Introduction to NIL

March 1983

Glenn Burke

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 is a primer for NIL, a New Implementation of Lisp. It is not a description or overview of the language, but rather documentation of some of the "ordinary" facilities available, so that those with some small LISP experience can use NIL in a basic way.

Acknowledgments

Most of the NIL system development for the past year has been performed by myself and George Carrette, who is also the primary source of VMS expertise within Tech Square. The implementation of the NIL editor is solely the result of the efforts of Christopher Eliot, who is also responsible for much of the numerical (especially bignum) code now available in NIL. Inaccuracies in the over-simplified documentation of the editor here, however, are totally the fault of GSB.

The AI Lab Robotics Group deserves credit for just "being there" as potential NIL users. In particular, Patric Sobalvarro has been extremely helpful with both NIL implementation, and documentation writing and proofreading.

Peter Szolovits has been (and will continue to be) a sounding board for problems of all sorts, and additionally deserves credit, along with Michael Dertouzos, Al Vezza, and Joel Moses, for continuing support of the project.

The NIL language is being converted to conform to the (as yet not officially complete) COMMON LISP standard. This effort is the result of the work of a great many people (see [2]).

Note

This document should be considered an informal paper for internal use. It is not intended for formal distribution or reference in its current form.

Table of Contents

1. Introduction

NIL is a dialect of LISP which runs on DEC VAXes under the VMS operating system. It is destined to be compatible with COMMON LISP [2]. COMMON LISP is a sufficiently complete specification of Lisp that many programs should be able to be written conforming to that standard, and thus be able to be transported to other COMMON LISP implementations. At this time, the "final draft" of the COMMON LISP manual has not been published, and also certain fine points are not finalized. NIL, although not nearly complete, conforms closely to much of the existing and known COMMON LISP functionality.

The NIL system provides a lexical interpreter complete with closures, a compiler, a primitive interactive debugger, and (under development now) an editor (written in NIL). NIL is compatible with MACLISP to whatever extent is possible, given that the interpreter uses lexical scoping rules, and that certain fine incompatibility points between COMMON LISP and MACLISP exist. Certain of these will be described later.

This document is oriented towards someone who has some knowledge of LISP, such as LISP 1.5 or (even better) MACLISP. It is not in itself a LISP primer. It deliberately ignores or downplays many complicated and esoteric features. In other areas, the documentation is simply abbreviated to what are (hopefully) the more useful points. Neither should this document be considered to be an "overview" of NIL; there are many facilities which are totally omitted, yet which are integral parts of the NIL language. Rather, it is both a primer and reference source with which someone with some minor LISP experience should be able to derive what knowledge is needed to do a fair amount of programming using the NIL interpreter. The hordes of special-case functions which are used to produce efficient production systems, have all been overlooked here. The compiler has been overlooked also, partly for simplicity, and partly because it is not capable of handling all of the control and binding constructs which the interpreter does. Highly sophisticated LISP users, or those familiar with a large and complex system such as LISP MACHINE LISP, may still find some things here useful (although perhaps tedious reading) because of incompatibilities.

The NIL system is still incomplete and under active development. It does not yet have a garbage collector. It has a compiler which produces pretty good VAX code directly, and which is expected to be eventually replaced by an even better one which will handle both more complicated lexical control and binding constructs, and type declarations for better numeric and string crunching. Eventually, NIL will support at least four different floating-point formats to allow numeric crunching of various flavors, and more sophisticated handling of complex numbers than it does now. It is expected to be able to talk to the CHAOS network itself, allowing it to directly access files on other hosts the way a Lisp Machine does. The editor in NIL is still fairly new, having only recently been released for use within the MIT community.

2. Style, Conventions, Etc.

2.1 Documentation Conventions

The conventions used in this manual are essentially the same as those used in the Lisp Machine Manual [3].

In this document, all numbers are decimal. In this it differs from both the Lisp Machine Manual, and the Maclisp Reference Manual [6].

In lisp code examples, the symbol => should be read as "evaluates to", and ==> as "macroexpands to". The latter should usually be taken as a paraphrase (or functional equivalence) rather than a literal expansion, as the actual expansion will usually be something more obscure and implementation-dependent.

Function documentation typically looks like

sample-function *string* &optional (*stream standard-output*) &rest *other-things*
sample-function takes one or more arguments. The first should be a string. The second, if it is supplied, should be a stream; if it is not supplied, the value of the variable standard-output is used instead. It outputs the characters of *string* to *stream*, immediately followed by the number of *other-things* supplied, in hexidecimal. Thus, the call

```
(sample-function "foo" terminal-io 'a 'b 'c)
```

would produce the output

```
foo3
```

In function documentation, the lambda-list keywords such as &optional, &rest, and &key are used, similar to the way they are used in defun. The exact meaning of them is described in section 6.1, page 15.

Variable definition looks like

sample-variable *Variable*
The value of this variable is the number of times the NIL language has broken due to DEC field service modifying the VMS system parameters.

