NIL Notes

for

Release 0.259

June 1983

Glenn S. Burke

George J. Carrette

Christopher R. Eliot

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for this research was provided in part by the National Institutes of Health grant no. 1 P01 LM 03374-04 from the National Library of Medicine, the U. S. Air Force under grant F49620-79-C-020, the National Aeronautics and Space Administration under grant NSG 1323, the U. S. Department of Energy under grant ET-78-C-02-4687, and the 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 *flavors* was written in part by Patrick Sobalvarro. The editor and its documentation is 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.

As this document is gradually transforming from "release notes" to a "manual", some descriptive segments have been lifted from the *NIL Primer* written by GSB, which is included with the NIL distribution.

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-ML.

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.

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 "final" draft of the COMMON LISP manual is not yet ready; for this reason, this document is in its present form; eventually, the NIL manual will be the COMMON LISP manual, supplemented by facilities provided only by NIL, and by other COMMON LISP facilities which are not part of the core COMMON LISP language requirement.

This is an interim document. The NIL language is in the process of being converted to COMMON LISP, as it is known to be at the time. Certain significant conversions have been performed since NIL Release 0; for instance, the "single-letter" arithmetic functions, such as + and -, are now generic (they used to be fixnum-only, as in MACLISP). The basic types for representing ratios (non-integer rational numbers) and complex numbers have been added, as has a new array implementation which provides the basis for conforming to the COMMON LISP array specification. 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).

These notes are designated as *Revision 0* of the notes for Release 0.259. Although they are largely based on the previous notes (Revision 1 of the notes for Release 0), they have changed too drastically (as have some parts of NIL) to be considered a simple revision of the previous notes.

This document is intended to be supplemented with the *Maclisp Extensions Manual* [3], which documents some facilities which behave pretty much the same in NIL, and have thus been largely omitted from this for expediency. Other revisions of these notes will appear informally from time to time, until such time as the COMMON LISP manual is publicly available, and the NIL documentation can be made to complement that.

So, read on. Hopefully grains of salt will not be in too much demand.

Oh, one last very significant note: *unlike most every other Lisp manual emmanating from MIT over recent years, unqualified numbers in this one are DECIMAL, not OCTAL.* NIL defaults both the input and output radices to decimal.

2. Data Types

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 FORTRAN floating-point. Note that there is no guarantee that the above types might not be further subdivided 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.

Integers are ordinarily represented in decimal notation, optionally preceeded by a sign character and optionally followed by a decimal point. See also chapter 3 of [3] which provides additional syntax for reading integers in other radices.

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.

2.1.2 Floating-point Numbers

NIL currently offers one floating-point format, which is double-precision, having an 8-bit exponent and 56-bit mantissa (significand) (including the hidden bit). This type is **double-float**. For various historical reasons, it is also called **flonum** (from Maclisp). Note that in NIL (as in all MIT lisp dialects), suffixing a sequence of digits by a decimal point does not produce a floating-point number, but rather forces the integer to read in in radix 10; to force floating-point, the decimal point must be followed by digits. Thus, "10." is the fixnum ten, but "10.0" is floating-point ten. The other syntax is to use exponential notation, as in 1.0e+10. In this too, in NIL, at least one digit is required after the decimal point, although none are required *before* the decimal point.

COMMON LISP allows for at least four kinds of floating-point representations, which must meet the following criteria (note that this differs from what was in the previous release notes!):

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

Eventually, NIL will provide 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 will require microcode support to avoid software emulation.

The various forms 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***; this should only be set to **double-float** right now, although eventually it will default to **single-float**.

2.1.3 Complex Numbers

#C(*realpart imagparg*)

The real and imaginary parts may be extracted with the **realpart** and **imagpart** functions. Only the basic arithmetic functions handle complex numbers currently.

Note that although computations with ratios get automatically reduced to integers when possible, and thus one should not see an object of type **ratio** with a denominator of **1**, it is possible to have a complex number with a zero imaginary part. In the future, complex numbers will be restricted to have real and imaginary types which are either both rational, or both floating-point numbers of the same format. Additionally, *gaussian rationals* with a zero imaginary part will always be reduced to ordinary rational numbers and will not be of type **complex**.

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 10 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

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.

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 *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 14, page 83).

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.

2.4 Lists and Conses

some basic lisp here

2.5 Arrays

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-charp** function, page 15) 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 11, page 67.

2.6 Structures

NIL provides a *structure* or *record* definition facility. This is supplied by the **defstruct** function (page 85), which is essentially the same one used in both MACLISP and LISP MACHINE LISP. In NIL, **defstruct**-defined objects are capable of being of their own type (by use of the **:extend** option to **defstruct**)—that is, **defstruct** provides a way for users to define their own types distinguishable by **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

2.8 Randoms

Internally used types.

2.8.1 Minisubrs

Pointers to special subroutines (not procedures). Stripping off the type bits results in the address of the subroutine, which is called via JSB. But argument-passing conventions vary.

2.8.2 Modules

A *module*, as a type, is a logical unit of compiled subroutines 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 name **module** should not be used for this purpose, so it will probably be changed in a future release.

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
```

This is almost the same as what used to be called the type **constant**; *internal-marker* excludes the null object.

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.

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 (perhaps for use by **assoc**), and returns an association list associating all of those keys with the same single value. 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 in order. *#'* 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 18). 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 15).

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 [6].

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 keywords which begin with the character &. 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. For example, 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. The name of the variable is used to generate the keyword which flags that particular parameter. For instance, fill is defined with the lambda-list

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

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, and is rarely used.

&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-form*s. 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 19) or **let*** (page 20); 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 a *simple general 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 care, if at all.

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 107), by defstruct with the :extend option (page 85), 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 given type specifiers.

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

(member *x1 x2 ... xn*)
This defines a type which is one of the objects *x1*, *x2*, ... *xn*. Equality is defined by eql (page 16).

(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*)
Hairier. Probably not worthwhile using yet.

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*.

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

of-type *object type-specifier*
This is the name NIL gave to two-argument (only) typep. Use of typep is preferred.

4.1.3 Specific Type Predicates

This first group are convenient, fast, supported in COMMON LISP, etc.

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 20), which is functionally equivalent because of the empty-list/boolean-false duality of nil, is used to test for boolean falsity. 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*)).

pairp *object*
Old NIL name for consp.

listp *object*
consp or null.

fixnump *object*
characterp *object*
These test for the exact types fixnum and character. Use of typep with a second argument of fixnum or character is preferred.

flonump *object*

Archaic and obsolete predicate for determining if *object* is of type double-float. Use of floatp (for floating-point numbers of *any* format), or (typep *object* 'double-float) is preferred.

string-charp *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 the function string-charp requires its argument to be a character, as opposed to just any object.

In NIL, this is characters with *bits* and *font* of 0 (see chapter 10).

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. They are provided mainly for MACLISP compatibility.

bigp *object*

MACLISP compatibility: (typep *object* 'bignum).

closurep *object*

Tells if *object* is a closure.

4.2 Equality Predicates

Note also null (page 14) and not (page 20), 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. There are certain kinds of objects which have no 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 22).

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 equal if they have the same element-type constraints, rank, and dimensions, and if all the corresponding components are **equal**. This means that that strings are *not* equal if any corresponding characters differ in case; in this, **equal** differs from the LISP MACHINE LISP definition, but is compatible with COMMON LISP. It also means that a string is not **equal** to a non-string vector which contains the same characters as the string.

5. Programming Constructs

The NIL special and toplevel forms.

5.1 Definition Forms

5.1.1 Defining Functions

defun *name bvl* {*declaration*}* [*documentation*] {*form*}* *Special Form*

Defines *name* as a function.

If *bvl* is the atom **macro**, then this is assumed to be a MACLISP-style macro definition, and is transformed appropriately.

If *bvl* 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).

Otherwise, *name* is defined as a lambda-expression with *bvl* as its lambda-list and the declarations and forms as its body.

Until full function specs are implemented, *name* may only be a symbol, in which case the dynamic function definition is set, or a two-list of the form (*name propname*), in which case the function is placed on the *propname* property of *name*. In principle, *name* is a general function-spec.

5.1.2 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*}* *Special 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 *form*s 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. 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)
  ...)
```

More details on the syntax is available in the *Maclisp Extensions Manual* [3]. Especially useful with macros is *backquote*, which is also documented there.

macro *name bvl* {*declarations*}* {*documentation*}* {*form*}* *Special Form*

Primitive macro definition. You probably shouldn't use this, at least not for routine macro definitions.

5.1.3 Defining Variables

defvar *var* [*init* [*documentation*]] *Special Form*

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*] *Special Form*

This is like defvar, only *var* is *always* set to the value of *init*.

defconstant *var init* [*documentation*] *Special Form*

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*.

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 be using defparameter.

special {*variable*}* *Special Form*

Globally declares each *variable* to be special. Note that this normally occurs within a declare special form. Within a declare form, when processed by the compiler, the declaration is made local to the compilation of that file. As a toplevel special form however, in addition to being declared local to that compilation, the form will be output into the compiled file so will make the declaration when that file is loaded (just as defvar and defparameter do).

