. When Find-File creates a buffer it uses the file name without any extention as the buffer name. Since the name of each buffer must be unique this doesn't work when you are editing two files which have the same name but are on different directories, or have different extensions (file types) so Find-File will notice if you are doing this and will ask you for a new buffer name to use.

`Control-X Tab` `Indent-Rigidly`
With an argument shifts all lines in the region right (or left if negative) that many columns.

`Control-X Control-L` `Lowercase-Region`
Convert all characters between point and mark to lower case.

`Control-X Control-N` `Set-Goal-Column`
Control-N and *Control-P* try to move to the goal column if there is one. With an argument removes the goal column. Otherwise set it to the current cursor position.

`Control-X Control-O` `Delete-Blank-Lines`
Delete all blank lines following point, and if the current is blank delete all blank lines before it.

`Control-X Control-P` `Mark-Page`
Put point at the beginning and mark at the end of the current page.

`Control-X Control-Q` `Set-File-Read-Only`
With positive argument sets file read only.
With negative argument sets buffer read only.
With zero argument allows any access.

`Control-X Control-R` `Visit-File`
Visit-File is like Find-File except that it re-uses the current buffer, destroying its contents. It is still safe since it will offer to save it if any changes have been made to it.

`Control-X Control-S` `Save-File`
Save-File writes the current buffer to its associated file, and changes the mode line to indicate that the buffer and file are now identical. (This is not done until the output is complete, so if there is a disk error or some other error you will not think it has been saved when it hasn't been.)

`Control-X Control-T` `Transpose-Lines`
Exchange the current and previous lines.

`Control-X Control-U` `Uppercase-Region`
Convert all characters between point and mark to upper case.

`Control-X Control-V` `Visit-File`
Visit-File is like Find-File except that it re-uses the current buffer, destroying its contents. It is still safe since it will offer to save it if any changes have been made to it.

`Control-X Control-W` `Write-File`
Write-File writes the current buffer to a file, but unlike save file it will prompt you for the file name.

`Control-X Control-X` `Exchange-Point-And-Mark`
Put point where mark is and mark where the point was.

`Control-X Altmode` `Re-Execute-Minibuffer`
Evaluate the symbol "+". *Meta-X* and some other commands setq + appropriately so this does the right thing.

`Control-X #` `Change-Font-Region`
Sets the font number of the region to the argument. Good for Bolio at least.

`Control-X (` `Start-Kbd-Macro`
Begins defining a keyboard macro.

`Control-X 1` `One-Window`
Make the current window fill the entire screen and discard all other windows.

`Control-X 2` `Two-Windows`
Split the current window into two windows. Can create any number of windows until they get two small.

`Control-X 3` `View-In-Other-Window`
Split the current window into two windows but stay in the top half.

`Control-X 4` `Visit-In-Other-Window`
Combines Find-File and two window mode. Asks for a file to find, then displays it in a

new second window.

`Control-X ;` `Set-Comment-Column`
Sets the comment column to the current cursor column. Comment commands try to start comments in the comment column.

`Control-X =` `What-Cursor-Position`
Shows the *X* and *Y* coordinates of the cursor on the screen, the current character and how far through the buffer you are.

`Control-X A` `Append-To-Buffer`
Adds the text of region to the end of another buffer.

`Control-X B` `Select-Buffer`
Asks for a buffer name and creates or selects a buffer of that name.

`Control-X F` `Set Fill Column`
Sets the fill column to be the argument, if given, or else the current cursor position.

`Control-X G` `Get-Q-Reg`
Asks for the name of a LISP variable and tries to interpret its value as text to insert into the buffer.

`Control-X H` `Mark-Whole-Buffer`
Put point at the beginning of the buffer and mark at the end.

`Control-X K` `Kill-Buffer`
Reads a buffer name and kills that buffer.

`Control-X L` `Count-Lines-Page`
Prints the number of lines in the current page in the mode area.

`Control-X O` `Other-Window`
Selects the next window.

`Control-X T` `Transpose-Regions`
Transposes two regions defined by point and the last three marks.

`Control-X X` `Put-Q-Reg`
Asks for a lisp variable and saves the text in the current region there. Designed to be undone with Get-Q-Reg (*Control-X G*).

`Control-X [` `Previous-Page`
Move point to the previous page boundary.

`Control-X ]` `Next-Page`
Move point to the next page boundary.

`Control-X Rubout` `Backward-Kill-Sentence`
Kills text to the previous end of sentence.

25.7.6 Meta-X Commands

`Apropos`
Searches the documentation for a string and prints all paragraphs which contain the string.

`Auto-Fill-Mode`
Toggle auto fill mode. With an explicit argument, turn it on if positive, and off if negative. I forget what 0 does. Unfortunately this does not change the mode line. It will in the next version.

`Bolio-Mode`
Bolio mode is built on Text mode, but has features from Lisp mode. In particular *Meta-.* does a Find Function and the parenthesis balancing hack is turned on. Comments are Bolio comments. Also, *Control-Meta-digit* and *Control-Meta-** insert a *Control-F* followed by themselves, as font switching commands.

`Comment-Region`
Adds comments to the beginning of each line between point and mark. Can be undone with *Meta-X Uncomment-Region*. Won't work for languages with a comment terminator (I think).

`Compile`
Compiles the file associated with the current buffer. With a pathname argument compiles that file instead. Asks if you want the file loaded when done.

`Copy-Mode-Line`
Copy the first non-blank line of the last buffer selected to the first line of this buffer. An argument is the name of a buffer to use instead.

`Delete-File`
Reads a file name and deletes it. Asks for confirmation.

`Describe`
Reads a command from the keyboard and searches for documentation on it.

`Describe-Char-Syntax`
Reads a character and lists its editor syntax. For normal characters just type the character and *Return*. For special chracters you must type its symbolic name in accordance with the current readtable.

`Evaluate`
Reads and evaluates one NIL form. Prints the value in the mode area. Passall mode is turned off during evaluation for safety.

`Fundamental-Mode`
Sets the major mode for the current buffer to Fundamental.

`Help-Meta-X-Commands`
Lists the *Meta-X* commands. This will probably go away and be subsumed under some more powerful help function.

`Kill-Local-Variable`
Removes the current buffer's local binding of a variable.

`Kill-Some-Buffers`
Asks for each buffer whether to kill it or save it.

`Kill-Variable`
Attempts to makunbound some variable. May change or go away.

`Lisp-Mode`
Sets the major mode of the current buffer to Lisp. Turns on the parenthesis echo hack and some other features.

`LL-Mode`
Sets the major mode of the current buffer to LL (Lisp Listener). Lisp Listener mode is built on Lisp mode, but has the feature that a defun is evaluated and printed into the buffer when it is finished. It acts like the top-level loop in many ways, except all input and output is saved in a buffer. You also get to use *Tab* and the other editor features which help typing LISP forms.

`Local-Bind`
Bind some variable to some value when in the current buffer. If prompting for input this will tell you what the current value is.

`LSB-Mode`
Makes the current major be LSB. Very similar to Lisp mode, except that { and }. are also parentheses.

`Make-Local-Variable`
Like half of Local-Bind. Makes the variable local to the current buffer, but doesn't change its value. Not sure if this is useful, it is an attempt to sort of be compatible with EMACS.

`Name-KBD-Macro`
If there is a keyboard macro this will allow you to name it and to put it on a key. Asks for the key, then asks for confirmation about that.

`Overwrite-Mode`
This is not a major mode. It is also not finished. It is supposed to make self-inserting characters overwrite the existing characters rather that move them over. This much works, but there is some other hair which is unimplemented.

`Query-Replace`
Replace all occurances after point of the first argument with the second argument. Asks about each replacement. "?" will list the options in the mode area. *Space* does the replacement, *Rubout* does not, *Escape* exits immediately, *Period* (.) makes the replacement then exits, and *Comma* makes the replacement, then waits for a *Space* before continuing (so you can see the change before moving to the next one).

`Rename-Buffer`
Change the name of the current buffer.

`Rename-File`
Takes two file name arguments. Renames the first to the second.

`Reparse-Mode-Line`
Reset the major mode and all local variables from the file property list of the file associated with the current buffer.

`Replace`
Replace all occurrences of the first argument with the second argument. Acts

instantaneously (well, as fast as a VAX can go) and leaves the cursor where it was. Note: Currently the cursor is left at where the last string was replaced.

`Save-All-Files`
Lets you save any modified buffers. Asks about each one separately.

`Set-Key`
The first argument is a Key and the second is a binding. *Control-X* keys can be specified like (#\Control-X #\Control-B). Keys should be specified is accordance with the current readtable.

`Set-Variable`
sets a LISP variable to some value.

`Set-Visited-Filename`
Changes the file name associated with the current buffer, but does not change the buffer or write any files.

`Test-Mode`
A major mode build on I.L. mode (Lisp Listener) but with passall turned off. Not really sure why I did this, except to test the editor, since Passall is off in I.L. mode when reading and evaluating a form.

`Text-Mode`
The major mode for editing text. Also try Bolio mode.