Special forms and macros are sometimes documented using lambda-list keywords like function documentation, but more often using a slightly different description of how they expect their forms:

sample-specform (*var form*) {*declaration*}* *forms...* *Special Form*
The special form sample-specform binds *var* to a stream which will broadcast its output to both the terminal and the stream which *form* evaluates to. The *declarations* are used just as with lambda or let. It evaluates *forms* in this environment. Thus,

```
(sample-specform (*stream* *disk-file*)
   (declare (special *stream*))
   (princ "The findings are:" *stream*)
   (display-findings))
```

prints "The findings are:" to both the terminal and the stream which is the value of *disk-file*. Presumably, display-findings uses the variable *stream* free, and prints the findings to that stream.

The format {*foo*}* means any number of *foo*s; {*foo*}+ means one or more *foo*s. [*foo*] means an optional *foo*, and *foo*... is the same as {*foo*}*.

2.2 Major Implementation Points

NIL, like LISP MACHINE LISP, utilizes a function cell in which to define functions and macros. It does not use the property list the way MACLISP does. Interpretation of whether a name is a function or a variable is done purely syntactically; in, for example,

```
(fact number)
```

fact is interpreted as a function, and number as a variable. Doing a setq of fact will not affect this interpretation, and neither will doing a defun of number.

NIL is one of the few "production" Lisps around now which utilize lexical scoping rules in the interpreter, as opposed to that only being the default interpretation when the code is compiled (as is the case in MACLISP and LISP MACHINE LISP). The NIL interpreter thus behaves somewhat like the compiler, and heeds variable declarations. This also has an impact on the efficiency of interpreted code, but no comparisons of its speed have been made with any comparable Lisp implementations. The lexical scoping (which is discussed in section 5.1, page 13) also affects the ways in which one must debug interpreted code. This is because lexically scoped variables may only be validly referenced from within the lexical (textual) scope where they were bound. As a result, it is difficult (if not impossible) to find their values from "outside" of the function they are bound in. The debugger (page 43) and exhibitor (page 45) are capable of compensating for this somewhat.

2.3 NIL and T

In NIL, the canonical objects for representing boolean true and false are atomic symbols, just as they are in MACLISP, LISP MACHINE LISP, and as specified by COMMON LISP. (This differs from earlier implementations of NIL including the one, made in December 1982, which is being distributed as Release 0.) One may not bind or setq these symbols. They do, however, have values and potentially functions associated with them, just as all other symbols do. The mechanism by which they have values which are not allowed to be modified is available to the user by use of the defconstant special form (page 17).

2.4 Syntax and Symbols

NIL by default "comes up" using the reader syntax specified by COMMON LISP. The primary difference of note involves the / and \ characters: the "quoting" or "slashification" character in COMMON LISP is backslash (\), rather than slash (/) as it is in MACLISP and LISP MACHINE LISP. Thus, the division function whose print name consists of the single character / may be typed in as just that single character, unlike (for instance) MACLISP where the slashes needed to be doubled; conversely, the function in NIL which performs the gcd operation on fixnums only (for MACLISP compatibility), and which has a print name consisting of two backslashes (\\), must be typed in as \\\\. In documentation, symbols are displayed without the (back)slashes which may be necessary for them to be typed in. Within lisp examples, lisp expressions are displayed with the necessary slashification, utilizing COMMON LISP syntax.

One other syntax difference is that NIL has a special data type to represent *characters*. Those who use the sharpsign (#) reader macro should note that in COMMON LISP syntax, there is no #/ syntax, only #\ which can accept both character names and single characters, and that the object represented is a character, not the fixnum representation of the character.

3. NIL and VMS

NIL runs as a separate process under VMS. In TOPS-20 terminology, it "keeps"; it does not disappear when exited the way many other programs do.

3.1 Getting In and Out of NIL

The `nil` command to the command language interpreter (CLI) runs a program which "manages" the NIL. Essentially, saying `nil` with no options resume an existing NIL if there is one, or create and start one if there is not. If one is "in" NIL, control-Y (the sort of general VMS "interrupt process" command) will suspend the execution of the NIL, and return to the command language interpreter.

There is some (probably timing) bug such that occasionally after a control-Y is typed, the NIL gets suspended but the command interpreter does not return and prompt. When this happens, the next line of input may "get lost". If the "$" prompt does not appear in reasonable time after typing of control-Y, then a couple more control-Ys and maybe a carriage-return or two will get it, and avoid having the next line of input get ignored. If this does not work, then something more serious is wrong.

The `nil` command may be given other arguments which are only meaningful if there is a NIL process around already:

`nil/kill`
: This kills the NIL. It is the same as executing **(quit)** from within the NIL.

`nil/proceed`
: Resumes the NIL process, but does not enable the NIL to perform terminal input and output. If the NIL attempts to do them, it will wait until it is resumed. Similarly, if any of the NIL functions which interact with the superior are called while the NIL is proceeded, it will wait until it is resumed.

3.2 Terminal Interrupts

VMS does not provide a very general mechanism for receiving interrupts by typing a character on the terminal. The only available option is to get an interrupt when control-C is typed. So, what NIL does is to handle control-C interrupts, and then prompt for input: you type a character to tell it what interrupt function to perform. Some of the more interesting characters are

control-G
: Quit to top-level Lisp. Aborts whatever was running.