This is documented because you should be able to write defvar yourself. special, as a special form, is not in COMMON LISP.

5.1.4 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). The canonical MACLISP way to do this is to return a progn special form, the first element of which is the form (quote compile). Although it is unclear that such a special kludge will always be required, it is certainly always *safe* to use such a (progn 'compile ...) form. Note that (progn 'compile *forms...*) is treated the same as (eval-when (load) *forms...*).

A simplified defparameter, which did not handle documentation, could have been written like this:

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

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

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

5.2 Binding

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

let evaluates all of the *values*, and then binds all of the *variables* to the corresponding values. The *forms* are then evaluated in that environment, and the result(s) of the last *form* is the result(s) of the let. For instance,

```
(let ((a form1) (b form2))
  compute)
==>
((lambda (a b) compute)
 form1 form2)
```

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 the interface to destructuring used by **defmacro**; see it, page 17, for more information on 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:

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

let* ({(*variable value*)}*) {*declaration*}* {*form*}* *Special 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 *value*s are evaluated in that environment. For instance,

```
(let* ((a form1) (b form2))
  compute)
 ==>
((lambda (a)
   ((lambda (b)
      compute)
    form2))
 form1)
```

5.3 Conditionals

if *predicate consequent* [*elseform*] *Special Form*

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 14), 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*}*)}* *Special Form*

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 *predicates* are false, then cond returns nil.

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

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

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

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

and {*form*}* *Special Form*

Evaluates each *form*, and if any returns nil, and immediately returns nil without evaluating any subsequent forms. (and) => t.

or {*form*}* *Special 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*}*)}* *Special Form*

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 16.)

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

Note that this function is what **selectq** thought it would once be, and may be used in place of MACLISP **caseq**.

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

typecase examines each of its "clauses" in turn. If *object* is of the type specified by that *type-specifier* (see **typep**, page 14), then the *forms* in that clause are evaluated, and the value of the last form is returned by **typecase**. Note that a *type-specifier* of **t** will always cause the corresponding clause to be selected.

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.

5.4 Iteration Constructs

5.4.1 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 (subl)
          (rplacd subl (delete (car subl) (cdr subl))))
      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 39).

mapcar and maplist each returns a list of the results of applying *function* to the 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*.

5.4.2 Special Iteration Forms

dotimes (*var count*) {*declaration*}* *body...* *Special Form*

Evaluates the forms in *body* in an environment where *var* is stepped from 0 up to (but not including) the value of *count*. *body* is actually a **tagbody** body, and **dotimes** establishes an implicit block named **nil**, thus **return** may be used to return a value from the **dotimes** before the iteration terminates; see section 5.4.3, page 24.

dolist (*var list*) {*declaration*}* *body...* *Special Form*

body is evaluated with *var* bound to the successive elements of the value of the form *list*. *body* is actually a **tagbody** body, and **dolist** establishes an implicit block named **nil**, thus **return** may be used to return a value from the **dolist** before the iteration terminates; see section 5.4.3, page 24.

dovector (*var vector*) {*declaration*}* *body...* *Special Form*

Similar to **dolist**, but for vectors.

loop *gubbish...* *Macro*

loop is described in another document [5]. It is noted here because it deals correctly with the incompatibilities between LISP MACHINE LISP and COMMON LISP blocks and **returns**.

loop will probably be changed incompatibly by COMMON LISP; the first edition of the COMMON LISP manual will not, however, have a specification for **loop**. The **loop** currently in NIL is the same one described in [5], and that documentation is quite accurate, even though it was written before **loop** was really introduced into NIL.

5.4.3 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*}* *Special 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* *Special 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* *Special 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*}* *Special 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* *Special Form*

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*}* *Special 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.

5.5 Non-Local Flow of Control

catch *tag* {*form*}* *Special Form*

COMMON LISP **catch**. *tag* is evaluated and the result saved. Then, if during the evaluation of the *form*s, 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* *Special 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*}* *Special Form*

protected-form is evaluated, and the result returned. Upon exit, the *cleanup-form*s 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 *clean.:p-form*s.

***catch** *tag {form}** *Special 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* *Special 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.

5.6 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))
```

multiple-value *variables values-form* *Special 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. 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* {*form*}* *Special 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* *Special 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
```

5.7 SETF

setf {*place value*}* *Macro*

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.

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, symeval, fsymeval, 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 a 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 return value from this expansion is incorrect in the current implementation, in that it actually does expand as shown.*

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.

push *object place* *Macro*

Approximately

```
(setf place (cons object place))
```

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

pop *place* *Macro*

Approximately

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

but the forms in *place* are evaluated only once.

There are a mess of other other such macros COMMON LISP defines, but they are deferred until the new setf in some future release of NIL. They are incf and decf (incrementing and decrementing the *place*), (shiftf *place1 place2 ... placen value*), which returns the original value of *place1*, stores the value of *place2* into *place1*, ... and *value* in *placen*. Also rotatef which is like shiftf but gets the *value* from *place1*.

6. Lists

Note also consp (page 14), listp (page 14).

6.1 Creating, Accessing, and Modifying List Structure

Note also the ' reader-macro (chapter 2 of [3]), which is convenient for creating list structure in template form, especially if large portions of it are constant.

car
cdr
c....r

rplaca *cons new-car*
rplacd *cons new-cdr*
See also setf (section 5.7, page 28) which can be used to update any of the above car/cdr references.

See also push (page 29) and pop (page 29) which can be used to maintain a list in FIFO form in an arbitrary setfable 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*).

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*

Creates a list of nils *size-of-list* long.

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.

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 '(a b . c))  =>  (b . c)
```

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,

```
a                      =>  (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.

list-length *list*

The list-specific length (page 37) function. In NIL, there is no particular benefit to using list-length over length other than to verify that the argument is a list (as opposed to a more general sequence); since length is the way to obtain the length of a list in MACLISP, programs which wish to remain MACLISP-compatible may use length without penalty.

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.

copylist *list*
Old name for copy-list.

copy-alist *alist*
Like copy-list, and additionally, each top-level element of *alist* which is a cons has that first cons copied also.

copyalist *alist*
Old name for copy-alist.

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 broken by COMMON LISP.

copytree *tree*
Old name for copy-tree.

first *list*
second *list*
third *list*
fourth *list*
fifth *list*
sixth *list*
seventh *list*
eighth *list*
ninth *list*
tenth *list*
car, cadr, etc.

rest *list*
cdr

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 differences from elt.

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

6.2 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 the previous 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 32). Be on the lookout for the use of this idiom in old code, for it can cause obscure bugs when uncopied structure is modified.

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)
```

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

6.3 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 :test :test-not :key

Returns the union of *list1* and *list2*.

intersection *list1 list2* &key :test :test-not :key

Returns the intersection of *list1* and *list2*.

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).

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.

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*. *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*, so modifications to the associations in the returned result will affect those assocations in *a-list* also.

assq *item a-list*

This is assoc, but uses #'eq as its test. assq is not defined by COMMON LISP, but is inherited from MACLISP. It is specially handled by the NIL compiler.

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. That is,

```
(if (member item list test-keyworded-args)
    list
    (cons item list))
```

6.4 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.

There are two different ways in which association lists may be used. In one, the association list is not considered to have more than one association with "the same" key (as determined by a user-specified predicate, as with many other NIL functions). In this case, it may be better to use hash tables (chapter 13, page 81), unless the number of entries is kept fairly small. The other is where the implicit ordering of a list comes into play. One entry in the association list might have the same key as another; 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.

assoc *item a-list* &key :test :test-not :key
Searches *a-list* for an association whose key 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 that if a function is specified with :key, it is only applied to the keys in the a-list, not to *item*.

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)
```

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.

7. Sequences

A sequence is considered to be either a list or a vector (which is by definition a one-dimensional array). A few functions which might be useful are presented here.

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.

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)
```

length *sequence*

Returns the length of *sequence*. In NIL, length will detect a circular list, and signal an error.

7.1 Creating New Sequences

make-sequence *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 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.

copy-seq *sequence*

Copies the sequence *sequence*. This might be necessary if the result is going to be modified, for instance.

7.2 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 :start :end :start1 :end1 :start2 :end2

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

```
(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)
```

start and *end* are used as defaults for *start1*, *end1*, *start2*, and *end2*, which otherwise default to 0 and the lengths of the corresponding sequences. The number of elements transferred is the length of the shorter subsequence.

7.3 Iteration over Sequences

The following functions iterate a user-specified function over one or more sequences in various ways. They are not handled specially by the NIL compiler yet, however, so cases where the function is a #'(lambda ...) form will not compile if the lambda expression contains free references to lexical variables in the containing code.

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* (cf. **make-sequence**), 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))
  => ((#\a . a) (#\b . b) (#\c . c))
```

If *result-type* is **list**, and the input sequences are all lists, then this is the same as **mapcar** (page 23).

some *predicate* &rest *sequences*

If the result of applying *predicate* to the corresponding elements of *sequences* is not **nil**, that value is returned; otherwise, **nil** is returned. The predicate is called with as many arguments as there are sequences; first on all of the first (index 0) elements, etc. Only the parts of the sequences up to the length of the shortest are considered.

every *predicate* &rest *sequences*

Like **some**, but returns **t** if the result of applying *predicate* to the elements of *sequences* is never **nil**, **nil** otherwise.

notany *predicate* &rest *sequences*

Returns **t** if the result of applying *predicate* to the corresponding elements of *sequences* is always **nil**, **nil** otherwise.

notevery *predicate* &rest *sequences*

...