`Trace-Current-Defun`
Tries to find the name of the current defun and call trace on it. Given and argument will trace that function instead.

`Uncomment-Region`
Tries to remove comments from a region of commented code. Meant to be used with *Meta-X Comment-Region*.

`Underline-Region`
If the terminal supports *underlining* change the visible part of the region so it is underlined. Waits for you to type a space, then reverts to the normal display and lets you continue.

`View-Buffer`
Shows the contents of a buffer in screenfuls.

`View-File`
Shows the contents of a file in screenfuls. Until the NIL garbage collector works this is much less efficient than visiting the file since all of the lines are wasted completely.

`View-Kbd-Macro`
Shows the sequence of characters in a keyboard macro in the mode area.

`View-Mail`
This is just a hack which runs View-File over the VMS mail file `sys$login:mail.mai`. If it doesn't work, don't use it.

`View-Variable`
Prints the value of a LISP variable. Doesn't barf if the variable is not bound. Other than that it is no better than *Meta-X Evaluate*.

`What-Page`
Prints the current page number and line number.

`Write-region`
Writes the text between point and mark to a file. Asks for the file name if it is not supplied.

25.8 Extending the Editor

Eventually the internals of the editor will be documented pretty completely. Currently the internals are subject to change, so any extention may be broken by future changes to the editor. However, as any hacker knows, a program does not change all that quickly... So one may assume that most of the internals will not change much. "Not being documented" means that I don't know which parts will change and which parts won't, so you pays your money and you takes your chances.

25.8.1 Editor Functions

An editor function is just a NIL function in the package STEVE. Currently the name of the function as given in this manual or with the Describe-Key command is the name of the NIL function, unless that conflicts with some other NIL function. (There has been some talk of adding a consistent prefix or suffix to all editor commands to distinguish them from other internal editor functions.) So you can call an editor function from NIL very easily, just find out its name. For example the LISP form (steve:forward-word) will move the cursor forward one word, just like *Meta-F* would. Numeric arguments are passed in the global variable **steve:*argument***.

`steve:editor-bind-key` *key-sequence binding* &optional *mode-name*
key-sequence may be either a character object or a list containing two character objects. It is evaluated. The code field of these characters should not be an ASCII control character; use the bits field to select a control character. A list is interpreted as a two character command using a prefix character (generally *Control-X*). The binding is not evaluated. It may be a function name, an editor command macro specification or a key indirection.

Normally the binding is a function name to call when the key is typed. The function will be called with no arguments.

If the binding is a character object the binding for that character object is used instead. This is only used for binding *Control-I* to *Tab*, so it may not be very robust.

A list is used to define an editor command macro. The **car** of the list is a function and the cdr is a list of arguments. When the editor is reading the key as a command the function is called and its values are returned as the "key" and command. This is hairy and should not be used lightly. Look at the code for numeric arguments and bit-prefixes to see how it can be used.

The *mode-name* is used to find the binding table for that major mode. The major mode must be declared when this is executed. The default is to use the current major mode, which is normally fundamental when not in the editor, i.e. when linking NIL.

If the *binding* is a symbol then it is also defined as a *Meta-X* command. Not sure if this is good but that's the way it is right now.

steve:editor-defun-key *key-sequence name* &body *forms*
A cross between defun and editor-bind-key. Defuns *name* to be a no argument function with a body of *forms* and binds it to *key-sequence* using editor-bind-key. There is some debate about whether to use this function or not.

A number of the editor functions take optional arguments which are intended to make it easier to use them from NIL code. Usually these are the arguments which the function uses. For example one may use the form (query-replace "foo" "bar") from NIL code. In particular most of the word functions take a numeric argument and use that instead of looking at the value of steve:*argument*. Some functions have an optional buffer-pointer as an argument. They will operate on this BP instead of the current cursor when they receive an argument.

25.8.2 Editor Objects

There are several special types of objects used by the editor. These are steve:buffer, steve:bp, steve:line, steve:edit-cursor, and steve:window-stream. All of them are flavors. The general intent is that they should not be changed in any way except by sending messages, nor should more messages be defined. The instance variables may be looked at using the accessor macros generated by defflavor, but be careful because the values are only valid until somthing changes.

A buffer object contains everything about a buffer including the text. It does not contain a cursor because there may be several cursors into one buffer. An edit-cursor contains a buffer a window and the position in the buffer where the upper right hand corner of the window is. An edit-cursor is also a bp, and as such it is the location of the cursor. A line is quite complex and should not be hacked under any circumstances. In addition to a string of characters and the length of the line it contains a list of the bps which point to that line. Whenever the line changes these bps must be relocated. A line also contains an index which indicates when it was last modified. This is used to optimize the redisplay. A bp (Buffer Pointer) is a pointer to some character in a buffer. The important instance variables are the line and position within the line. Remember that each line has to point to all bps that point to the line. A window-stream is an output-stream with an x-size, y-size and an x-position and a y-position. The redisplay does not know how to handle windows whose x-position is not zero, or whose x-size is not equal to the terminal width.

The correct way to create these objects is with these functions.

steve:make-bp *buffer line position*
Returns a bp pointing to the *position* character (zero based) in *line*. *buffer* may or may not be ignored. In any case the *line* must be in the *buffer*.

steve:make-line *buffer previous next* &optional *string*
Returns a line in *buffer* between *previous* and *next* containing *string*. If *next* is nil this will be the end of the buffer.

steve:buffer *spec* &key :create
spec may be a pathname, a buffer name (as a string), a buffer or an edit-cursor. The value is either nil or a buffer, which is found or created using *spec*. The keyword argument *create* determines if the buffer is created when it does not exist already. The default is to create a new buffer.

steve:point *spec* &key *create*
Like buffer except returns an edit-cursor. The argument *create* controls whether a buffer is created in order to build the edit-cursor. (If there is a buffer then an edit-cursor will always be returned, regardless of the value of *create*. An edit-cursor must have a buffer.) The edit-cursor may or may not have a window.

steve:point-selected *spec* &key *create*
Like point except that the edit-cursor is selected as the current cursor and its buffer is the current buffer.

This last function uses primitives which are useful in their own right.

steve:select-point *point*
Make *point* be the current cursor and its buffer the current buffer.

steve:select-point-in-current-window *point*
Like select-point except the window of the current cursor is stolen. This is usually the right way tp select a cursor.

Some common operations on lines. These are done carefully, so as to do the right thing at the beginning and end of the buffer.

steve:line-next *line*
Return the line after *line* or nil if at the end of the buffer. This is a macro generated by defflavor.

steve:line-previous *line*
Return the line before *line* or nil if at the beginning of the buffer.

steve:nth-next-line *line n*
Return the line *n* lines after *line*. If the end of the buffer is reached, the last line in the buffer is returned. If *n* is 0 the first argument is returned. If *n* is negative, moves backward.

steve:nth-previous-line *line n*
Like nth-next-line except moves up for positive *n*.

Some operations on bps.

:advance-pos *n*
Ask the bp to advance by *n* chars. Line separators count as 1 character. Bombs back to the editor top level at beginning and end of buffer.

:move *line n*
Place the bp pointing to the *n*th character of *line*.

:get-char
Return the character that the bp points to.

:get-char-forward
Return the character that the bp points to and advance over it.

:peek-char-backward
Return the character before the one that the bp points to.

:get-char-backward
Return the character before the one that the bp points to backup to point to it.

Note the unpleasant asymmetry of names. However, none of these can be interpreted as standard stream messages.

25.8.3 Other Functions and Conventions

Editor errors.

steve:save-all-files
This is the *Meta-X Save-All-Files* function. It may be called from outside the editor if the editor is broken, and may be able to save your buffers.

steve:ed-lose *format-string* &restv *format-args*
Abort any operation immediately. Print the *format-string* and ring the bell, then return to the editor top-level. The *format-string* is printed in the mode area. Passall mode is turned off while aborting to the top level, so if a bug causes a repetitive error you can escape by typing *Control-C* at the right instant. Keep trying, it works, but it may take a few tries.

steve:ed-warn *format-string* &restv *format-args*
Like ed-lose except the bell is not rung. In general ed-lose is used when the editor detects an error, and ed-warn is used for predictable events, like the *Control-G* abort out of a command reader. I feel that if the user has already done something to cause an abort he/she will not want to hear how upset the editor is. The bell is to bring attention to something unexpected.

steve:ed-warning *format-string* &restv *format-args*
Print *format-string* in the mode area, and continue. Does not cause an exit to the editor top-level, but continues any operation in progress.

steve:with-no-passall &body *forms*
Execute *forms* with the terminal not in passall mode. Sets up an unwind-protect form so an abort is o.k.

steve:*editor-device-mode* *Variable*
The editor sets the terminal to passall mode only if this variable is t. If you write an editor function which turns passall off and on you should always use the form:

```
(send terminal-io :set-device-mode
      :passall steve:*editor-device-mode*)
```

Arguments.

steve:argument?
Use the form (steve:argument?) to determine if any numeric argument was given.

steve:c-u-only?
Returns t if the argument was *Control-U* with no number.

steve:real-arg-sup?

```
(and (steve:argument?) (not (steve:c-u-only?)))
```

But more efficient in code and runtime.

steve:buffer-begin? &optional *bp*
Test whether *bp* (or the current cursor) is at the very beginning of the buffer.

steve:buffer-end? &optional *bp*
Similar; test for the end of the buffer.

steve:first-line? &optional *bp*
Returns t if the *bp* is anywhere in the first line of its buffer.

steve:last-line? &optional *bp*
Analogous.

steve:not-buffer-begin &optional *bp*
steve:not-buffer-end &optional *bp*
steve:not-first-line &optional *bp*
steve:not-last-line &optional *bp*
Return to the editor top level if the *bp* fails the given test. Otherwise do nothing.