B Enters a break loop (see **break**, page 44). When the break loop is exited, program execution resumes.

D Enters the Lisp debugger. The stack and program environment can then be examined. If the debugger is exited with the Q command, then program execution resumes as before.

? Lists the characters available and what they do.

N Ignore the interrupt and continue whatever was happening before.

X Also control-X but you probably can't type that (as explained below). Modeled after the MACLISP control-X interrupt, this quits somewhat less fatally than control-G: it aborts back to the most recently established **errset** (therefore **break** loop), or to top-level.

It should be noted that the control-C interrupt is handled by NIL at Lisp-level. If the NIL is busy doing something uninterruptibly, it will ignore the control-C until it is done. Typing multiple control-Cs will only make the NIL think that it is broken, and call up the VMS debugger, which is generally fatal to the NIL.

VMS does not provide a notion of "which job is using the terminal". Because of this, often a NIL which is not running will have its control-C interrupt triggered when control-C is typed at some other program. This generally does not *happen to a NIL which is running without-the-terminal (as by* `nil/proceed`*), however. All that can be done is to type N at the* `"INTERRUPT>"` *prompt when it occurs. This will be fixed when the interrupt system is revamped.*

3.3 Other Perversities

There are several other characters which VMS places special interpretations on.

xon/xoff

Control-S and control-Q block and continue output from VMS, respectively. This is allegedly for "flow control", even though it is impossible for it to work in all but the simplest cases. Supposedly your terminal sends control-S when it is getting too much output from the host, which then shuts up for a while (until control-Q is sent). More conveniently, it is something you can type to stop the output in its tracks, so you can read it; you can then type control-Q to get output to resume. NIL normally does not disable this, except when within the editor.

Control-T

This displays some information about what is running, similar to the TOPS-20 control-T. Unfortunately, VMS does not handle cursor control itself (forcing NIL to), and prints out this information causing NIL to get confused about where the cursor is. You will also note that VMS prefers not to clear lines it types on before it types there.

control-O

This aborts output. It causes output to the terminal to be discarded, until some input is performed (or certain other exceptional conditions occur like a control-C interrupt). NIL does not recognize this condition, and VMS again types somethign which confuses NIL about where the cursor is. If you wish to abort output, it is probably (depending on the context) better to use control-C and type "X" at the interrupt prompt; this will abort back to the most recently established **break** loop, or to top-level.

control-U

When standard buffered input is being done (as when you are typing at the command interpreter), control-U flushes the input buffer. The NIL input processor uses this to delete a line of input (not the whole expression).

control-R

When standard buffered input is being done (as when you are typing at the command interpreter), control-R reprompts and retypes the buffered input.

control-X

To the VMS buffered input reader, this means both control-U, and flush typeahead (the characters which have been typed but not read yet). In order that this functionality be implemented in a completely general way, the VMS terminal input driver is kind enough to translate this character into control-U. Thus, when you type control-X at NIL, it sees control-U.

VMS offers a way around all of the above randomness, known as *passall mode*. This mode disables *everything*, including both control-Y and control-C. It also apparently disables "broadcast messages", such as announcements that the system is about to go down. When the NIL editor is entered, it enters this mode. It leaves it temporarily at certain strategic points, but not too frequently. What this means is that if the editor gets into an infinite loop, you get totally wedged, and will probably need external help (with privileges) to even be able to log out. Maybe DEC can get it right next time.

4. Data Type Overview

4.1 Conses

The data type **list** is called **cons** in NIL, to distinguish it from the meaning of **list** in which **nil** is the empty list. Other than that, all is ordinary.

4.2 Symbols

Symbols serve as names for variables, functions, or simply as keywords. Every symbol has a *property list* or *plist*, a list of alternating property indicators and values. It has a *print name* or *pname*, which is a string that "is" the printed representation of the symbol. It can have both a value and a function associated with it. It also has a package (section 4.6.2, page 11).

The symbol **nil** is somewhat special. It has the data type **null** (a subtype of **symbol**), and is the only object of that type. Otherwise, it is treated the same.

4.3 Numbers

NIL provides representations for and operations on both rational and floating-point numbers. It also provides a complex number representation, but only the simpler arithmetic operations know how to deal with it right now. Similarly, many operations on non-integer rational numbers are deficient.

4.3.1 Rationals

An integer is a rational number whose denominator is 1. If an integer is within a particular range (is representable in twos-complement notation in 30 bits), then it is the special type **fixnum**, otherwise it is a **bignum**. Conversion between the two is automatically performed by the arithmetic functions.

Non-integer rational numbers are of the type **ratio**. The "top" and "bottom" of a ratio are its *numerator* and *denominator*, and may be accessed with the **numerator** and **denominator** functions, which work on integers also. (The numerator of an integer is itself, and its denominator is 1.) The numerator and denomimator of a ratio will always be returned in reduced form, and the denomimator will always be positive. Ratios are notated by typing the (possibly signed) numerator, a slash, and the (unsigned) denominator, as in **-4/3**.