7.4 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. Symbols

8.1 The Property List

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

plist *symbol*
symbol-plist.

setplist *symbol newplist*
Sets the property list of *symbol* to be *newplist* Note that unlike in MACLISP, setplist does not work on disembodied property lists, only symbols.

The proper COMMON LISP style is to use setf of the symbol-plist.

get *symbol indicator*
Standard MACLISP get. *symbol* may actually be a disembodied property list. COMMON LISP will be adding other primitives for dealing with symbols and property-list associations in other structures.

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.

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.

putprop *symbol value indicator*
Standard MACLISP putprop. If *symbol* already has an *indicator* property, this replaces it with *value*, otherwise puts a new one.

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.

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.

8.2 The Print Name

symbol-print-name *symbol*
Returns the print-name of *symbol*.

Note that print-names of symbols should never ever be modified.

In NIL, the print-name of a symbol will always be a "simple" string, never a general character sequence or even a string of adjustable size.

The name may be shortened to symbol-name. Which is used hardly matters.

get-pname *symbol*
Superceded by COMMON LISP symbol-print-name.

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.

8.3 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.

copysymbol *sym* &optional *copy-props*
If *copy-props* is nil, then this is the same as (make-symbol (get-pname *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.

The COMMON LISP name of this, like other copy*something* functions, is copy-symbol. The name copysymbol is MACLISP compatible (but in MACLISP the second argument is not optional). copysymbol 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.

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 *symbol*) sets the prefix to be the first character of *symbol*, and then makes a gensym.

(gensym *integer*) sets the counter to *integer*, and then makes a gensym. Actually, *integer* must be a non-negative fixnum.

gentemp &optional *prefix*

Hairer form of gensym. This is not strictly compatible with the COMMON LISP definition, so i won't be more specific. The point, however, is that an *interned* symbol is generated, which may be used as a variable or function name or somesuch. The print-name of the generated symbol will have *prefix* as a prefix, probably an incrementing numeric suffix, and potentially large quantities of gubbish in the middle.

Use this for creating variables for use in macro expansions, because the symbol can be typed in.

symbol-package *symbol*

Returns the contents of the package cell of *symbol*. This should be a package object, or nil.

8.4 The Value and Function Cells

See also chapter 3, page 8 for discussion about scope, extent, and binding, and chapter 3, page 8 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.

symeval *symbol*

Returns the current dynamic (special) binding of *symbol*.

set *symbol value*
Sets *symbol*'s dynamic (special) binding to be *value*.

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

boundp *symbol*
Returns t if *symbol* has a defined binding, 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 binding of *symbol*.

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

fsymeval *symbol*
Returns the current dynamic (special) function binding of *symbol*.

fset *symbol function*
Sets *symbol*'s dynamic (special) function binding to be *function*.

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

fmakunbound *symbol*
Analogous to makunbound.

8.5 Internal Routines

%symbol-cons *string*
Internal symbol conser. Creates a symbol with *string* as its print-name. (Note there is no mechanism provided for modifying the print-name of a symbol.)

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.

%symbol-print-name *symbol*
Returns the print name of the symbol *symbol*.

%symbol-package *symbol*
Returns the contents of the package cell of *symbol*.

%set-symbol-package *symbol new-value*
Modifies the contents of the package cell of *symbol* to be *new-value*, and returns *new-value*.

%symbol-property-list *symbol*
Returns the contents of the property list cell of *symbol*. This is, in fact, the property list itself.

%set-symbol-property-list *symbol new-value*
Modifies the contents of the *symbol*'s property list cell to be *new-value*, and returns *new-value*.

%symbol-link-cell *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 described in chapter 3, page 8.

9. Numbers

NIL provides several different representations for numbers. Presently, 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 only works with the basic arithmetic functions right now.

NIL integers are currently of two kinds. There are *fixnum*s and *bignum*s. *Fixnum*s 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 *bignum*s. *Bignum*s 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.

Floating-point numbers in NIL currently are of only one type, *double-float*s. A *double-float* utilizes the VAX double-precision floating-point representation, which takes up 64 bits; this provides 56 bits of precision in the fraction, and binary exponent from -127 to 127.

NIL also has provision for floating-point numbers of other formats, as described in section 2.1, page 3.

9.1 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.

9.2 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 56), which only work on fixnums, and max$ and min$ (page 60), which only work on double-floats.

9.3 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 will be provided by the COMMON-LISP **truncate** function (not yet in NIL).

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. *Previous release notes lied on this point.* **quotient** performs truncating division on integers; it does not produce a result of type **ratio** unless one of the arguments is a ratio.

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**.

lcm *integer* &rest *more-integers*
Returns the least common multiple of the integers.

9.4 Irrational and Transcendental Functions

Beware floating-point error.

9.4.1 Exponential and Logarithmic Functions

exp *number*
Returns *e* raised to the power *number*, where *e* is the base of the natural logarithms. This currently always works by coercing *number* to a double-float, and returning a double-float result.

expt *base-number power-number*
Returns *base-number* raised to the power *power-number*. If *base-number* is an 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. Not all types and type combinations are supported yet.

log *number* &optional *base*
Returns the logarithm of *number* in the base *base*, which defaults to *e*, the base of the natural logarithms. This currently uses only double-float arithmetic, so must coerce *number* and *base* to double-floats.

sqrt *number*
Returns the principal square root of *number*.

If *number* is a negative non-complex number, a complex number is returned.

The actual square-root computation is only performed using double-float arithmetic now, so the answer (or its real part if it is complex) is currently always a double-float (but see isqrt, below).

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.

9.4.2 Trigonometric and Related Functions

abs *number*
Returns the absolute value of the argument.

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.

sin *radians*
cos *radians*
tan *radians*
Standard trig functions. Will eventually accept complex arguments. The current versions only produce double-float results, and must coerce their arguments to double-floats.

asin *number*
acos *number*
asin returns the arcsine of the argument, and acos the arccosine. The result is in radians. These will eventually accept complex arguments. The current versions only use double-float arithmetic, however.

atan *y* &optional *x*
An arctangent is calculated and the result is returned in radians.

With two arguments y and x, neither argument may be complex. The result is the arctangent of the quantity y/x. The signs of y and x are used to derive quadrant information; moreover, x may be zero provided y is not zero. The value of tan is always between $-\pi$ (exclusive) and π (inclusive).

describe special cases

sinh *number*
cosh *number*
tanh *number*
Hyperbolic sine, cosine, and tangent.

asinh *number*
acosh *number*
atanh *number*
These aren't yet implemented.

9.5 Type Conversions and Component Extractions on Numbers

float *number*
Converts *number* to floating-point. This is currently always a double-float.

remainder *integer divisor*
Standard MACLISP remainder. Will become hairier in the future.

9.6 Logical Operations on Numbers

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

In MACLISP, and in the previous release of NIL, most of the functions presented here were fixnum-only. However, they now accept (and produce) arbitrary-precision integers. See also the section on fixnum-only arithmetic, section 9.9, page 56.

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 that these functions are not yet defined; for use on fixnums only, use logior&, logxor&, logand&, or logeqv& (page 57).

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 non-negative number of arguments.

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.

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)))
```

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.

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*

Not yet in??

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.

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 generic ldb and dpb functions.

Returns the high *count* bits of the binary representation of the absolute value of *integer*, or the low *-count* bits if *count* is negative.

9.7 Byte Manipulation Functions

There are various functions in NIL and COMMON LISP to deal with arbitrary-width contiguous fields of bits within integers.

Conceptually, most of these functions use objects called *byte specifiers* to describe such fields. The representation of a byte specifier, and the restrictions on the fields it is capable of describing, are implementation dependent. The function byte accepts two integers representing the *size* and *position* of the byte, and returns a byte specifier. Such a specifier designates a byte whose width is *size*, and whose right-hand bit has weight (expt 2 *position*), in the terminology of integers used as logical bit vectors.

byte *size position*

Returns a byte-specifier.

byte-size *bytespec*
byte-position *bytespec*

Returns the size/position of the bytespec.

Historically, the functions ldb and dpb, although first implemented in Lisp Machine Lisp, are derived from the PDP-10 instruction set. In that context, a four-digit octal number is used as a bytespec; it is typically referred to as *ppss*, because, in octal, the byte position is *pp*, and the byte size is *ss*. The ldb, dpb, mask-field, and deposit-field functions were originally implemented with such a bytespec, usually typed in by the user in octal. Both MACLISP and NIL originally implemented ldb and dpb in this way, such that they only operated on fixnums. However, the COMMON LISP definitions must work on all integers. Additionally, it was decided that the range restrictions of the *ppss* bytespec were too limiting. For that reason, NIL has incompatibly changed the bytespec format previously used. Functions are provided which utilize both the new and the old bytespecs.

ldb *bytespec integer*

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

A fixnum-only version which utilizes the new NIL bytespec format is available as **ldb&**, page 58.

The original fixnum-only version, which utilizes the obsolete *ppss* bytespec format, is available as %ldb, page 59.

There is also load-byte (page 59), which takes the byte position and size separately, and operates only on fixnums.

ldb-test *bytespec integer*