Redisplay

steve:make-screen-image
This is poorly named. It used to be different. Now it is the redisplay entire and complete. Just call it and the screen will be redisplayed. (If a character has been typed it will exit immediately.)

steve:setup-mode-area
Generate and print a current mode line.

Some functions use the upper area of the screen to print things. The redisplay must be told that this has happened. This is handled by using several special functions to position the cursor and to do terpri. It is possible that this will be changed and that there will be a special stream which keeps track of such things. I was sick of defining special purpose streams when I got to this.

steve:overwrite-start
Begin to overwrite the display. If there has been some overwriting of the screen since the last redisplay start after it. Otherwise start at the top.

steve:overwrite-home
Start at the top always.

steve:overwrite-terpri
Move the cursor to the next overwrite line. This will do *more* processing as needed.

steve:overwrite-done
Always call this when finished with an overwrite display. This makes overwrite-start begin in the right place if called before a redisplay.

Reading from the terminal.

steve:mx-prompter *function format-string* &restv *format-args*
Prompts in the mode area using *format-string* and *format-args*, then reads from the terminal using *function*. Handles *Control-G* and has some additional internal hair which allows completing functions to be defined. May be modified to handle ? as a help key somehow.

steve:read-file-name
Can only be used as an argument to mx-prompter. Reads a file name and returns it as a string. Some day this will do completion.

steve:read-buffer-name
Only for use as an argument to mx-prompter. Will do buffer name completion and respond to ?. Example:

```
steve:(mx-prompter #'read-buffer-name "Foo(~a): " foo)
```

26. The Patch Facility

The patch facility provides a means by which a program (whatever that might mean) may be incrementally updated; it essentially a bookkeeping operation, and is primarily designed for providing the updates necessary for a dumped-out system. In the context of the patch facility, such a program unit is called a *patchable system*; use of the term *system* in this context means the same thing, but may not in other contexts. (NIL has no more sophisticated system-building tools currently, although it certainly has whatever primitives might be needed.)

The design of the NIL patch facility is originally derived from the LISP MACHINE LISP patch facility [12]. That was first implemented from scratch in MACLISP, and some time later the MACLISP version was copied and modified to be more appropriate for NIL. This is noted because there are various design flaws and misfeatures of the facility, which are inherited and are due in part to the application of the techniques used to a different programming environment. A future release should have a redesigned facility which will correct these things.

Patchable systems have both *major* and *minor* version numbers. The major version number corresponds to a complete new system generation, like when a NIL maintainer (one of the authors) loads up a new NIL, having incorporated any fixes into the source files and recompiled any files which needed it. The minor version number is incremented whenever an update is made. The updates are maintained on disk; each one corresponds to a particular file (a *patch file*) which implements the fix (usually, some function and variable definitions the same as in a newer version of some source file). A *patch directory* is maintained for each major version number; it enumerates (and describes) the patches for each minor version number. Finally, each patchable system has a *patch system definition file*, which primarily provides all kinds of default attributes about the system, which include the current version number and the location of the *other* files in the filesystem (thus the only place a pathname need ordinarily be specified to the patch facility is when pointing at the patch system definition file to define the patch system originally).

A typical cycle of usage for the authors might thus look like this. We have a freshly-made NIL, say version 175 (the Release 0 version). As bugs are found, they are accumulated into patch files. One person might accumulate several fixes over the course of a day into a single patch file. This might then be the update which makes Lisp 175.0 become Lisp 175.1. Exportation of the patch directory and the patch files for Lisp 175 to other sites will then allow them to be loaded by other dumped-out NILs of Lisp version 175. Eventually, one of us will decide that the changes are too far-reaching or too numerous, and decide to go on to another system version. This normally involves ensuring that all updated sources are recompiled, loading up a new NIL, and telling the patch system we want a new major version number. Note that the last is independent, conceptually, from loading up a new NIL: it is an operation which says that what we have on disk is a new version. A conceptual bug in the distributed NIL is the ease with which one may load up a NIL and increment the version number. Unless one is actually modifying the files which get loaded, one's site should remain at NIL version 175. If it does not, then a bug report referring to the NIL version is meaningless to us.

At the end of section 26.4, page 285, is a description of a more common usage of the patch system, where it is used for a system which is *not* dumped out.

26.1 User Functions

load-patches &rest *poorly-designed-keywords*

Loads patches for the specified (or all) systems. This takes keyword arguments in a non-standard fashion, although that is expected to be changed incompatibly in the future. All of them except for :systems take *no arguments*. They are:

:systems *list-of-systems*
Load patches for the specified list of patch systems, rather than all those currently defined.

:verbose
Be verbose. (Verbosity is forced when there is interaction, of course.) This is the default.

:silent
Don't be verbose.

:noselective
Don't be interactive, just load the patches. The default is to query the user on each patch.

:selective
Query for loading of each patch. This is the default. Note that one may answer P instead of Y or N to the query: this means proceed, which will cause all succeeding patches to be loaded non-interactively. load-patches is (supposed to be) clever to force verbose typeout when it is going to ask, and inhibit it again if :silent was specified and the loading was proceeded.

The standard NIL default init file does a

```
(load-patches :noselective)
```

to load patches without querying, but verbosely (so that you see what might be taking it a while during startup).

The following two functions, if given no arguments, print information about all defined systems; otherwise, about the systems given as arguments.

print-system-modifications &rest *systems*

This prints information about the systems as they exist in core. For each system, it lists its (current) status, and lists the minor version numbers that have been loaded, and their descriptions.

print-system-history &rest *systems*

This reads the patch directory for the named systems off of disk, and displays the information; all patches and their descriptions are listed (whether or not they have been loaded), status changes (the system status may change with a particular minor version number) are noted, and the "in-core" status with respect to all of this is shown.

Note that although the patch directory is read from disk, the patch system must be defined in-core in order for this to know where to look for the patch directory.

26.2 Patch System Information

si:system-version-info &optional *briefp*

Returns a string describing the versions and statuses of the patch systems defined. If *briefp* is specified and not nil, then the status information will be abbreviated, some ("insignificant") systems will not be shown, and the name of the primary system ("Lisp") will be omitted (it always comes first).

si:get-system-version &optional *system*

Returns multiple values describing the current version of the specified patch system:

* the major version number,
* the minor version number,
* and the system status keyword.

By some special strange dispensation, if *system* is not defined as a patch system, nil is returned as each of the values.

si:get-system-version-list *system*

This is a vestigial remnant of Maclisp implementation. Equivalent to

```
(multiple-value-list (si:get-system-version system))
```

(In MACLISP, the multiple-value support code does not normally reside in core, and code which runs interpreted and needs to examine system version information (for instance when loading up a system) might not want to force it to be loaded.)

si:print-herald &optional *stream look-out-of-core?*

This is what prints the startup message. If *look-out-of-core?* is not nil, then si:print-herald reads the patch directories off of disk so that it can show what the current versions and statuses are (what you would get if you do load-patches). With a non-null *look-out-of-core?*, *si:print-herald* effectively does a (si:update-system-statuses nil) (q.v.).

si:update-system-statuses? *system-list*

Looks on disk and corrects (if necessary) the in-core status information for each of the systems in *system-list*, or all defined patch systems if that is nil. The reason for this is that it is possible for the status of a system to change on disk (a particular patch might be deemed to be broken, or the system might be deemed to be no longer experimental, for instance). This is done implicitly by load-patches, and by si:print-herald with a non-null second argument.