Ratios are typically created by rational number division. This can occur when integer division is being performed using the COMMON LISP **/** function, and the dividend is not an exact multiple of the divisor. Thus,

```
(/ 8 6)    =>    4/3
```

To get truncation, the MACLISP compatible **quotient** function can be used; it will produce a ratio only if one of the inputs is a ratio.

4.3.2 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.

4.3.3 Contagion and Conversion

Most of the standard arithmetic functions operate on any/all numeric types, and convert from one to another as necessary. In general, rationals convert to floating-point, and non-complex to complex. Note that it is not significant to talk about interconversion of ratios, integers, and fixnums; an integer is just a special case of a rational number, as is a ratio.

Unlike with rationals, complex numbers with imaginary parts are never reduced back to non-complex numbers. Otherwise, the operations on the real and imaginary parts undergo contagious conversion in the same way.

For example:

```
(+ 2 3.0)        =>  5.0
(+ 2 3)          =>  5
(/ 2 3)          =>  2/3
(/ 2 3.0)        =>  0.6666666666666667
```

4.4 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).

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.

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).

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. The ASCII control characters read from files or the terminal are not mapped into characters with the control bit, except in special situations (like by the editor itself). They are left as-is, and in principle these characters would make up an extended graphic character set like LISP MACHINE LISP uses. These characters however do not have printed representations on standard ASCII terminals, so they have their own naming syntax: the character object for ASCII control-A, for instance, will display as #\^A, and may be typed in as such. Essentially, this "uparrow" syntax does a logical *xor* of the following character (code) with 100 octal.

4.5 Arrays

Most of the remaining types of interest in NIL are various sorts of *array*. Arrays in NIL may have any rank from 0 to (about) 250. The indexing is always zero-origined. Arrays specialize in various ways. One-dimensional arrays are also of type vector, and may be used (interchangeably with lists) as sequences with various sequence operations. Arrays also specialize according to what types of data may be stored in them. In particular, there are special array types which can only hold the fixnums 0 and 1, known as bit-arrays, and some which can only store objects of type string-char; one-dimensional arrays (i.e., vectors) with this element-type restriction are *strings*.

Vectors which can hold objects of any types, called *general vectors*, may be typed in by starting the sequence of objects with #(and ending it with). For instance,

```
#(foo (bar) #\A)
```

is a vector with three elements: the symbol foo, a list of the symbol bar, and the character uppercase a. Such a vector may be empty (have zero length):

```
#()
```

Strings are typed in and printed out by enclosing the characters of the string with doublequotes. If the string is to contain doublequote or backslash characters, they must be preceded by a backslash character. The syntax

```
"foo\\\""
```

is the printed representation for a string with five characters: f, o, o, backslash, and doublequote.

Bit vectors are displayed as the characters #* followed by a sequence of 1 and 0 characters. Thus

```
#*001110
```

is a bit-vector of length 6; its elements numbered zero to five are 0, 0, 1, 1, 1, and 0.

NIL arrays can support various hair dealing with indirection, sharing, overlaying of datastructures, and adjustable sizes. Some of these are incomplete or undebugged, others are in use, but none will be discussed further in this document.

4.6 Packages and Readtables

Details of these are not fit for this document. However, they need to be mentioned, because they will probably be encountered in some context or another.

4.6.1 Readtables

A *readtable* is the datastructure which defines the syntax used by read. The NIL environment contains a few for various purposes. Two in particular are of interest: these are the COMMON LISP readtable, and the NIL readtable. The default readtable used by NIL is the COMMON LISP readtable. The NIL readtable is the one utilizing the syntax designed for NIL several years ago, which is incompatible with COMMON LISP syntax. The irony here is that the NIL readtable is more compatible with the default MACLISP and LISP MACHINE LISP reader syntax than the COMMON LISP readtable is, due to the interchange of slash and backslash for character quoting within tokens. For this reason, certain otherwise-vanilla source files will sometimes explicitly specify the NIL readtable. There are other disparities having to do with the reading of characters as both fixnums and as character objects, but the point is to note why a specification of the readtable might be necessary.

4.6.2 Packages

In NIL, as in LISP MACHINE LISP, not all symbols share the same "name space". They are organized into *packages*, which are a structured organization of symbol tables. (This is like having multiple *obarrays* (of MACLISP) or *oblists* (of LISP 1.5), but there is more structure imposed on it.) Packages will not be documented here, other than what their existence may imply even when they are not used. Knowledge of *interning* (at least in the LISP 1.5 sense) is assumed.

The colon (:) character is reserved for specifying the package into which the following symbol is to be interned. In fact, the package system provides inheritance, so the symbol may actually be inherited from some "superior" package. There is a "root" package, called global, which the symbols which are for use by everyone are inherited, for instance. It contains the symbols which name system functions (like car), variables (like the constants most-positive-fixnum and pi), and some just used as symbols (like the type name double-float). The syntax si:internal-eval means "the symbol internal-eval read in (as if from) within the package named si". (si is the short name for system-internals, where most of the random innards of NIL reside.)

There is a special package which is referred to with an empty "package prefix"; this is used for keywords, those symbols which are tested for equality with eq and must remain so no matter which package they are read into. Many functions take arguments "by keyword" so that there is less confusion about which arguments go in which position. For convenience, these keywords self-evaluate (and are constants). Thus, one can type both

```
(fill frobozz something :start 4 :end 5)
```

or

```
(fill frobozz something :end 5 :start 4)
```

equivalently.