```
(not (zerop (ldb bytespec integer)))
```

mask-field *bytespec integer*

??Not yet in??

dpb *newbyte bytespec integer*

Returns a number which is the same as *integer* except that the bits specified by *bytespec* are replaced by *newbyte*, which is interpreted as twos-complement, and truncated if necessary.

For limited fixnum-only applications using new the bytespec format, see dpb& and set-ldb&, page 58. For utilizing the obsolete *ppss* bytespec format, see %dpb, page 59.

There is also the fixnum-only deposit-byte (page 59), which takes the position and size separately, and operates only on fixnums.

deposit-field *newbyte bytespec integer*

??not yet in??

9.8 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, *modulus* must be either an integer or floating-point number. The answer returned is a number of the same type (an integer or the same floating-point format), 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. As an efficiency note, this means that bignum arithmetic might be performed to produce a result for a fixnum *modulus* in order to manipulate the intermediate quantity 10 bits longer.

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*).

9.9 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.

It is true, however, that declarations would not solve all problems. Consider, for example, the following:

```
(let ((x compute) (y compute) (z compute))
  (declare (fixnum x y z))
  (let ((n (plus x y z)))
    (declare (fixnum n))
    ...)
```

In order to be totally correct, the compiler could not inline-code that call to **plus** as fixnum-only arithmetic without any checking (at least not in any simple way), because an intermediate result could overflow. For **plus**, this can only be done when there are two (or fewer) arguments. For the fully generic COMMON LISP /, the result could be a rational number! So, NIL provides a full complement of fixnum-only routines to parallel the generic routines which are normally used. In some cases, the routines are hardly worth having as type-specific versions because they are not inline-coded (for whatever reason), but are provided for completeness; mostly, however, they will be compiled with no checking whatsoever.

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.

9.9.1 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**.

9.9.2 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. Note that this is truncating division.

**** *fixnum1 fixnum2*
See below:

\& *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.

9.9.3 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.

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).

logcount& *fixnum*
Not yet in?

haulong& *fixnum*
Not inline-coded, but provided for completeness (see, perhaps, %fixnum-haulong, page 59). Maybe this should be dyked, 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).

9.9.4 The Super-Primitives

Getting closer still to the hardware...

%fixnum-haulong *fixnum*
NIL uses this routine to tell it how to do the haulong computation on fixnums. That is, it exists to be open-compiled by such functions as haulong and in one or two other critical places. The expansion is not nice to look at. This routine is mostly superceded by some other special compiler primitives which are not described here anyway.

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* *Special Form*
overflow-code...
sys:%fixnum-difference-with-overflow-trapping *x y* *Special Form*
overflow-code...
sys:%fixnum-times-with-overflow-trapping *x y* *Special Form*
overflow-code...
sys:%fixnum-ash-with-overflow-trapping *x y* *Special Form*
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.

9.10 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 at that time.

+$ &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*

10. 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 is not used for anything by NIL right now, but the information can be there if anyone wants to make use of it.

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 10.6, page 65.

char-code-limit *Variable*
char-font-limit *Variable*
char-bits-limit *Variable*

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.

10.1 Predicates on Characters

standard-charp *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*. *For some reason,* Backspace, Tab, *and* Form *seem to be in here and shouldn't.*

graphic-charp *character*

Returns t if *character* is a graphic (printing) character; that is, it has a single-position glyph. Although this could in principle be true for most characters without bits attributes, it is currently only true for those which are guaranteed to be so in standard ASCII.

alpha-charp *character*
uppercasep *character*
lowercasep *character*
bothcasep *character*
alphanumericp *character*

Predicates on character objects. All are nil for characters with any *bits*, and ignore the

font.

char= &rest *characters*
char< &rest *characters*
char<= &rest *characters*
char> &rest *characters*
char>= &rest *characters*

All of these routines require two or more arguments, which must be character objects. They perform comparison strictly on the character as given; that is, case, font, and bits matter. When more than two arguments are given, the routines return t if the comparison succeeds for all consequetive pairs of characters. All of these routines get open-compiled.

Note: COMMON LISP only defines these as taking two arguments?

char-equal *char1 char2*
char-lessp *char1 char2*
char-greaterp *char1 char2*

char1 and *char2* must be character objects. These behave like char=, char<, and char>, ignoring font, bits, and case.

characterp *frobozz*

Returns t if *frobozz* is of type character, nil otherwise. Because character is a primitive VAX NIL data type, this routine is efficiently coded by the compiler.

10.2 Character Construction and Selection

character *frobozz*

Coerces *frobozz* to a character. It may be already a character, a fixnum, in which case char-int is applied, or a string or symbol of length 1, in which case that single character is used.

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.

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))
```

10.3 Character Conversions

char-upcase *char*
char-downcase *char*
Upper- or lower-casify *char*, preserving the font and bits attributes of it.

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*
int-char returns a character object *c* such that (char-int *c*) is equal to *integer*, if that is possible; 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-charp) 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.

Character name symbols are symbols in the keyword package.

digit-charp
digit-weight
digit-char
Random things are happening to these, avoid them at present. See some of the routines described in section 10.5, page 64.

10.4 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, whcih 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.

10.5 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.

%valid-digit-radixp *radix*
This defines the valid range of radices which %digit-char-in-radix operates on. The radix must be a fixnum.

%digit-char-in-radixp *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-radixp, 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,

```
(char-equal c1 c2)
```

is the same as

```
(char= (%char-upcase-code (char-code c1))
       (%char-upcase-code (char-code c2)))
```

10.6 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. 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). The data in multidimensional arrays is always stored in row-major order; this is compatible with MACLISP, although normally it does not matter.

11.1 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 11.2, page 68.

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 11.3, page 69.

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 11.5, page 71.

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 11.4, page 71.

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).

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.

11.2 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 68).

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 cause the references to open-compile.

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). Many NIL complex datastructures, including the current implementation of bignums and more complex arrays which hold bits and small bytes, are built from simple bit vectors. Because of their utility, bit arrays are discussed further in section 11.8, page 73.

string-char
The array can only hold characters which satisfy the predicate string-charp (page 15). 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 more general strings. Chapter 12 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 12).

(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 (currently called **simple-general-vector**, although that will probably be shortened to just **simple-vector**). Such a vector is especially efficient, and may be accessed specially (see **sgvref**, page 72).

11.3 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 37), 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, and are protected by the garbage-collector; 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.

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**.

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 11.5, page 71.

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.

The *vector* and *object* arguments to vector-push and vector-push-extend are expected to be reversed by COMMON LISP, in order that they not be different from those to push (page 29). Ah well.

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.

At some point in the future, the function fill-pointer will be introduced. It will take one argument, a vector with a fill pointer, and return the value of the fill pointer. At that time, use of reset-fill-pointer will be superceded by setf of fill-pointer.

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.

11.4 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 38); 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.

11.5 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 11.3, page 69). 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.

11.6 Special Vector Primitives

There are several vector primitives in NIL which are around for mostly historical reasons, being superceded by more generic array primitives. It is *not* the case that these routines are any more efficient than the COMMON LISP-defined generic routines, as all of the routines in this section work on all sorts of vectors so differ trivially if not at all.

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 the internal vector-referencing subroutine.

vector-length *vector*

The only difference between this and the generic sequence length function is that length also accepts lists. The efficiency difference between the two is trivial, if at all measurable.

About the only justification for using vector-length is that it is more restrictive about the type of its argument than length is (both perform runtime type checking since they have to dispatch).

11.7 Simple Vectors

Simple vectors of element-type t, the primitive data-type simple-general-vector, are 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.

sgvref *simple-vector index*

References a simple general 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.

sgvref may be used with setf.

Because the term simple-general-vector may be shortened to simple-vector, sgvref may be renamed svref. The name sgvref will be maintained indefinitely for upwards-compatibility if this happens, however.

simple-general-vector-length *vector*
Returns the length of the simple vector *vector*. I suppose this could be renamed to sgvlength (svlength) in the future, by analogy to sgvref.

11.8 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. Bit arrays, of course, may be multi-dimensional. See also section 9.6, page 51.

bit *bit-array* &rest *subscripts*
Just like aref (page 67), but only works on arrays of element-type bit.

setf may be used with bit.

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.

11.8.1 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), 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-bits *size* &optional *initial-element*
Creates a simple bit vector *size* long, initialized with *initial-element*, which must be either 0 or 1.

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.

12. Strings

Strings are vectors of characters which satisfy the predicate string-charp (page 15). 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 16), 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-charp). 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.

The COMMON LISP string functions will all take keyworded arguments. Most of the string functions defined do not do so yet. When the change is made, they will be arranged to figure out how they were called, and behave accordingly.