26.3 Adding Patches

For the following set of functions, a default system/minor-version-number pair is maintained, from which *system-name* and *minor-version-number* are defaulted. The system name originally defaults to lisp, which is the name of the NIL patchable system. (This should be changed.) si:add-patch creates a new minor version number, allocates it in the patch directory, and sets this in-core default patch version to that. Then one can (for instance) do (si:compile-load-patch) to test that patch. If the function does not know which minor version number to deal with, then it will cycle through all of them, from the "most likely" one first, asking. One way to force this behavior is to specify a system but not a minor version number to one of these functions.

si:add-patch &optional *system-name description* &rest *options*
This allocates a new minor version number for the patchable system *system-name*, with a description of *description* and environment options of *options* (see the :environment-options keyword to the si:initialize-patch-system function, on page 285). It then calls si:re-edit-patch, below.

si:re-edit-patch &optional *system-name minor-version-number*
Creates a patch file for the appropriate file (if necessary), and calls the built-in editor on it.

si:compile-patch &optional *system-name minor-version-number*
Compiles the specified patch. This routine returns several values; the first of which is the pathname of the compiled file, so that it may be loaded.

si:compile-load-patch &optional *system-name minor-version-number*
Compiles and loads the specified patch.

si:finish-patch &optional *system-name minor-version-number*
"Completes" the specified patch; that is, marks it as finished. If a patch is not "finished", then load-patches will not load it (nor any succeeding patches).

si:abort-patch &optional *system-name minor-version-number*
Flushes (aborts) the specified patch. Any patch files are not deleted, however; you should consider doing that manually. If the minor version number was the highest in use, it will be reused, in which case a later si:add-patch will use the existing text file to start. Otherwise, there will be a missing minor version number, which is ok.

si:set-patch-environment *system minor-version-number* &rest *options*
In case you forgot with add-patches, this sets the option environment to *options*. Note it does not update the file attribute list in the source file of the patch! You must do that manually.

si:set-system-status *system status* &optional *major-version minor-version*
Note that this takes weirder than normal arguments. This sets the status of the specified version of *system* to be *status*. It is willing to modify the status list of major versions differing from that defined in the current environment. (Not to say that that would not be equally useful for some of the other functions...)

The typical use of this is to set either the current or the 0 minor version number of the current major version of some system to either :released or :broken, with the current status being :experimental (the default when a new major version number is made). Or, to change the status of an antiquated system from :released to :obsolete.

26.4 Defining Patch Systems

si:new-patch-system *system-name pathname* &optional (*do-what :increment-and-define*)

si:initialize-patch-system *system-name pathname* &key :initial-version
:patch-directory :patch-file :compilation-function :editing-function
:insignificant :default-directory :default-device :nodefault
:environment-options

:nodefault
If not nil, then si:initialize-patch-system will read in an existing version of the patch system definition file from *pathnam* (appropriately defaulted) to provide defaults for those options not specified. Otherwise, hopefully appropriate default defaults are used.

:initial-version
May be used to specify the version to be used. This will be written into the patch system definition file as the *current* version, which means that calling si:new-patch-system with the (typicall) :increment-and-define keyword will increment it first.

:patch-directory
A format string which should take one argument, which is the major system version, to construct the patch directory pathname for that major system version.

:patch-file
A format string which should take two arguments, the major and minor system versions (in that order), to construct the patch file pathname for the patchs of those versions. Alternately, it may be a list of two such format strings: the first will be used as the source file pathname, the second for the vasl file. (This may be used to split the stuff across directories or even structures, for instance if the sources are kept in a different place because of lack of disk space, or are simply not kept somewhere on some particular machine.)

:compilation-function
The function called by compile-patch etc. By default, compile-file (page 244) is used. This should be a symbol, not a closure or compiled-function object. The function will be given a first argument of the input pathname, and other keyworded arguments of :output-file and :set-default-pathname (which will be nil so as to not modify pathname defaults). That is,

```
(lsbcl input-pathname
    :output-file output-pathname
    :set-default-pathname nil)
```

:editing-function
The editing function which should be used to edit a patch file. It is called with a single argument, the patch file pathname. The default function simply returns the list of "now" "edit" and the pathname, which is then returned by si:add-patch or si:re-edit-patch.

:insignificant
If not nil, then si:system-version-info will not show this system when in brief

mode.

:default-directory
:default-device
These are used to construct the default pathnames used for the :patch-directory and :patch-file options, when they are not supplied. They are *not* used in defaulting (although they probably should be). The default-defaults for these are taken from *pathname*, and if absent from that, the directory name defaults to *system-name*. These options are significantly less useful in NIL than they were in the MACLISP version of this code...

:environment-options
This and some of the code involved is partially but not totally archaic; it predates NIL file attribute lists, and was put in to compensate for their absence. The code in Release 0 still performs redundant bindings of the involved attributes. However, the data in this option list is used to also initialize the textual file property list when si:add-patch initializes the patch file. Because of the kludginess of this, only a few options are supported, although it is extensible if need be (see the code). The options currently handled are

:package
The package name (default is "SYSTEM-INTERNALS", which is probably a poor choice)

:input-radix
The input radix (default is decimal).

LSB, which has been distributed with NIL, is a patchable system also. However, the normal NIL environment does not have LSB loaded by default. There is a file which can be loaded which will load up all of the parts of LSB. (It is NIL$DISK:[LSB]LOAD.LSP, if you have LSB online.) Essentially, it sets up the lsb package and loads up all of the component files of LSB and performs whatever initializations are needed, and then does

```
(si:new-patch-system
  "LSB" "NIL$DISK:[LSB.PATCH]SYSDEF"
  :define)
(load-patches :noselective)
```

The file NIL$DISK:[LSB.PATCH]SYSDEF.PSD was created with the *si:initialize-patch-system* function. Once that has been created, this reference to it in the LSB loadup file is the only pathname reference necessary; all others are contained in that file.

With the LSB patchable system, the files which are loaded by the loadup file are not normally modified except via patches. However, at strategic points, like when many files are being changed at once, or incompatible changes are being made, or the patches become numerous, then all of the files are changed (for instance, recompiled) at once, and the maintainer manually increments the version number of the LSB patchable system by doing

```
(si:new-patch-system 'lsb "nil$disk:[lsb.patch]sysdef"
                     :increment)
```

which increments the on-disk version number. Then, when someone loads the loadup file, they get the new files, the new major version number, and (until new patches get made) no patches loaded.

Maintaining the system in this way also results in a shorter turnaround time for testing out small fixes, and getting them "installed"; larger source files do not need to be recompiled.

27. Talking to NIL

27.1 Startup

The first thing NIL does when it starts up is to attempt to figure out what kind of terminal you are using. The way NIL figures out how to talk to a particular terminal is that it uses "terminal capabilities" database (a UNIX *termcap* database). The VMS logical name term is used to name the terminal type; NIL ignores VMS terminal information. If no such logical name is defined, then NIL will assume the terminal is a simpleminded printing terminal, and prompt you for a terminal name.

The terminal names which are both supported and known to work fairly well are

vt52
: Standard DEC VT52.

c100
: Human Designed System's concept-100. This will probably work for their Concept-108 also.

aaa
: Ann Arbor Ambassador

vt100
: DEC vt100. Obviously you should make sure your vt100 is in ANSI mode. Also, auto-linewrap should be disabled.

In DCL, one might say

```
$ define term "c100"
```

if one was on a concept-100. Or, if your terminal varied, you might put in your `login.com` file

```
$ inquir term "Terminal type (in doublequotes, default vt52)"
$ if term .eqs. "" then term := "vt52"
$ define term "''term'"
```

which would prompt for the terminal type to assign to the term logical name, defaulting it to whatever was convenient.

When NIL starts up, it loads your *init file* if it exists. This would be a file on your login (not default) directory named `NIL.INI`. (Init file conventions are discussed on page 200.) Then it enters its standard read-eval-print loop.

27.2 The Toplevel Loop

`*` *Variable*

The value of this is the last thing the toplevel (or breaklevel) loop evaluated (and presumably printed).

`**` *Variable*

The previous value of `*`.

`***` *Variable*

The previous value of `**`.

`+` *Variable*

The value of this is the last thing read in by the toplevel (or breaklevel) loop.

`++` *Variable*
`+++` *Variable*

Previous values.

`\\` *Variable*

This has as its value the *vector* of values returned from the last thing evaluated by the toplevel (or breaklevel) loop. That is, its first (number 0) element will be the value of `*`.

This variable is also used by the debugger the way `*` is by the toplevel loop, but that will be changed eventually.

If an evaluation error occurs and you abort back to toplevel, then the value of the `*` variables does not get cycled, but the `+` variables do; thus, `+` is the form which got the error, but `*` is still the last thing returned by toplevel evaluation. COMMON LISP intends to change this. (What NIL does is compatible with MACLISP.)

27.3 Entering and Exiting NIL

Typing the character control-Y normally exits from NIL. The same command which started the NIL initially may then be used to resume it. The NIL can be resumed in other ways too. For instance, if `nil` was the command used to start the nil,

`nil` will resume the existing NIL,

`nil/kill` will kill the NIL, and

`nil/proceed` will resume the NIL, but not allow it to type out, and will leave you in the command interpreter. If the NIL attempts to type out (or, in fact, calls any of the following functions), it will wait until it is explicitly resumed.

If NIL is reading input from the terminal, the input processor command for "meta-altmode", which may be typed as the character sequence *control-\ altmode*, will return control to the command interpreter. When the NIL is resumed, it will automatically redisplay the typein it is accumulating.

Several functions are provided for returning from NIL to the VMS command language interpreter (CLI) in a more programmable fashion. In all of the following functions where a string is involved with passing control back to the CLI, the string may have a maximum length of 256 characters. This is checked for by NIL.

valret &optional *command-line*

(valret) returns control to the CLI. The NIL is suspended until later resumed.