For simple applications, it is usually not necessary to interact with the package system other than to use colons to type keywords (as above), and to avoid them in other contexts. NIL starts up in the user package, which is itself empty but provides access to all the globally defined symbols. Restricted applications can usually function entirely from one package and not refer to any others. For instance, when MYCIN is loaded with (load-mycin), it creates the mycin package, loads up the MYCIN files into it, and makes that the current package. From then on, all interactions can be performed from there; all of the functions and variables of MYCIN are available.

5. Interpretation

5.1 Binding and Scoping

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 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 17). 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 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*.

6. Variables and Definitions

6.1 Lambda List Interpretation

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.

6.2 Defining Functions

defun *name lambda-list {declarations}* forms...* *Special Form*

defun is pretty much like a MACLISP or LISP MACHINE LISP defun. In NIL, in case you are interested, a defun in the interpreter globally defines the function *name* as a closure over the lexical environment in which the defun is performed. The *lambda-list* is a COMMON LISP compatible lambda-list, which is mostly the same as that used by LISP MACHINE LISP; the special keywords &optional, &rest, &aux, and &key are interpreted accordingly, as described above; defun simply associates the function name with a corresponding lambda-expression constructed from the lambda-list, declarations, and forms.

defmacro *name pattern forms...* *Macro*

defmacro is used for defining macros... It is described in the Maclisp Extensions Manual [5].

6.3 Defining Variables

Because of the lexical nature of the NIL interpreter, those variables which are going to be used "free" from functions, or which are going to be globally assigned values, are usually best "defined" at top-level. Global declaration that a variable with a particular name will be special avoids problems with subtleties of local declarations. For the same reason, it is conventional for special variables (which are not constants defined with defconstant) to begin and end with the character *. (Note that most NIL system variables do not obey this convention, mainly because they have not been converted yet.)

defvar *variable* [*init*] [*documentation*] *Macro*

defvar is used for "defining" variables. It should only be used "at top level" in a source file, not from within code. If there is no *init* form specified, then it simply globally declares *variable* to be special. If there *is* an *init* form specified, then, if *variable* does not have a value already, the *init* form will be evaluated and *variable* set to that. *documentation*, if present, should be a string which "documents" the variable. It will be remembered somewhere where no one is likely to find it.

defparameter *variable init* [*documentation*] *Macro*

This is like defvar, except that *variable* is unconditionally set to the value of *init*. Thus, the *init* form must always be specified.

defconstant *variable init* [*documentation*] *Macro*

This is like defparameter, but additionally states that *variable* is a constant. It is an error to bind or setq that variable. Sometimes the NIL compiler may take advantage of this information. If, when a defconstant form is evaluated, *variable* already has a value which is different from what *init* evaluates to, a correctable error is signalled.

defconstant will fail to make *variable* truly constant if it already has a value not assigned by defconstant, or even if there is compiled code which references *variable*.

7. Predicates

7.1 Equality

eq *object1 object2*

Returns **t** if *object1* and *object2* are "the same". In general,

```
(setq x (compute-something))
(eq x x)
  => t
```

Equal numbers are not necessarily **eq**, however. For numeric equality, one should use **eql**, or, if arithmetic equality irrespective of type is desired (i.e., conversion from integer to floating may be needed), **=** (page 28) should be used.

eql *object1 object2*

This predicate is a slight extension of **eq**. It is used as the default equality predicate by many NIL functions. Essentially, it is **equal** for non-structured objects: two numbers are **eql** if they are of the same type and numerically equal, character objects are **eql** if they represent the same characters, etc. However, structured objects, such as lists, strings, vectors, and arrays, are **eql** only if they are **eq**. Thus,

```
(eql (add1 (sub1 0.0)) 0.0)      =>      t
(eq (add1 (sub1 0.0)) 0.0)       =>      nil
(eql (cons 'a 'b) (cons 'a 'b)) =>      nil
```

equal *object1 object2*

This is the more traditional (and more general, in some sense) equality predicate. If *object1* and *object2* are not structured, then they are compared like **eql** (above) does. Two lists are **equal** if their **car**s and **cdr**s are respectively **equal**. Otherwise, if they are both arrays (which includes strings, vectors, and bit-vectors), they are **equal** if they have the same rank and dimensions, the same element types (a string cannot be equal to a non-string array of characters), and **equal** elements.

Note that in MACLISP and LISP MACHINE LISP, **equal** never descends into arrays other than strings. In NIL, it is more consistent in its handling of the various types of array. Note also that in LISP MACHINE LISP, the special treatment of strings is different in that it performs a case *independent* comparison, which is inconsistent with both the ideas of simply recursively descending into the object and performing recursive **equal** tests, and the heuristic that two objects are **equal** if the "print" the same.