12.1 String Coercion

string *frobozz*

This routine coerces *frobozz* to a string. If it is a string, that string is returned. If it is a symbol, the print-name is returned. If it is a character object, a string one-character long containing that character is returned; this string will probably *not* be freshly created. If *frobozz* is a fixnum, then the result is as if (string (character *frobozz*)) were performed.

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. When nil is really the symbol nil in a future release of NIL, this routine will probably be quite confused, and will be the wrong thing for code to use to do symbol -> string coercion.

12.2 String Comparison

None of the string functions are firmly defined with Common Lisp at this time. The set of functions provided is a subset of those defined long long ago in the NIL design process, and those found useful in the Lisp Machine implementation. Those which are defined by Lisp Machine Lisp will in general be generic (that is, they will coerce their "string" argument(s) to strings using string). There are also routines which do not do coercion, in order that they might be able to be open-compiled if possible.

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. Not specifying a count will cause it to default to a value such that the substring will continue to the end of the string. An explicit specification of nil for a count is not allowed in this circumstance: the fact that this may work in some cases is purely fortuitous. The original design intent had been to be able to determine at compile time how the *count* must be computed. Routines using this convention sometimes name these arguments *skip* and *take* rather than *start* and *count*.

string-equal *s1 s2* &optional (*start1 0*) (*start2 0*) *end1 end2*
: This is the basic routine for performing case-independent string-equality checks. The substring of *s1* from *start1* up to but not including *end1* (which defaults to the length of *s1* if nil or not supplied) is compared to the substring of *s2* from *start2* up to but not including *end2* (defaulted similarly). *s1* and *s2* are coerced to strings using string. The character comparison is that defined by char-equal.

string-lessp *s1 s2* &optional (*start1 0*) (*start2 0*) *end1 end2*
: Similar to string-equal; the comparison is that defined by char-greaterp, with a shorter string "less than" a string it is a prefix of.

12.3 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.

char-n *string index*
Essentially, (char-int (char *string index*)). This routine should probably not be used. For clarity and transportability it is better to use char-int or char-code on the result of char, or, if appropriate, one of the functions like char< (page 62) or char-lessp (page 62).

rplachar *string index new-character*
This replaces the *index*th character of the string *string* with the character object *new-character*. It returns *new-character*.

The preferred (and COMMON LISP-defined) way to update a character of a string is to use setf with *char*.

rplachar-n *string index new-character-int*
This replaces the *index*th character of the string *string* with the character which the fixnum *new-character-int* is the integer representation of. *new-character-int* is returned.

12.3.1 Low-Level Access

+internal-char-n *string index*
char-n on the primitive string *string*. Open-compiled always.

+internal-rplachar-n *string index new-character*
Similarly, rplachar-n, always open-compiled.

12.4 String Creation

make-string *length* &key :initial-value :initial-contents
Makes a string *length* long. Since no complicated array options may be specified, this will always be a simple-string. If *initial-contents* is specified, that should be a sequence which is used as the initial contents of the string; if it is not as long as the newly-created string, then the last element will be used repeatedly. If *initial-value* is specified, it should be a character; the string created is filled with that character. If neither is specified, then the initial contents of the string is undefined by COMMON LISP; in NIL, it will be filled with the character whose code is 0, #\null. As with make-array, it is an error for both *initial-value* and *initial-contents* to be specified.

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 37). 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-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 37), which will not accept symbols, but *will* accept sequences (of characters) other than strings.

string-replace *string1 string2* &optional (*index1 0*) (*index2 0*) *count*
Replace a substring of *string1* with a substring of *string2*. If *count* is not specified, then it is chosen such that neither substring referred to exceeds the bounds of the string.

This is an old NIL primitive which is obsoleted by the generic sequence function replace (page 38).

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 range specified is the entire string, *string* itself is returned; transportable code should not depend on this however, but rather use copy-seq (page 38).

string-upcase *string* &key *start end*
string-downcase *string*
Returns a copy of *string* with all characters converted as by char-upcase (or char-downcase. In NIL, the result is currently always a copy, but transportable code should not depend on this.

12.5 More String Functions

string-reverse *string*

string-nreverse *string*
string must be 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 62).

string-left-trim *char-set string*
string-right-trim *char-set string*
string-trim *char-set string*
char-set is interpreted as by string-search-set, page 78. What happens if no characters are trimmed? The result is as defined by substring, q.v.

string-search *key string* &optional (*from 0*) *to*
string-reverse-search *key string* &optional *from* (*to 0*)

12.6 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.

set-string-length *string length*
Sets the length of *string* (which *must* be a string) to be *length*. *length* must not be greater than the starting length of the string. This should have a different name, because it is *not* always going to be matched with string-length. This will only work on simple strings.

%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 is the primitive string-replace. It transfers *count* characters from the string *source* to the 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 character to store. 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 primitive string.

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.

***: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.

13. 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.

13.1 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*.

13.1.1 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?

13.2 Hash Predicates

sxhash *object*

This is the general LISP hashing function, based on the predicate equal (page 16). 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.

sys:sxhash-combine {*hash*}+ *Macro*

This macro might be useful to writers of :sxhash methods. 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...)

13.3 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.

14. Packages

Sketchy. This is all probably 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 84). 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. Defstruct

15.1 Introduction

This chapter is a modification of the description of **defstruct** appearing in the Maclisp Extensions Manual [3]. The primary modification has been the deletion of those things which (1) are not applicable to NIL, or (2) do not yet work in NIL. For this reason, 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.

15.2 A Simple Example

defstruct *Macro*

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: **color**, **size**, **name**, **make-elephant** and **alter-elephant**. Note that there were no symbols **make-elephant** or **alter-elephant** in the original form, they were created by **defstruct**. The definitions of **color**, **size** and **name** are easy, they expand as follows:

```
(color x) ==> (car x)
(size x)  ==> (cadr x)
(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. 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 15.4.3, page 89.

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 5.7, page 28).

```
(setf (name x) 'bill)   ==> (rplaca (cddr 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.

15.3 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

9.7, page 53), so is discouraged for the present.

15.4 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.

15.4.1 :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 (vectors in NIL, hunks in PDP-10 MACLISP, arrays on Lisp Machines and lists on Multics). It is possible for the user to teach defstruct new ways to implement structures, the interested reader is referred to section 15.6, page 98, for more information. Many useful types have already been defined for the user. A table of these "built in" types follows: (On the Lisp Machine and in NIL all defstruct type names are interned on the keyword package.)

:list

Uses a list. This is the default on 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 15.4.4, page 91.)

: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 15.4.9, page 93) 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 15.4.9, page 93) 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; until the new array scheme is implemented, however, NIL users are advised to use :vector instead.

: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 15.4.12, page 94) and :sfa-name (section 15.4.13, page 95) options. (SFAs are available in NIL for compatibility with PDP-10 MACLISP.)

:vector

Uses an vector. This is the default type in NIL, although in the future the default will produce a typed object (the :extend defstruct type).

: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. It uses the NIL flavor system to define the structure. The effect of this is that the structure will have a type name of the name symbol, which will work with typep, and can otherwise be treated as a flavor defined with defflavor, except that the existence of instance variables is not defined: one may use defmethod to define methods, and access the slots with the defstruct-defined accessor macros. The type defined by defstruct will inherit methods for printing, describe, and inspect.
See also the :class-symbol option (section 15.4.11, page 94).

15.4.2 :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 (not necessarily the name of an accessor macro; see the :conc-name option, section 15.4.8, page 92), or one of the special keywords allowed for the particular type of structure being constructed. 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 15.4.12, page 94). On

the Lisp Machine and in NIL all such constructor macro keywords (those that are *not* the names of slots) are interned on the keyword package.

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.

15.4.3 :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.)
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 (caddr 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 15.4.17, page 95).

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 15.6.3.2, page 100 for how accessors with more than one argument can come to be, there are no built-in defstruct types with this property.)

15.4.4 :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.

It is an error to use this option with an argument.

15.4.5 :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.

15.4.6 :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 120), 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.

15.4.7 :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.

15.4.8 :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. If the :conc-name option is not present, then no prefix is 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.

15.4.9 :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)
                   :conc-name)
  name
  age
  sex)

(defstruct (spaceman (:include person)
                     :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.

15.4.10 :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)))
```

15.4.11 :class-symbol

For use with the :extend defstruct type available only in NIL (section 15.4.1, page 88), 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.

15.4.12 :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.

15.4.13 :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.

15.4.14 :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.

15.4.15 :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.

15.4.16 :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.

15.4.17 :but-first

This option is best explained by example:

```
(defstruct (head (:type :list)
                 (: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.

15.4.18 :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.

15.4.19 :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.

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.

15.4.20 :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 15.5, page 97, for more information.

15.4.21 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.

15.4.22 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.

15.5 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 15.6, page 98). 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 15.4.20, page 96, and section 15.4.22, page 97).

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 15.6, page 98.

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.

15.6 Extensions to defstruct

defstruct-define-type *Macro*

The macro defstruct-define-type can be used to teach defstruct about new types it can use to implement structures.