(valret *command-line*) returns to the CLI (suspending the NIL), and additionally causes *command-line* to be interpreted as a command line by the CLI.

The passall terminal mode is cleared, and restored when the NIL is resumed. valret with a string argument works by calling the VMS lib$do_command library routine.

quit

(quit) exits the NIL, causing it to be killed.

Currently, (quit *string*) kills the NIL and causes *string* to be printed instead of "NIL Terminated"; however this will probably be changed so that *string* will be interpreted as a command line, as with valret.

Passall mode is cleared on exit.

When the NIL is terminated, there is a noticeable pause before the command language interpreter returns. This is due to the controlling program (RNIL, running in the CLI) waiting for the process to actually go away. VMS image rundown takes a noticeable time, and if one were to not wait after requesting process deletion, starting up a NIL of the same name immediately could cause the new RNIL to be confused. (This is the same as happens when `nil/kill` is used in the CLI.)

exit-and-run-program (*pathname*)

Control is returned from NIL to the CLI, and the program image found in *pathname* is then run. The NIL will have been suspended. NIL applies no defaulting to *pathname*, however, the command interpreter will supply a default file type of exe and will default the device and directory to the RMS default.

Passall mode is cleared on exit. exit-and-run-program works by having the RNIL program call the lib$run_program library routine.

proceed-nil &optional *string*

Control is returned from the NIL to the CLI. However, the NIL is resumed, so will continue running "without the terminal". If a *string* is supplied, then that is printed (as with quit). Similarly (as with quit), the interpretation of *string* should probably be changed to be a command line for the CLI to execute.

27.4 VMS

VMS usurps control-Y as interrupt-to-superior. Resuming your NIL gives it a tty-return interrupt which makes it frob the cursor so that it knows where the cursor is. (This is why it goes to the bottom of the screen on a display tty.)

Other terminal interrupts are all performed by dispatching from control-C; after the control-C is typed another character is read. ("?" lists options.) The control-C processing is performed by Lisp code. This means that control-C will not be processed if interrupts are severely inhibited. The NIL system by special dispensation will enter the VMS debugger if multiple control-Cs are typed and are being ignored; so if something large and uninterruptible is happening, like creating a vector with a million elements, or something not so large but your system is slow, you might throw the NIL out to the VMS debugger.

It is highly unlikely that NIL will enter the VMS debugger unless explicitly told to do so. Here are two VMS debugger commands that are useful for returning back to the world. Say

```
call debug
```

and the lisp debugger will be entered. (Exiting softly from the Lisp debugger with "q" will return to the VMS debugger. One may also perform a non-local exit from the Lisp debugger with control-G or X.) One can also just

```
call quit
```

from the VMS debugger, which does a throw just like control-G does from the control-C prompt.

The only problem with all this is that if the VMS debugger gets dynamically loaded into the NIL, some part of it will end up where NIL thinks it is putting its heap (i.e., where it is allocating the memory to use for consing), and the NIL will die shortly thereafter.

si:lisp-debugger-on-exceptions *Variable*

If non-null, then error conditions and faults will trap to the LISP error handler rather than bombing out to the VMS debugger. This can happen from memory access violation errors, floating overflow and underflow, integer divide by zero, etc.; in general, any such error. For instance, a reserved operand fault might occur if a variable-field byte instruction was given a bad size.

If the lisp debugger is used from some exceptional condition, remember that the stack may not be in a nice-looking state, so examination of what the debugger thinks are local variables near the top may result in more trouble. Note also that the Lisp code is not run at AST level, but rather as a continuation of the condition; returning a value from the debugger returns that value from the most recent VAX procedure call, which is probably the function within which the error was signalled. Also, the LISP code which handles such errors binds si:lisp-debugger-on-exceptions to nil when it is running; examination of non-LISP data by the debugger from such an error as if it were LISP data might cause a memory protection violation, and blow out to the VMS debugger.

Note also the set-privileges and get-privileges functions (page 233), which can be used to set or get the privileges the NIL process has enabled.

27.5 Installation

There are 4 parts to installing VAX-NIL at your site.

1. Restoration of the `nil` directory hierarchy from the backup tape.
2. Definition of the required logical names and symbols.
3. Invoking the LISP dynamic linker.
4. Handling System and User considerations such as setting up the proper logical names and symbols (system or group wide and/or in user login files), and installing certain images for efficiency reasons.

[Step 1]
It is highly recommended that a rooted device definition be used for NIL$DISK, for example:

```
$ DEFINE/SYSTEM NIL$DISK "__DBA0:[LISPROOT.]"
```

The entire hierarchy, including executable, object, and source files in MACRO-32, BLISS-32, and LISP; and including various DCL command files and sundry data files and documentation comprises 1600 files and 40 sub-directories, using approximately 40 thousand blocks of disk space. If you have the disk space then restore the whole thing, if not, then use selective backup of [NIL.PORT], e.g.

```
$ BACKUP/LOG MTA0:NIL.BAK/SELECT=[NIL.PORT]*.*  NIL$DISK:[*...]
```

and then select the files according to MINIPORT.COM and VASPORT.COM.

[Step 2]
Use the following command:

```
$ @NIL$DISK:[NIL.COM]SYM
```

[Step 3]
If you are running VMS version 3.1 or above, then all you need run now is the lisp linker:

```
$ LISPLINK
```

This will result in a rather verbose display of "loading" messages, (which will take a minute or two to load the 120 or so files) after which the message "; Suspending Environment" will be printed. Followed after a silent pause of about 30 seconds by the standard system startup herald. At this point a read-eval-print loop is entered, where you will want to type (quit) to exit to DCL level. The newly created saved lisp enviroment may be restarted by

```
$ NIL
```

If you are not running VMS 3.1 or above then you may have to run the VMS linker, (in which case you had better have restored the obj files from BACKUP),

```
$ SET DEF NIL$DISK:[NIL.FOO]
$ NLINK
$ @RNILLINK
```

Sometimes the RNIL.B32 may need to be recompiled, to do that:

```
$ BLISS/LIB NILLIB:NILLIB
$ BLISS RNIL
```

then do the link commands as above of course.

example: To go from VMS 3.0 to VMS 3.4 you may have to do the following:

```
$ BLISS/LIB NILLIB:NILLIB
$ SET DEF [NIL.FOO]
$ BLISS RNIL
$ @RNILLINK
$ NLINK
$ LISPLINK
```

[Step 4]

See the file NIL$DISK:[NIL.COM]NILINSTAL.COM, which can be moved (perhaps edited first) to SYS$MANAGER. Then add this to SYSTARTUP.COM:

```
$@SYS$MANAGER:NILINSTAL
```

Then users who want to use NIL must have executed in their login files:

```
$@NIL$DISK:[NIL.COM]USYMS
```

This will set up the the standard way of calling NIL,

```
$ NIL
```

Which will run NIL as a subprocess. ↑Y or (valret) will exit the subprocess; to resume, type:

```
$ NIL
```

or to kill the subprocess:

```
$ NIL/KILL ! from DCL

(QUIT)      ; from LISP
```

The directory [NIL.SITE] has two files of interest: [NIL.SITE]SITEPARAMS which if it exists in compiled form will be loaded right before the LISPLINK saves the virtual memory image. And [NIL.SITE]DEFAULT.INI, which is loaded at "re-startup" time if the user does not have a SYS$LOGIN:NIL.INI file. After NIL is created on your system then you should edit SITEPARAMS and compile it. The information is noncritical however.

Upon startup NIL will look for the logical-name "TERM" to determine the type of terminal it is connected to. For example:

```
$ Define TERM "vt100"
```

Presently it does all its own cursor positioning using the data in the file NIL$TERMCAP. If the logical name "TERM" is not defined then NIL will prompt the user for the info upon startup.

[Optional Verification]

In NIL do:

```
(LOAD "NIL$DISK:[NIL.VERIFY]VERIFY")
(VERIFY "TEST")
```

Then sit back and watch the little demonstration. No, we do not have program verification technology to the point where this gives a proof of correctness for the NIL. However..., then run

```
$ DIFFERENCES NIL$DISK:[NIL.VERIFY]TEST.LIS
```

[What if Failure?]

If you ran out of disk space in step 1, then we can suggest that you somehow make more space temporarily, (e.g. backup and delete files), and then prune down to the minimum given in NIL$DISK:[NIL.PORT]MINIPORT.COM when step 3 is completed.

Step 2 couldn't fail, as all it does is define logical names and symbols.

Step 3 could fail if various system generation parameters and account quotas are not set high enough. Many sites will fail here, as the default VIRTUALPAGECNT of 8 thousand pages is not sufficient. (Although it is sufficient to do LXBNIL which does not load the compiler). 16 thousand pages is enough to get started in lisp programming. Other things to look out for are insufficient pagefile and per-account pagefile quota.

Default account parameters as supplied by DEC have found to be sufficient under VMS 3.0, but some sites have been found to severly restrict parameters, which has proved to be extremely frustrating at that subset of those sites where the local expertise for debugging problems caused by such restrictions is insufficient.