7.2 Types

null *x*
Returns t if *x* is nil, nil otherwise.

not *x*
Same as null. not implies that nil is representing boolean *false*, null implies that it represents the empty list.

atom *x*
t if *x* is not a cons, nil if it is. Note that arrays (and strings and vectors) are all atoms, even though they have structure (which may be descended by such functions as equal).

listp *x*
t if *x* is a cons or nil (the empty list), nil otherwise. Note that listp does not examine anything but the type of the object; it does not verify that *x* is actually a "proper list", one whose last cdr is nil.

Note that in LISP MACHINE LISP, listp is false for nil. NIL is compatible with MACLISP (and COMMON LISP) here.

consp *x*
t if *x* is a cons, nil otherwise.

integerp *x*
fixp *x*
t if *x* is an integer, nil otherwise. fixp is provided for MACLISP compatibility. It is entirely identical to integerp.

floatp *x*
t if *x* is any kind of floating-point number, nil otherwise.

numberp *x*
arrayp *x*
vectorp *x*
stringp *x*
The list goes on and on.

typep *object* &optional *type*
If no *type* argument is supplied, this returns the name of the type of *object*. This returned result is almost always something much too internal for practical use, and is certainly not compatible with MACLISP. In this case typep isn't even a predicate.

If the *type* argument *is* supplied, then typep returns t if *object* is of the type *type*, nil otherwise. *type* may be a general COMMON LISP type specifier. Typically it is a symbol which names a type. An exhaustive list of the allowable types is not provided here (and is, in fact, extensible anyway). It includes such type names as null, cons, list, integer, rational, number, character, string, array, closure, vector, function, hash-table, and stream. typep will complain if *type* is not a valid type specifier.

8. Control and Program Structure

8.1 Binding

let *letlist* {*declaration*}* *forms...* *Special Form*

let binds the variables to the values specified in *letlist*, and evaluates *forms* in that environment, returning the value(s) resulting from the evaluation of the last *form*. The *letlist* is a list of the form

({*variable* | (*variable*) | (*variable init-form*)}*)

In the first two variants, the variable will be bound to nil; in the last, to the evaluation of *init-form*. All of the *init-forms* in the letlist are evaluated before any of the variables are bound. For example,

```
(let ((x 1) (y 2))
  (let ((x y) (y x))
    (list x y)))
==>  (2 1)
```

That form is equivalent to

```
((lambda (x y)
   ((lambda (x y)
      (list x y))
    y x)))
 1 2)
```

let* *letlist* {*declaration*}* *forms...* *Special Form*

let* is like let syntactically, but the variables are bound sequentially rather than in parallel. That is, first the first init-form is evaluated and the first variable bound to that, then the second init-form is evaluated in that new environment, etc. So,

```
(let* ((x 1) (y 2))
  (let* ((x y) (y x))
    (list x y)))
=> (2 2)
```

8.2 Flow of Control

cond {(*predicate-form* {*consequent*}*)}* *Special Form*

Ordinary everyday LISP cond. Each "clause" is examined in succession: the predicate-form is evaluated, and if the result is not nil, then the *consequents* of that clause are evaluated, and the value(s) of the last are returned as the value(s) of the cond, without further examination of any other clauses. If there are no *consequents* in the clause, the value the predicate form returned is returned instead. If no predicate-form "succeeds", cond returns nil.

```
(defun fact (n)
   (cond ((= n 0) 1)
         (t (times n (fact (sub1 n))))))
```

if *predicate-form then-form* [*else-form*] *Special Form*

A simpler binary **cond**.

```
(defun fact (n)
  (if (= n 0) 1 (times n (fact (sub1 n)))))
```

case *item* {(*keyspec* {*consequent*}+)}* *Special Form*

This is similar to, but somewhat more general than, the **caseq** and **selectq** macros of MACLISP and LISP MACHINE LISP (which NIL define in terms of **case**). The *item* is evaluated, then is matched against *keyspec* of each clause in turn. If it matches, then the *consequent*s of that clause are evaluated, and the value(s) of the last returned as the value of the **case**; if none match, the **case** form evaluates to **nil**.

A *keyspec* may be an atom or a list. The key matches if *keyspec* is a list and the value of *item* is a member (using the predicate **eql**—see page 18) of that list, or if *keyspec* is the atom **t** (allowing an "otherwise" clause), or if *keyspec* is an atom and is **eql** to the value of *item*.

```
(defun integer-description (n)
  (case n
    (0 "none at all")
    (1 "just one")
    (2 "a couple")
    ((3 4) "a few")
    ((5 6 7) "several")
    (t "many")))
```

8.3 Iteration

mapl *function* &rest *lists*
mapc *function* &rest *lists*
maplist *function* &rest *lists*
mapcar *function* &rest *lists*
mapcon *function* &rest *lists*
mapcan *function* &rest *lists*

These are the "traditional" functions for iteration down lists. There must be at least one list specified; the iteration terminates when any list "runs out". The function is called on as many arguments as there are lists. **mapl**, **maplist**, and **mapcon** apply the function to successive sublists of the lists; **mapc**, **mapcar**, and **mapcan** to successive elements.

mapl and **mapc** are primarily for effect; they both return the first *list* argument given to them.

maplist and **mapcar** return a new list, the elements of which are the results of applications of *function*. That list will thus have as many elements as the shortest of the *lists*.