15.6.1 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 15.5, page 97.) 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.

15.6.2 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.

15.6.3 Options to defstruct-define-type

This section is a catalog of all the options currently known about by defstruct-define-type.

15.6.3.1 :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 15.5, page 97) 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 15.6.3.6, page 102, and section 15.4.2, page 88.)

It is an error not to supply the :cons option to defstruct-define-type.

15.6.3.2 :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 15.5, page 97).

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 15.4.7, page 92). 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.

15.6.3.3 :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 15.4.5, page 91). 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)))))
```

15.6.3.4 :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 15.4.15, page 95, and section 15.4.14, page 95.)

15.6.3.5 :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).)

15.6.3.6 :keywords

The :keywords option to defstruct-define-type allows the user to define constructor keywords (section 15.4.2, page 88) 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 15.6.3.1, page 99).

15.6.3.7 :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 15.4.22, page 97), but soon it will check to be sure that each *option-i* is *only* used as an option with structures of this type.

15.6.3.8 :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.

15.6.3.9 :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 15.4.10, page 94). 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)))
```

15.6.3.10 :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*.

16. The Flavor Facility

16.1 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.

16.1.1 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.

16.1.2 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 (now subsumed by apply) 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*}* *Special Form*

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}** *Special Form*

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...* *Special Form*

flavor-name is the name of the flavor being defined. After it is defined, it is acceptable as a second argument to typep (page 14), 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 for the flavor. 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.)

Instead of an instance variable, one may specify a list of an instance variable and an initialization form. Each such form will be evaluated if necessary to determine the default value for that instance variable, at instance creation time. (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.)

If no initialization form is specified, the instance variable will be unbound. This will cause an unbound variable reference in the NIL interpreter. Compiled code (and external references) will pick up the unbound pointer and probably behave spastically; however, the unbound pointer will print showing the name of the variable, and the fact that it is a lexical instance variable.

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 105), in which case they refer to all of the *instance-variables* of the defflavor, or listed with those hey 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 alledgely 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...* *Special Form*
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...* *Special Form*
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.

16.2 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* *Message*
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* *Message*

The object should return t if it 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 16).

:sxhash *Message*

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 82, for the semantics which must be enforced, and note also the default :sxhash method, page 111.

:eval *Message*

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.

:funcall *argument-vector* *Message*

This is what happens by default when a funcall is performed on an instance. *argument-vector* is a stack vector (section 3.1, page 12) of the arguments.

:describe *?arguments?* *Message*

This is what is used by the describe function (page 145).

:exhibit-self *stream* *Message*
:select-nth *n* *Message*
:store-nth *n value* *Message*

These are used by the exhibit function (page 145) 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?* *Message*
:pp-anaphor-dispatch *Message*

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.

16.3 Message Defaults

Here are some of the messages provided by vanilla-flavor, and what they do.

:print-self *stream level slashify-p* *Operation On* vanilla-flavor
Prints something vaguely informative.

:get-handler-for *message-name* *Operation On* vanilla-flavor
Returns the handler function for the message *message-name*, or nil. In NIL, this is necessarily a subr.

:operation-handled-p *message-name* *Operation On* vanilla-flavor
Returns a non-null value if the object supports a message *message-name*, nil otherwise.

:send-if-handles *message* &rest *args* *Operation On* vanilla-flavor
If the object supports *message*, then it is sent that message with arguments of whatever *args* were passed; otherwise, nil is returned.

:which-operations *Operation On* vanilla-flavor
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 *Operation On* vanilla-flavor
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 *Operation On* vanilla-flavor
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* *Operation On* vanilla-flavor
:select-nth *n* *Operation On* vanilla-flavor
:store-nth *n value* *Operation On* vanilla-flavor
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?* *Operation On* vanilla-flavor
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 *Operation On* vanilla-flavor

The default :pp-anaphor-dispatch method does nothing, on the grounds that the :pp-dispatch method will not be printing any components.

17. 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.

17.1 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 the input.

trace-output *Variable*
This is the stream to which output from tracing (see the trace function, page 143) 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.

17.2 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 126).

: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 <not-yet-written>.

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 support 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...* *Special Form*
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...* *Special Form*

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...* *Special Form*

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.

17.3 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.)

17.3.1 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 15.3, page 86.

17.3.2 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.

17.4 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.

17.4.1 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 119). 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.

17.4.2 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.

17.5 Formatted Output

format *destination format-string* &rest *format-args*
format is used for producing simple formatted output; for instance, outputting a text string with things substituted in in particular formats. The documentation on format in the *Maclisp Extensions Manual* [3] describes the implementation of format in NIL, except that NIL is lacking the operations which deal with floating-point numbers.

format will be supported by COMMON LISP. There is one known and significant point of incompatibility: the ~G operator will mean something different; avoid its use if possible (~* is one alternative).

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-prin1 attempts to determine the existence of circular structure, and show this somehow without blowing up.

As of this release of NIL, this is now the pretty-printer documented in [7].

pretty-print *object* &optional *stream*
pretty-prin1, with a terpri first and output of a space character after. This, pretty-print is to pretty-prin1 as print is to prin1.

pretty-prin1-datum *object* &optional *stream*
Like pretty-prin1, but does *not* assume that *object* is LISP code.

pretty-print-datum *object* &optional *stream*
Similar.

17.6 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 117), 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*
Similarly.

17.7 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 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 use CHAOSNET. At the moment, only the local VMS filesystem is supported.

17.7.1 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 filesystem.

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.

17.7.1.1 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 124.

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 a local VMS filesystem, this is the RMS default device/directory string, which is what is modified by `set default` in DCL. If for some reason the string returned by RMS does not have a device specified, `SYS$DISK` will be supplied for the device component. 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.

17.7.1.2 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 124).

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 124).

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 124.

For example, the current NIL compile function creates a file named aaafoo.vas, and supplies the device and directory from *scratch-pathname-defaults*.

17.7.2 Opening Files

Files are opened with the open function (page 114), or with-open-file (page 115), and may be closed with close (page 115). 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, which is used to (among other things) drive host-name lookup.

17.7.3 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, or nil if this cannot be determined.

What does that mean if the file isn't there? (This is the common-lisp definition.)

The absolute value of the integer is almost complete garbage right now, however, two of them may be compared with greaterp or lessp or equal. This number is precise only to seconds, which is less than VMS provides.

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.

17.7.4 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...)

17.7.5 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.

17.7.6 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 15.3, page 86.

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* *Message*
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.

17.7.7 Internals for VMS Record Management Services

17.7.7.1 Data Structures

See the NIL sopurce files RMSSTR and RMSSUB; the file VMSFILE contains some examples.

si:make-fab

si:make-rab

si:make-nam

si:make-xab

si:fab *Resource*

si:rab *Resource*

si:nam *Resource*

si:xab *Resource*

17.7.7.2 RMS Hacking

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 string. This is a fairly direct interface to the SYS$SETDDIR system service.

If something goes wrong a system error status code may be returned instead of a string.

17.8 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 119) 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...

17.8.1 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* *Operation On* si:display-cursorpos-mixin
termcap-struct is what would be returned by si:make-termcap.

17.8.2 Making More Terminal Streams

As noted elsewhere (page 114), 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.)

17.8.3 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* *Operation On* si:display-cursorpos-mixin
This is what implements write-char to a display terminal, with all the interpretation of *char* described earlier.

:oustr *string start count* *Operation On* si:display-cursorpos-mixin
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* *Operation On* si:display-cursorpos-mixin

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* *Operation On* si:display-cursorpos-mixin

Analogous.

18. Syntax

18.1 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 [10] 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 print 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 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 the . character (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 fully documented in the *Maclisp Extensions Manual* [3].

, Comma is used for pe.forming 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 67). 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.

18.2 The Lisp Reader

18.2.1 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"))
```

18.2.2 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.

18.2.3 Readtable

readtable *Variable*
The value of this variable is a datastruction 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)
```

18.2.4 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)
  #$(v[0,0]:COS((A-B)*X)/(2-2*(A-B)^2)+COS(V[1,1]*X),
     v[0,1]:COS((A+B)*X)/(2-2*(A+B)^2)-V[1,0]*V[0,0],
     v[1,0]:V[1,1]*V[0,0],
     v[1,1]:V[0,1]*V[1,0])$)
```

19. Debugging and Metering

19.1 Flow of Control

19.1.1 Tracing

trace *function* *Macro*

Puts a trampoline in the function cell that causes printing of the arguments and the return value around a function call. *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.

19.1.2 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 :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.

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 14, page 83), 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 84); 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.

19.2 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.

19.3 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*) *Macro*

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.

***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.

19.4 Metering

19.4.1 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.

runtime

This returns the compute time of the process since process creation as a fixnum, in hundredths of a second (centiseconds). Note that this increment is incompatible with the MACLISP function of the same name; MACLISP runtime returns the runtime in microseconds—this would overflow into bignums in NIL, and also the data for NIL is only accurate to 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 current time using the Smithsonian time standard, but that could conceivably change. This quantity is only really accurate to hundredths of a second, even though it is potentially accurate to 100-nanosecond tics.

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 is likely to 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.

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.

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.

19.4.2 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^cl$call_meter+2(slp)
```

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))
```

19.5 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.

19.5.1 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 (of-type 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*)
```

19.5.2 "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.