It is possible to run NIL on a VAX-11/750 with a single RK07 disk, (a mere 27 megabytes!) as we do here at MIT. However, it is not possible to link a NIL on such a tight system. Ideal system environments have been found on sites configured to run large databases efficiently.

In step 4, note that the running NIL image is mostly pure, sharable, code and data, so there is a big performance payoff in proper installation and SYSGEN tuning on a multi-NIL-user system. If LISPLINK works, but NIL does not, then it may be due to insufficient global sections and pages.

[Other Options]
If you want to be able to use the VMS debugger on the running lisp image then execute the following:

```
$RUN [NIL.HACKS]SETDEBUG
```

Giving NIL.$DISK:[NIL.EXE]LISP.EXE; as the filename, and answering Y to the question. With this setting NIL will start up in the VMS debugger, and you must type GO<CR> to actually start it.

27.6 How the NIL Control Works

This section notes how some of the above stuff works, for the interest of VMS hackers, or those wishing to extend the above functionality.

Program control of NIL under VMS works in a fairly strange way (or at least so it will appear to someone used to operating systems in which there is more explicit job/terminal association and more "monitor" control of inferior processes). This is a function of VMSs lack of a concept of a job "having control of the terminal", and the fact that the NIL process does not contain a command language interpreter in its image; the `spawn` and `attach` commands are only implemented by conventions applied by the CLI.

The command `nil` typically invokes the RNIL program. This is an image which runs within the CLI process, and "controls" the NIL, which is kept in a separate process. The `nil` command implicitly supplies lots of arguments to RNIL, one of which is the job name of the NIL process. RNIL will create one if there is none, or will do something else to it (like resuming it) depending on additional arguments given (like `nil/proceed` or `nil/kill`). RNIL communicates to the NIL process with mailboxes. When the NIL is resumed, the RNIL attempts to read from one,

waiting. It returns either when it succeeds in reading a message (as happens with valret), or if it is abnormally exited (as with typing control-Y).

VMS in its current state does not have the concept of a particular process having "control" of the terminal it shares with the rest of the process tree. NIL handles this by having a number of event and state flags which tell it whether or not it is allowed to read from or write to the terminal. When a NIL is exited, the RNIL program clears those flags; when it is resumed, they are turned back on again.

Exiting from NIL with control-Y works in a particularly strange fashion. The VMS terminal driver will give a control-Y AST to any process which has enabled it, with no conceptualization of what program is "in control of" the terminal. The control-Y is handled by the CLI, which then commences image rundown of the RNIL program. RNIL has an exit handler which then sets the terminal input and output enabled flags off in the NIL process. (As a special case, it may also exit similarly if the NIL is terminated some other way, perhaps by `exit` to the VMS debugger in the NIL process. It recognizes the mailbox message for this, and prints "`NIL Terminated`".)

Control-C has a similar control problem. When a control-C is typed on the terminal, the terminal driver runs the AST routines for *all* processes which have enabled them. (Multiply, if a process has enabled more than one.) In the current implementation, the NIL process enables the control-C AST. When the AST routine is run, it attempts to determine if it should be the recipient of that interrupt, by checking to see if it "has control of the terminal" (i.e., the terminal-input-enabled flag is on). If not, it ignores the interrupt (and of course re-enables the control-C ast). If it thinks it was the recipient of the interrupt, it cancels the control-C (to help keep other NILs on the same terminal from having to think about it, I guess), and queues a LISP interrupt for control-C. There is one time when this can break down: if the NIL is suspended when the AST is delivered, the AST will not be run until the NIL is resumed. However, when the NIL is resumed, the RNIL delivers it a couple other ASTs which cause the terminal input and output flags to be turned on! If this manages to happen before the control-C AST routine gets around to checking these flags (as it invariably does), then the NIL will think that this control-C was for it, and behave accordingly. So, if you resume a NIL and it acts like you just typed a control-C, it is probably because of that control-C you typed at the `display` program half an hour ago; type "n" at the ">Interrupt>" prompt to make it go away.

There is a design change which eliminates this problem, and additionally allows controlled interruptibility out of arbitrary wait operations (not just terminal input and output, which are special cased). It involves a sweeping change to lots of code, however, so cannot be put in bits at a time.

28. Peripheral Utilities

This chapter will accumulate documentation on various minor utilities which are distributed with NIL, but which are not necessarily part of NIL proper.

28.1 The Predicate Simplifier

NIL offers a predicate simplifier, which simplifies LISP-format predicates into disjunctive normal form. This program was originally written by Deepak Kapur with the help of Ramesh Patil for the PROTOSYSTEM automatic programming project directed by William Martin at MIT in the mid 1970s. Since then, it has been converted to use LSB [4], brought up in both LISP MACHINE LISP and NIL, and improved at a low level. The code for this is not loaded by default in NIL; it exists as `nil$disk:[nil.utilities]simp`, and to load it, the package definition file `nil$disk:[nil.utilities]simp.pkg` should be loaded first. This may be performed automatically by use of

```
(require 'simp)
```

See require, page 123.

This simplifier only really works on simple predicates and connectives. It performs some trivial canonicalizations of arithmetic operations and inequalities (equal, greaterp, and lessp), but it does not truly recognize identities or other relations among them. There is also a read-time (compile-time) conditionalization for whether it attempts to deal with existential quantification, as represented by forms of the form

```
(for-some (k1 k2 ... kn) pred)
```

This feature is normally turned off, which simplifies the internal datastructures used and improves the efficiency in the other cases. Again, simplification of forms containing existential quantification does not always reduce as well as it should. To get this one would have to recompile simp with that feature turned on.

simp *pred-form*
: Simplifies *pred-form*. For example,

```
(simp '(and c d (or a b)))  =>  (or (and a c d) (and b c d))
```

simpor *p1 p2*
simpand *p1 p2*
: Approximately equivalent to

```
(simp (list and-or-or p1 p2))
```

simpnot *pred*
: Simplifies the *not* of *pred*.

simporlist *pred-list*
simpandlist *pred-list*
: Simplifies the or or and of *pred-list*.

***simpor** *p1 p2*
***simpand** *p1 p2*
p1 and *p2* must already be in disjunctive normal form, i.e., already simplified (as returned by some simplification call). This is faster than using simpor or simpand.

***simporlist** *pred-list*
***simpandlist** *pred-list*
Simplifies the or or and of the predicates in *pred-list*, which must be already simplified.

***simpnot** *pred*
Simplifies the not of *pred*, which must be already simplified.

There is also a hack for doing both uniquizing of predicates returned, and also "atomizing", associating an atomic symbol with a predicate (which will be expanded out in subsequent simplification). The former was important in the PDP-10 MACLISP version when large databases associating predicates with probabilities were in use. See the source code if either of these are desired.

28.2 A Mini-MYCIN

This is a small production rule system upon which class projects in the MIT course 6.871 were implemented. Some students in the course taught this term by Prof. Peter Szolovits and Dr. Ramesh Patil used this code in NIL. The directory NIL$DISK:[MYCIN] has what the students got to start with. This is more of an example lisp program than it is a utility. Here is part of a script of a run of the test example:

```
(load "nil$disk:[mycin]loader")
(load-mycin)
Indeterminate context: RULE4 flushed.
Type (return t)
;bkpt ERROR
1>break>(return t)
Indeterminate context: RULE5 flushed.
Type (return t)
;bkpt ERROR
1>break>(return t)
(run)
Creating new context node: PERSON-2
```

The files GOBBLE.LSP and MYCINF.LSP are the basic system, upon which students built sets of rules to do something useful or interesting. The example above, from MYCINT.LSP is not interesting, just (barely) illustratory. TAXAID.LSP has a completed project a student did in 1980.

28.3 Maclisp Compatibility for Macsyma

The directory NIL$DISK:[MACSYMA] has some code in that is used for compiling and running MACSYMA in NIL and that will be useful to anyone porting a MACLISP program.

The file ALOAD.LSP has an autoloading handler that works by handling the :undefined-function error condition. This might be a generally useful thing to have around.

The file PKGMC.LSP illustrates the use of pkg-create-package and intern-local in order to build a namespace that shadows conflicting or incompatibly defined functions and variables.

The file NILCOM.LSP gives definitions of the functions map, subst, member, and assoc, which are compatible with MACLISP.

29. Foreign Language Interface

29.1 Introduction

It is desireable to be able to call from NIL procedures that are written in other VMS supported languages, such as FORTRAN, COBOL, PL1, BLISS, C, PASCAL, lan{Basic}, et. al., not to mention procedures written in MACRO32, and VMS library routines and system services. Fortunately this is easy, due to the the uniform VMS object and symbol table file format, uniform procedure call mechanism, and rich set of NIL datatypes from which to construct datastructures compatible with what various foreign language routines expect to receive.

The presently implemented interface is by no means the last word in such endevors; for example it makes no attempt to enforce datatype restrictions in argument passing; however, it is found to be functional, and is used in the NIL system itself to access some VMS system services, to incrementaly debug parts of the assembly-language kernal, and to interface to "number-crunching" FORTRAN subroutines and to some users existing C libraries.