mapcon and **mapcan** "splice together" the results of the applications of *function*, (as if) by **nconc**. This allows multiple new elements (or none) to be returned.

```
(mapc #'print '(a b c))
```
prints
```
a
b
c
```
and returns
```
(a b c)
(mapl #'print '(a b c))
```
prints
```
(a b c)
(b c)
(c)
```
and returns
```
(a b c)
(mapcar #'(lambda (x y) (plus (times x 2) y))
        '(1 2 3) '(4 5 6))
  => (6 9 12)
(defun odd-ones (l)
  (mapcan #'(lambda (x) (and (oddp x) (list x))) l))
(odd-ones '(1 2 3 4))
  => (1 3)
(defun destructively-remove-duplicates (l)
  (mapl #'(lambda (sublist)
            (rplacd sublist
                    (delq (car sublist) (cdr sublist))))
        l))
(destructively-remove-duplicates '(a b b c b d e c))
 => (a b c d e)
```

Note that **mapl** is the same as the MACLISP and LISP MACHINE LISP function **map**. In NIL, the function **map** is a more general sequence iteration function, which takes different arguments.

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 8.4, page 23.

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 8.4, page 23.

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

Similar to dolist, but for vectors.

8.4 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*.

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 do, dolist, dotimes, dovector.

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.

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.

loop *gubbish...* *Macro*

loop is described in another document [4]. It is noted here because it deals correctly with the incompatibilities between LISP MACHINE LISP and COMMON LISP blocks and returns.

8.5 Generalized Variables

setf *place value* *Macro*

setf is a general macro which is used to modify things. *place* is a form which describes the place to be modified; it may be something like x, a variable, (car *something*), or even a user-defined structure reference. It expands into code which will modify that *place* to be *value*, additionally guaranteeing that the return value of the setf form is that value. Thus, one may use

```
(setf (car l) 'foo)
```

instead of

```
(rplaca l 'foo)
```

The utility of this is that the special functions (if indeed there are any) necessary to perform the specified updating need not be known about. In fact, COMMON LISP does not even define most updating functions, only that the referencing functions may be used with setf.

If *place* is a variable, then setf is equivalent to setq. *place* may also be any car/cdr function, sequence function (such as nth, elt, vref, and bit), or any structure accessor defined by either defflavor (which is not described here) or defstruct (in the Maclisp Extensions Manual, [5]).

9. List Manipulation

car
cdr
c**r**
car and cdr combinations to 4 deep, as in MACLISP.

cons *x y*
ncons *x*
xcons *x y*
list* *e1 e2 e3 ... en last-cdr*
list *e1 e2 e3 ... en*

append *{list}**

```
(append '(a b c) '(d e f)) => (a b c d e f)
```

All but the last of the *list*s are copied.

nconc *{list}**
Concatentates the lists, destructively; the last cdr of each is bashed to point to the following list.

```
(setq a '(a b c))
(setq b nil)
(setq c '(d e f))
(nconc a b c)
   => (a b c d e f)
a  => (a b c d e f)
b  => nil
c  => (d e f)
```

nth *index list*
Returns the *index*th element of *list*, zero-origined. nth is compatible with the nth of MACLISP and LISP MACHINE LISP, and differs from that of INTERLISP. Note that the argument order differs from that used for most other sequence-accessing functions defined by NIL and COMMON LISP.

nthcdr *index list*
Returns the result of cdring *list index* times.

last *list*
Returns the last cons of *list*, unless *list* is nil in which case nil is returned.

nreconc *list other*
This destructively reverses *list* and bashes its last cdr to be *other*; that is, it is like doing

```
(nconc (nreverse list) other)
```

Certain other list operations, such as reverse and nreverse, are actually generic sequence operations, described in section 13.1, page 33.

9.1 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 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).

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.

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

10. Symbols

get *symbol indicator*

Although get works on "disembodied property lists" for MACLISP and LISP MACHINE LISP compatibility, it treats a first argument which is neither a symbol nor a list as an error. (MACLISP get will return nil for other objects.)

putprop *symbol value indicator*

As in MACLISP, works on both symbols and disembodied property lists.

remprop *symbol indicator*

MACLISP-compatible remprop. If *symbol* (which may also be a disembodied property list) has an *indicator* property, it is spliced out of the property list, and the subpart of the property list whose car is (was) the value for that indicator is returned. If there is none, nil is returned.

plist *symbol*

Returns the propertly list of *symbol*. Does *not* work on disembodied property lists.

setplist *symbol new-plist*

Bashes the property list of *symbol* to be *new-plist*. Does *not* work on disembodied property lists. Use of this should normally be left to only those functions which carefully interlock addition and deletion from the property list, to avoid both timing screws and elimination of properties needed by "someone else".

11. Numbers

NIL numbers may be integers, floating-point, ratios, or even complex. Most arithmetic functions operate on any types, and convert as necessary; in general, contagion proceeds from integer -> ratio -> float -> complex. For the operations performed on the real and imaginary parts of complex numbers, the conversion is similar.

Complex numbers in NIL are on the order of days old. Although they exist, they will not be supported by most functions in NIL for some time.