19.5.3 Information in Modules

The exhibit function can be used to look at modules interactively.

19.5.4 Related Utilities

These are somtimes 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.

19.6 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.

20. 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.

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 (<not-yet-written>).

: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.

:new-constant-value *symbol old-value new-value*

This is somewhat similar to :symbolic-constant-update, but is signalled by (the primitive used by) defconstant (page 18) 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.

21. Compilation

Compilation is essentially the process of translating from one specification into another which is presumably 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 (as described in section 17.7.6, page 129). That done, it reads and processes each form in the file. What it does with each form depends on what the form is.

`(declare` *declarations*`)`
: The *declarations* are processed, and take effect for at least the remainder of the compilation. (NIL often makes the declarations globally right now; they might remain after the compilation, but usually they do 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 **(progn 'compile** *forms...*`)`.

`(compiler-let` *bindings forms...*`)`
: Establishes the bindings specified by *bindings*, then recursively processes *forms* in that environment. See page 156.

`(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.

`(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, inside of an 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 17.7.6, page 129).

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 125) 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]")
```

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 124) and *scratch-pathname-defaults* (page 126).

The above described calling sequence is the intersection of what is provided now, and what COMMON LISP defines for compile.

compiler-let ({(*var val*)}*) {*form*}* *Special Form*

When evaluated, this binds *dynamically* each *var* to the evaluation of each *val*, and then evaluates the *form*s in that environment. Syntactically, this is like let and let* (page 19). At this time, there is no guarantee that the variables are bound in parallel, or sequentially, however.

When compiled, however, the binding evaluation of the *val*s and binding of the *vars* is done at *compile* time, and then the *form*s (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. compiler-let is still useful, of course, for communication between macros.

21.1 Summary of Compiler Flags

21.1.1 Compilation Control

compiler:*open-compile-carcdr-switch *Variable*

If this is nil, which it is by default, then car, cdr, and all their compositions and update functions, are compiled as quick subroutine calls into the NIL kernel, which will perform type checking. Otherwise, they will be totally open-compiled by the compiler. Having the type checking on by default is a great aid in debugging. What might happen otherwise is that the program will attempt to reference non-existent memory. Although this also gives a lisp error, it is not very helpful as far as finding out why the error occurred. Then, one would have to recompile the program and start over to reenable error checking. Because of this, it is recommended that this switch be left nil by default, and only enabled for critical functions or loops which do lots of iterating down lists, and which are carefully checking types along the way. For instance, the built-in function copy-tree (page 32) is defined as

```
(defun copy-tree (tree)
  (compiler-let ((compiler:*open-compile-carcdr-switch t))
    (if (atom tree) tree
        (do ((tree tree (cdr tree))
             (l () (cons (copy-tree (car tree)) l)))
            ((atom tree) (nreconc l tree))))))
```

compiler:*open-compile-xref-switch *Variable*

This switch controls whether the si:xref (page 207) function is totally open-compiled. When this switch is nil, as it is by default, si:xref is compiled as a quick subroutine call into the NIL kernel, which will perform type and bounds checking on the arguments. si:xref is the primitive which is used for referencing defstruct-defined structures, and by the macros which implement outside-accessible-instance-variables for flavors. (Instance variable references inside of methods are always open-compiled.)

21.1.2 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.

22. Introduction to the STEVE editor

22.1 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.

22.2 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.

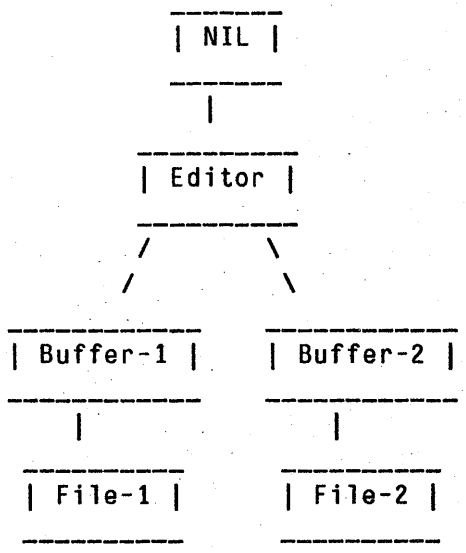
Now that you know how to exit the editor you may be curious how to enter it. Of course this is not an editor command, but rather a NIL function.

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.

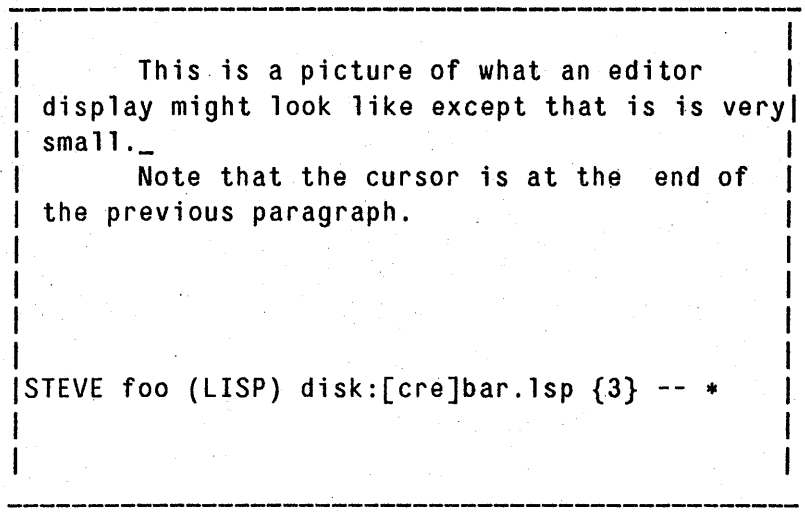
22.3 Editing Files

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.

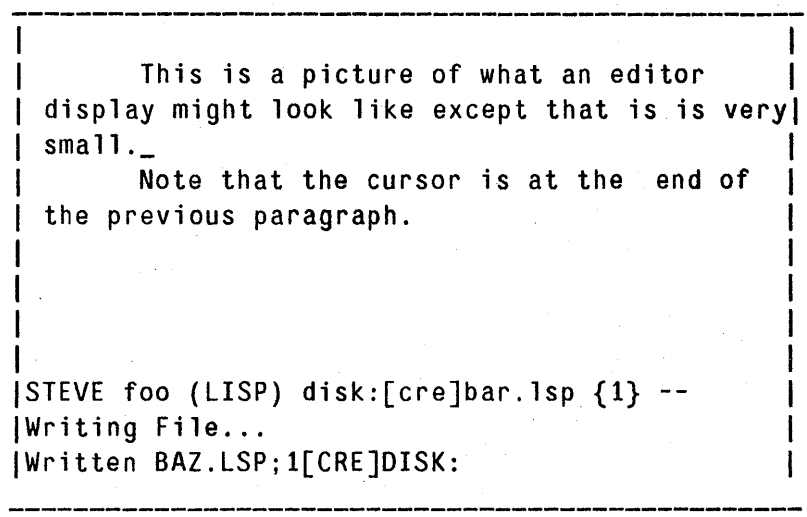
As an example, suppose that the screen looks like the diagram above. If you type *Control-X Control-W* (Write-File) the editor will prompt you for a file name. Assume you want to save the file into `disk:[cre]baz.lsp`. You type `"disk:[cre]baz.lsp"`. The screen will look like this:

```
|                                                  |
|      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.)



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.

22.4 Modifying the buffer

22.4.1 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.

22.4.2 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.

22.4.2.1 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.

22.4.2.2 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.

22.4.2.3 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*.

22.4.2.4 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.

22.4.2.5 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.)

22.4.2.6 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.

22.4.2.7 *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).

22.4.2.8 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.

22.5 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.)

22.6 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.)

22.7 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

22.7.1 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.

22.7.2 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.

22.7.3 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.

22.7.4 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.

22.7.5 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.

22.7.6 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 LL 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 LL 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.

22.8 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.

22.8.1 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* *Special Form*

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* *Macro*

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.

22.8.2 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* *Macro*
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* *Macro*
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* *Operation On* bp
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* *Operation On* bp
Place the bp pointing to the *n*th character of *line*.

:get-char *Operation On* bp
Return the character that the bp points to.

:get-char-forward *Operation On* bp
Return the character that the bp points to and advance over it.

:peek-char-backward *Operation On* bp
Return the character before the one that the bp points to.

:get-char-backward *Operation On* bp
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.