29.2 Kernel and System-Services

The executable code for such procedures is already in the lisp process address space, therefore accessing them is only a matter of defining an argument-data-convention interface, searching the lisp or system symbol table to get the required machine address, and creating a lisp subr trampoline, similar to an element of a transfer vectors the VMS linker would create when one references sharable libraries.

si:defsyscall (*lisp-name vms-symbol*) &rest *argumentspecs*

Does everything needed to reference a routine in "LISP.STB" or "SYS.STB". The lisp-name is defined as a special-form taking alternating named arguments as in a defstruct defined constuctor. For example, the routine to convert a vms error code into a human-readable string:

```
(defsyscall ($getmsg sys$getmsg)
  (msgid :in :long :required)
  (msglen :out :word :required)
  (bufadr :out :string :required)
  (flags :in :byte)
  (outadr :in :bits))

(defun decode-vms-error-code (loss-code &optional (flags 15)
                                          &aux len)
  (using-resource (string-buffer string 256)
    ($getmsg msgid loss-code msglen len
             bufadr string flags flags)
```

N.B. Calls to SI:DEFSYSCALL, and to many other system internal primitives work when *compiled*, but not when intepreted. The example above is from code in the systems-internals package.

29.3 VMS object files

To call a procedure in a vms object file the user must do three things, define an argument interface, call the dynamic loader, and enable the trampolines for specific procedures. For example:

```
(def-vms-call-interface myfoo)

(defun hack-foo ()
  (list (hack-vms-object-file "[gjc.nil]footest")
        (enable-vms-call-trampoline
           'myfoo 'foo "[gjc.nil]footest.stb")))

(hack-foo)

! Sets up for this BLISS

MODULE FOOTEST =
BEGIN
GLOBAL ROUTINE FOO = 259;
END
ELUDOM
```

si:def-vms-call-interface *name* &rest *arglist*
Same as defsyscall, but doesn't actually look into any symbol table or create any trampoline. Only works when compiled.

si:hack-vms-object-file *obj-file*
Calls the VMS linker on a single object file, and then reads the executable code into a bitstring in the lisp address space. Presently a VMS subprocess interface is not implemented, (which is the easiest way for lisp to invoke the VMS linker), so instead the user is asked to execute a VMS command file lisp writes. This happens twice for every file so hacked. What a kludge.

si:enable-vms-call-trampoline *name vms-symbol stb-file*
Sets up the trampline for name using the address of the vms-symbol from the stb-file.

29.4 Data Conversion

At a certain level it helps to know the data representations supported by the VAX hardware itself, and what representations the various language compilers, including lisp, build on this base. Lets face it, at this point, unless you are willing to deal with such issues its best to forward specific interface requests to the implementors, and we'll try to at least provide a family of existing examples which should make things obvious, or presolved. Even though the macrology provided by defsyscall et al. may make it easy, it by no means makes things foolproof, as any such excursions outside the lisp-world-firewall we set up are frought with frustrating debugging problems.

In garbage collection, the system will not be forgiving of any violation of the rules of register and stack usage, and raw address placement.

29.5 lower level routines

As if the ones above weren't low-level enough.

si:locate-symbol-table-value *symbol* &rest *stb-filenames*
Returns null or a fixnum.

si:construct-system-symbol-trampoline *hi8-bits lo24-bits*
Returns a trampoline subr which jumps to the address specified.

30. What Will Break

Various changes are anticipated for future releases of NIL, just as some have taken place for this release. This chapter notes some of the significant implementation changes which have already occurred, and describes some which are anticipated.

30.1 What Broke Since Release 0.259

The single most significant change since release 0.259 is that there are now four different formats of floating-point numbers. Because the new formats do not appear spontaneously, and the default conversion from rational to floating-point still produces **double-float**, it is unlikely that this will be particularly noticeable. About the only possible point of lossage here is that the value of the system constant **pi** (page 76) is now a **long-float**. This is in keeping with the philosophy that such constants should be in the longest format possible, and that uses which require a shorter format can use a form like

```
(float pi 0.0d0)
```

or

```
(coerce pi 'double-float)
```

which will perform the coercion at compile time. Most other changes to arithmetic have been additions, extensions, and fixes. For instance, the COMMON LISP division and conversion-to-integer functions **floor**, **ceiling**, **truncate**, and **round** now exist. Certain routines (most notably those four "division" routines, and **numerator**, **denominator**, **gcd**, **oddp**, **evenp**, **rational**, and **rationalize**) have been extended to complex numbers, in order to allow manipulations on gaussian rationals. See chapter 10, page 71, on numbers, for complete information.

The use of **nil** in a lambda-list as a placeholder for an unused var is now discouraged; the compiler will issue a warning about it. The variable should be named, and then declared with the **ignore** declaration, as in

```
(multiple-value-bind (foo bar baz) (mumbledy-frotz)
  (declare (ignore bar))
  ...)
```

A slight change has been made to the **loop** macro. The syntax

```
(loop for x in form1 form2
      do ...)
```

which is interpreted as

```
(loop for x in (progn form1 form2)
      do ...)
```

is being phased out, *except* for those "clauses" which are run totally for effect, not for value. When multiple expressions are encountered in a place where a value-returning expression is expected, a warning is issued, but the expansion is the same as before. The only "clauses" for which such multiple expressions will remain valid are **initially**, **finally**, and **do**.

There are a number of other small isolated changes which are not worth listing here, but should generally be evident when the code is compiled, having to do with what names are preferred for certain low-level functions, and corrections to what keyworded arguments certain functions take.

Pathname stuff which used to give you SYS$DISK as a device name now (correctly) uses its translation (specifically, user-workingdir-pathname). This is mentioned because it was predicted in the last release.

The length (page 49) and list-length (page 59) functions have been changed slightly. Neither finds a non-null termination of the list acceptable any more; this is in keeping with the COMMON LISP definition, and (for length) is more consistent with its use as a general sequence length function (a list with a non-null end is not considered a proper sequence). length continues to give an error if the list is circular; list-length will return nil when given a circular list.

Certain defaults within defstruct (page 125) have changed, as have certain conventions. The default type of a defstruct-defined structure is now a typed object; objects of this type can be checked for with typep by using the name of the structure as the type name. The :conc-name option is now on by default, and one must specify (:conc-name nil) to get the old behaviour. Finally, the keywords specified to keyworded constructor macros, and to alterant macros, are intended to be keyword symbols (interned in the keyword package), e.g.,

```
(make-person :head head :name name)
```

although non-keyword symbols will be accepted in the interim.

Use of "destructuring" with let is being phased out. Use the function *not-yet-written* instead; see *not-yet-written*.

Declaration scoping for special forms is slightly different than it used to be. It is, in fact, somewhat more consistent than before. Chapter 6, which is new, describes declarations in detail. The main difference is that declarations now scope over the "init forms" in special forms that have them, like let, let*, do, prog, etc. That is, the *dcl-specs* in

```
(let ((var-1 val-1) (var-2 val-2))
  (declare dcl-spec-1 dcl-spec-2 ...)
  forms...)
```

that do not pertain to bindings of the variables *var-1* and *var-2*, are "in force" not only during the evaluation of *forms*, but of *val-1* and *val-2*. The major consequence of this is that, with declarations, let cannot be trivially turned into a lambda expression without parsing the declarations. In the expression

```
((lambda (var-1 var-2)
   (declare dcl-spec-1 dcl-spec-2 ...)
   forms...)
 val-1 val-2)
```

the declarations have *no* influence whatsoever on the evaluation of *val-1* and *val-2*.

30.2 Future Changes

Some of these are the same as those anticipated as of Release 0.

30.2.1 Default Floating-Point Format

The default floating point format is going to change from the current **double-float** to **single-float**. This will almost certainly happen with the next release. The format of course could not be changed with this release and maintain any semblance of upwards compatibility, because the *previous* release did not support **single-float** format. Because of contagion rules, there are really just two ways one might start generating single-floats instead of double-floats: when reading, and when performing some irrational, transcendental, or otherwise awful function on rational inputs. If in fact the change matters at all for an application (it might not if only generic arithmetic was being used), then one could avoid problems by judicious use of explicit conversion to floating point, and by being careful to explicitly specify the exponentiation character in LISP source files so that the datatype will be explicit.

30.2.2 New Package Facility

The COMMON LISP package definition is finally final, and should appear in the next release. It is unlikely that anything other than code which operates on or with packages explicitly will have to be changed, with the possible exception of references to "internal" in one package made from another package.

30.2.3 Vector-push and Vector-push-extend

The argument order to **vector-push** and **vector-push-extend** will be changed so as not to be unmnemonically different from that of **push**. Currently, the vector is the first argument, and the object to push is the second; these two will be reversed. This change became known just as release 0.259 became ready, and got lost or forgotten so never made it into this release. Ah well.

30.2.4 Multiple Values

In the future, NIL will "natively" support a multiple-value return mechanism. For it to do so requires that the compiler understand them at a moderately low level; it will be producing code for receipt of them from function calls, it will have to flag function calls which will be passing them back to another caller, and it must recognize and compile away all local multiple-value passing.

The mechanism, which has only been designed at a fairly high level, is this.

Given the compiler behaviour described above, the only place which compiled code can receive multiple values from is a function call (or a non-lexical **throw**, but we will ignore this for the sake of simplicity in this description). This means, that when multiple values are being passed back (returned or generated), if we can recognize those function calls which are simply being made to pass back any values to their callers (and thus also recognize those which are expecting

some value or values in particular), we can trace up the stack to find the call which is ultimately expecting the multiple values. (The function call frames are quite formal and stylized in NIL.)

So what we do is to have the caller which is expecting multiple values allocate a place for them on the stack and put a marker there, before it allocates the call frame for the function it is about to call.

We have all function calls which simply pass back their values as the value of the function they are contained in, marked as such, so that examination of the call can determine this. In the following, the calls to foo and bar would be so marked, but baz would not:

```
(defun frobnicate (x y)
   (if (zerop x) (foo y)
       (mvprog1 (bar x (sub1 y))
                (baz (sub1 x) y))))
```

If (say) within bar there is a call (values *this that*), then a special subroutine goes looking up the stack, finds the frame where bar is called, sees that it passes back its values out of its calling function, so traces the function frame pointer to the caller of frobnicate, etc.

If, on the other hand, with baz there is a similar values call, tracing back to that call to baz reveals that baz is expected to return only one value (of interest, at most), so no further tracing is done.

There are two further points of interest about this scheme. By appropriate use of specialized markers where multiple values *are* expected, fast dispatching may be performed for dealing with various situations, such as multiple-value-list, for instance.

A somewhat kludgey extension is to use this for things like "number calling"—one routine calling another for (say) flonum value. The one producing the flonum, instead of consing it, looks back and if the final destination is expecting a flonum in some special way (having, for instance, pre-allocated a space on the stack for it), then the representation is stored there without consing, otherwise the value is consed in the heap and passed back via the normal value return mechanism. The kludge involved here is that if the producer is interpreted code, someone has to coerce the *normal* value return into the hacked one. This only is necessary when such a compiled routine is calling into interpreted code, so will probably be done by the interpreter-trapping subroutine.

The interpreter-trap wrapper must be capable of recognizing when a value *has* been stored "properly" into the compiled receiver, because if the *producer* is compiled and the interpreter has produced the value such that it got passed back "naturally", then it has been stored already without being consed, even with intervening interpretation!

It is of note that this stack-searching is not directly analogous to a deep-binding variable-binding scheme, in terms of efficiency and paging overhead etc., because in the variable binding scheme the searching must be done up the entire stack (or alist or whatever) every time, the time for each search growing in proportion to the depth of the stack, but for this the search terminates whenever values are not being passed back to the previous caller.

30.2.5 Variable Naming Conventions

COMMON LISP is establishing a uniform naming convention for system-defined parameters and constants. Essentially, all system-defined parameters (those variables whose values are allowed to be changed, i.e. that parameterize the behavior of the system) will have asterisks (*) at each end of their names. Thus, the variable base will become *base* (NIL uses si:standard-output-radix now anyway), and package will become *package*.

All system-defined constants, such as char-code-limit, will not. Part of the justification for this is that the compiler and interpreter should be able to determine when one is modifying a constant, but not a parameter, so the constants require less visual distinction. This is in fact currently the case, as defconstant (page 24) now works.

The change of these variables is indeed going to be catastrophic to both users and the NIL system itself. Note, however, that one may do (say)

```
(who-calls 'package :type :value)
```

to find all modules (i.e., restricted to compiled code) which reference the special value cell of package.

30.2.6 Garbage Collection

When the garbage-collector is finished, there will be two major incompatibilities noticeable. First, the format of compiled output files will change. Although initially (for bootstrap and debugging purposes) old format files will be accepted, it is unlikely that this will still be the case by the time the garbage collector is released. Even if it is the case that such old files can be loaded, the garbage-collector will not be able to safely run afterwards. Second, there is an incompatible change which must be made to the way unwind-protect is compiled. This cannot be handled upwards-compatibly, as it involves compiler knowledge about stack usage from the NIL kernel, so old code might not be able to run correctly. Recompilation, of course, will fix everything.

Obviously, when there is a garbage-collector, dirty operations like playing with addresses and changing types become substantially more dangerous, and should be avoided by all code except for the garbage-collector itself.

30.2.7 Error System

A new error system and debugger interface is being designed. The arguments to error, cerror, and/or ferror may be changed incompatibly, although it is hoped that old uses will be able to be distinguished from new uses. condition-bind uses will have to be recompiled. Condition names may work upwards-compatibly, however. Note that now, signal erroneously forces entry to the debugger if the condition goes unhandled; this will be changed. To enhance the ability of future code to detect old uses, a few conventions may be helpful:

(1) The first two arguments to cerror should always be t or nil.

(2) Always use a string for the "error string" or "format string" for all three error functions. (MACLISP-compatible use of error can get by without this if the "string" is a symbol, but contains at least one space in its text.)

The future debugger, which is mostly complete now but needs the new error system, will be much better able to parse the stack in use by compiled code. This will include the ability to recognize data on the stack which is not LISP objects but rather binary data, show which arguments to pending function calls have not been computed, etc. To the extent that the compiler leaves around more specific information, the debugger will be able to show typed values for the binary data (for example intermediate or local-variable floating point values on the stack).

References

1. Steele, G. L., *Common Lisp Reference Manual*, Carnegie-Mellon University Department of Computer Science Spice Project, (in preparation). Actually, it's now in-press with Digital Press.

2. Steele, G. L., *et al.*, *An Overview of Common LISP*, paper presented at 1982 ACM symposium on LISP and Functional Programming. 1982 ACM 0-89791-082-6/82/008/0098

3. Bawden, A., Burke, G. S., and Hoffman, C. W., *Maclisp Extensions*, MIT Laboratory for Computer Science, Cambridge, Mass. TM-203, July 1981.

4. Burke, G. S., *LSB Manual*, TM-200, MIT Laboratory for Computer Science, Cambridge, Mass., (June 1981).

5. Burke, G. S. ,and Moon, D., *Loop Iteration Macro*, TM-169, MIT Laboratory for Computer Science, Cambridge, Mass., (January 1981). (Revision in preparation.)

6. Crocker, David H. (revised by). *Standard for the Format of ARPA Internet Text Messages*, Network Information Center Request For Comments; SRI Internaltional, Menlo Park, CA, (August 1982).

7. Hawkinson, L. B., and Burke, G. S. Unfinished, unpublished memo/documentation on the pretty-printer noted in section 19.5, page 186; copies of an interim version are included with NIL distribution kits.

8. Mathlab Group, *Macsyma Reference Manual*, MIT Laboratory for Computer Science, Cambridge, Mass., (1977).

9. Moon, D. A., *MACLISP Reference Manual*, MIT Laboratory for Computer Science, Cambridge, Mass., (1974).

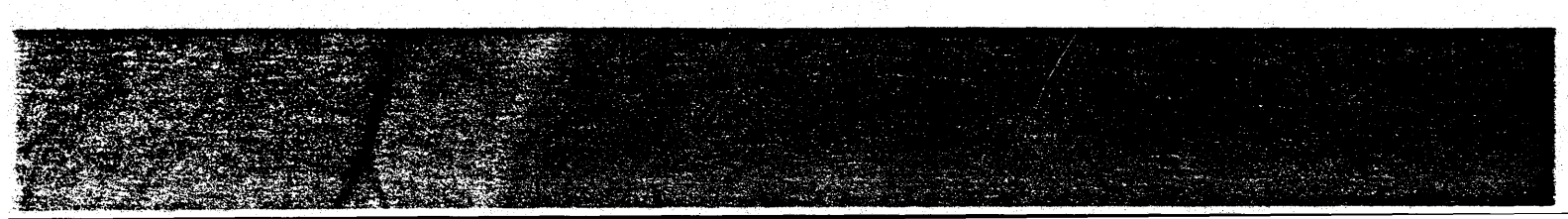
10. Postel, Jon, and Harrenstien, Ken, *Time Protocol*, Network Information Center Request For Comments; SRI Internaltional, Menlo Park, CA, (May 1983).

11. Pratt, Vaughan R., *CGOL - an Alternative External Representation for Lisp users*, AI Working Paper 121, (March 1976)

12. Weinreb, D., and Moon, D., *Lisp Machine Manual*, MIT Artificial Intelligence Laboratory, Cambridge, Mass., (July 1981).

13. White, J. L., *Constant Time Interpretation for Shallow-bound variables in the Presence of Mixed SPECIAL/LOCAL Declarations*, paper presented at 1982 ACM symposium on LISP and Functional Programming. 1982 ACM 0-89791-082-6/82/008/0196

Concept Index



Message Index

Resource Index

Variable and Constant Index

Function, Macro, and Special Form Index