22.8.3 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* *Special Form*
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? *Macro*
Use the form (steve:argument?) to determine if any numeric argument was given.

steve:c-u-only? *Macro*
Returns t if the argument was *Control-U* with no number.

steve:real-arg-sup? *Macro*

```
(and (steve:argument?) (not (steve:c-u-only?)))
```

But more efficient in code and runtime.

steve:buffer-begin? &optional *bp* *Macro*
Test whether *bp* (or the current cursor) is at the very beginning of the buffer.

steve:buffer-end? &optional *bp* *Macro*
Similar; test for the end of the buffer.

steve:first-line? &optional *bp* *Macro*
Returns t if the *bp* is anywhere in the first line of its buffer.

steve:last-line? &optional *bp* *Macro*
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)
```

23. 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 [10]. 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 23.4, page 194, is a description of a more common usage of the patch system, where it is used for a system which is *not* dumped out.

23.1 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.

23.2 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.

23.3 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 194). 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.

23.4 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 155) 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.

24. Talking to NIL

24.1 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 124.) Then it enters its standard read-eval-print loop.

24.2 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.)

24.3 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.

24.4 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.

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.

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.

24.5 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.

24.6 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, 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 contorl-C you typed at the `display` program half an hour ago.

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.

25. 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.

25.1 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 simplifier only really works on simple predicates and connectives. It performs some basic 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.

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.

25.2 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.

25.3 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.

26. NIL Extended Data-Types

This chapter describes the implementation of the NIL extended data type (or just *extend*) facility. This is the stuff from which flavors and closures are built. It is not necessary to read this chapter to make normal and efficient use of flavors or structures; rather, this is presented for the curious, and those who may need such information to interface flavors or structures to non-LISP code. It makes no attempt to fully explain the implementation of flavors and type inheritance using this.

It is not clear why I am documenting this here.

26.1 The Extend Structure

NIL has about 20 primitive data types, which are differentiated by the type bits in the pointer. One of these types is named si:extend. When typep encounters this type, rather than just returning its name, it looks further. Implementationally, an extend is structured similar to a simple general vector; it is effectively just such a vector, *except* that it has an extra header slot in which the *flavor object* is stored. (This is what used to be called the *class* of the object.) This is a structured object (itself an instance of a flavor), which describes the type. Different extends of identical types will have the same (eq) flavor objects.

flavor-of *object*

This is the primitive which finds the flavor object of *any* LISP object. If *object* is an extend, then the flavor object of the extend is returned. Otherwise, the pointer type of the object (with a little fudging and hedging) is used to find the flavor object which is associated with that primitive type. Thus, this primitive integrates the primitive types with the extended types. It is what is used by typep of one argument (of-type), and even by send.

si:xref *extend index*

Returns the *index*th element of *extend*, which *must* be an extend. This primitive is normally compiled as a call to a NIL kernel subroutine which will perform type and bounds checking (but see compiler:*open-compile-xref-switch, page 157).

si:xref is usable with setf.

si:%xref *extend index*

This is an alternate name for si:xref, which is always inline-coded by the compiler without error checking. It also is usable with setf.

si:make-extend *size flavor-object* &optional *initialization*

This is the primitive creation functions for extends. An extend of size *size* is created, with a flavor-object of *flavor-object*. If *initialization* is nil or not specified, then the slots of the created extend are initialized to nil. Otherwise, *initialization* must be either a simple general vector, or another extend, which is *at least size* long. The created extend will have its slots copied from *initialization*. make-instance uses this to initialize constant slots from a template extend, as it is substantially faster than initializing them individually.

Because this is a low-level implementation primitive, no error checking is performed anywhere.

26.2 The Flavor Object

The flavor object is a structure which describes the type of the object. Certain of the components are defined to occur at specific offsets, for the benefit of the NIL kernel. Code written in MACRO32 inserts the file `nil$disk:[nil.vm]:clsoff.mix` to define these offsets (which have prefix `cls$`). Some of the comments and entries in that file are out-of-date, however the slots named there are a superset of those which are actually used by the kernel.

The flavor object is a structure of type si:flavor. The definition of it is in the file `nil$disk:[nil.src]flavm.lsp`, and is somewhat gross and hairy; this is a result of historical and bootstrap reasons (it used to be defined without using defstruct). The accessors which fetch components of this structure have names such as si:flavor-type-bits. Some of those of interest in the kernel are:

si:flavor-name
: The name of this type.

si:flavor-types
: This is a list of all type names from which this type inherits. Two-argument typep, when the type-specifier is the name of a flavor-defined type, looks to see if that type specifier is a member of the flavor-types of the flavor object of the object. That is,

```
(typep x 'foo)
```

is (when foo is the name of a flavor-type) done by

```
(memq 'foo (si:flavor-types (flavor-of x)))
```

The first element of this list is *always* the name of the type.

si:flavor-type-bits
: For efficiency reasons, certain non-primitive types are given bit assignments. This slot holds a fixnum which has a bit set for each of those types which this type inherits from. The types which have such distinction are some of the numeric and array types. The official assignments of these bits are in the file `nil$disk:[nil.src]flavsetup.lsp`. The position assignments for MACRO32 code (suitable for use with bbc, for instance) are in the file `nil$disk:[nil.vm]flvtypes.mix`.

27. Foreign Language Interface

27.1 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.

27.2 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* *Special Form*

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 intcpreted. The example above is from code in the systems-internals package.

27.3 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.

27.4 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.

27.5 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.

28. 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.

28.1 What Broke Since Release 0

28.1.1 NIL, T, etc.

Since Release 0, the null object is now also considered to be a symbol. That is, the symbol nil and the object () are one and the same. Recompilation of code which uses symbolp, plist (and setplist), get-pname, and symbol-package is necessary. If code depended on the distinction between the two, it will have to be fixed. Note that, in particular, this change means (for instance) that the null object will work as a first argument to get now, and that the null object is ambiguous with other symbols (which could affect symbol-table hacking and interning, that kind of thing).

The special object representing boolean truth (#t) is gone. The symbol t takes its place. Compiled code which references the object #t does not actually have to be recompiled, because such references will magically turn into the symbol t when the files are loaded. Of course if the code had been read into the COMMON LISP readtable, it should not contain #t, and there are probably numerous other reasons why it should be recompiled in the new release anyway. Any code which depends on the object representing boolean truth not being a symbol must be fixed.

Implementationally, the null object is still the same, but has a pseudo-symbol associated with it which is used by intern and plist etc. instead. This pseudo-symbol, and the symbol t, have their values initialized by a mechanism identical to that used by defconstant. Code which is not recompiled in the new release may be able to change the values of these and other constants.

28.1.2 Common Lisp Arrays

As described in the notes on Release 0, most of COMMON LISP arrays have been implemented. This modifies the NIL type hierarchy, and thus the semantics of some of the accessors and predicates. The incompatibility which arose most commonly within the NIL sources was that vectorp is now true of all one-dimensional arrays, i.e., strings and bit-vectors and other types of "arrays". (In release 0, strings, bit-vectors, "general" vectors, and "arrays", were all disjoint types.) As a result, successive type tests done with cond or typecase may have to be reordered. Also, vref works on all one-dimensional arrays, which may mask an incorrect typecheck order.

28.1.3 Generic Arithmetic and New Numeric Types

As noted in the Release 0 notes, the basic "single-character" arithmetic function names now refer to generic arithmetic functions. These functions do not have a high penalty for use, however, as they are mostly implemented as subroutine calls into the NIL kernel, which handle the common fixnum and double-float cases, including coercion.

The types **complex** and **ratio** have been added. The primary incompatibilities are that the "basic" division function **/** now will return a ratio rather than performing truncating integer division. Although the complex arithmetic is not complete, a complex number may now pop up on one when doing something like **sqrt** of a negative number.

See chapter 9, page 46, on numbers, for complete information.

28.2 Future Changes

Some of these are the same as those anticipated as of Release 0.

28.2.1 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.

28.2.2 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 18) 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.

28.2.3 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.

28.2.4 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).

28.2.5 New Package Facility

The COMMON LISP package definition is practically finalized now; implementation will start once it is guaranteed stable. 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.

28.2.6 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. Unfortunately, this (as a modification of an earlier COMMON LISP specification) only came about a matter of days before the NIL release is expected to be ready, in fact, *after* the announcement of this release. To ease the change, the future vector-push and vector-push-extend will, when they see first or second arguments of the wrong type, see if they would be correct when reversed, not that this will always work.

28.2.7 Miscellaneous Other Things

The functions subst-if, subst-if-not, and some similar others (which are not documented here anyway) may have their argument order changed to be consistent with some other functions (which are not implemented here).

All places in the pathname code which supply a device of SYS$DISK will be changed to supply the one-level translation of that logical name. For instance, user-workingdir-pathname currently might return something like "SYS$DISK:[GSB]"; it should in fact be returning something like "USRD$:[GSB]".

References

1. Steele, G. L., *Common Lisp Reference Manual*, Carnegie-Mellon University Department of Computer Science Spice Project, (in preparation).

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).

6. 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

7. Hawkinson, L. B., and Burke, G. S. Unwritten memo/documentation on the pretty-printer noted in section 17.5, page 120.

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. Weinreb, D., and Moon, D., *Lisp Machine Manual*, MIT Artificial Intelligence Laboratory, Cambridge, Mass., (July 1981).

11. Pratt, Vaughan R., *CGOL - an Alternative External Representation for Lisp users*, AI Working Paper 121, (March 1976)

Concept Index

Message Index

Resource Index

Variable Index

Function Index