= *number* &rest *more-numbers*
/= *number* &rest *more-numbers*
Return t if all of the numbers (in all pairwise combinations) are equal or not-equal respectively. Performs type conversions as necessary to do the comparisons.

< *number* &rest *more-numbers*
<= *number* &rest *more-numbers*
>= *number* &rest *more-numbers*
> *number* &rest *more-numbers*
The numbers must all be real numbers. Each function returns t if all consequetive pairs of arguments satisfy the appropriate comparison. Type conversions are performed as necessary to do the comparisons.

zerop *number*
t if *number* is integer, floating, or complex 0, nil otherwise.

plusp *real-number*
minusp *real-number*

1+ *number*
add1 *number*
Adds one to *number*.

1- *number*
sub1 *number*
Subtracts one from *number*.

+ &rest *numbers*
plus *numbers*
Adds together the numbers.

- *number* &rest *more-numbers*
(**-** *number*) performs unary negation on *number*. With more than one argument, it subtracts the sum of the rest from the first.

difference &rest *numbers*

Subtly different from -; difference is MACLISP compatible. If there are no *numbers*, 0 (the identity) is returned. Otherwise, the sum of all the numbers but the first is subtracted from the first. That is, (difference *x*) => *x*, not (- *x*).

* &rest *numbers*

times &rest *numbers*

Returns the product of all the *numbers*. If none are given, * returns 1, the multiplicative identity.

/ *number* &rest *more-numbers*

(/ *number*) returns the inverse of *number*. Otherwise, / divides the first by the product of the rest. If all of the involved quantities are integers, then / will produce a ratio rather than truncating the answer (as quotient does).

quotient &rest *numbers*

If there are no *numbers*, then *quotient* returns 1, the multiplicative identity. Otherwise, it divides the first by the product of the rest; with exactly one argument, that argument is returned. If all of the involved quantities are integers, then quotient will return a truncated integer result, rather than convert to either a ratio or a floating result. This is compatible with MACLISP.

remainder *integer1 integer2*

(remainder *n1 n2*) => *r*, such that

```
(plus (times (quotient n1 n2) n2) r) => n1
```

This is MACLISP compatible.

expt *base power*

Raises *base* to the power *power*. If both are integers the result will be an integer. If either are floating, then the result will be floating, computed by logarithms.

exp *number*

Raises *e*, the base of natural logarithms, to the power *number*. This currently always uses floating point, and *number* must be real.

log *number* &optional *base*

Returns the logarithm of *number* in *base*, which defaults to *e*. This currently always uses floating point, so *number* and *base* must be real numbers.

sqrt *number*

Returns the (principal) square root of *number*. This currently uses only floating point, so *number* must be real although the result may be complex.

isqrt *integer*

Integer square root. Returns an integer (truncates).

sin *radians*
cos *radians*
tan *radians*
sinh *radians*
cosh *radians*
tanh *radians*
asin *real*
acos *real*
Standard trig. Only deal with floating point, hence the arguments must be real.

atan *x* &optional *y*

11.1 Logical Operations

Integers in NIL are all represented in twos-complement notation. They may be viewed as a vector of bits extended infinitely; a negative integer extended with ones, a non-negative integer extended with zeros.

logbitp *bit-number integer*
Returns t if the bit at position *bit-number* is on in *integer*, nil otherwise. *bit-number* is zero-origined, with 0 being the least significant bit. Thus,

```
(logbitp 0 1)            =>  t
(logbitp 0 -2)           =>  nil
(logbitp 1 1)            =>  nil
(logbitp 259259259 15)   =>  nil
(logbitp 259259259 -2)   =>  t
```

logand &rest *integers*
logior &rest *integers*
logxor &rest *integers*
logeqv &rest *integers*
lognand *integer1 integer2*
lognor *integer1 integer2*
logandc1 *integer1 integer2*
logandc2 *integer1 integer2*
logorc1 *integer1 integer2*
logorc2 *integer1 integer2*
lognot *integer*
These routines supercede the use of the **boole** function, providing more mnemonic names. "c1" and "c2" suffixes can be read as "-complement-first-arg" and "-complement-second-arg"; thus, **logandc2** is logical **and** of the first argument with the complement of the second argument.

integer-length *integer*

This returns the size of *integer*. This is defined such that *integer* in twos-complement notation can fit in a field that long plus one.

```
(integer-length 0)      => 0
(integer-length -1)     => 0
(integer-length 1)      => 1
(integer-length -2)     => 1
```

11.2 Conversions

float *number*

Converts number to floating-point.

fix *number*

Converts *number* to an integer, by flooring (rounding towards negative infinity). This is compatible with MACLISP.

12. Characters

Here are some of the functions on characters which may be useful.

char-equal *character1 character2*
Returns t if *character1* and *character2* are the same character, ignoring case, font, and bits, nil otherwise.

char-greaterp *character1 character2*
Returns t if *character1* is "strictly less than" *character2*, ignoring case, font, and bits. All alphabetic characters, and all digits, collate in the obvious way with respect to one another.

character *frobozz*
Coerces *frobozz* into a character.

char-code *character*
char-bits *character*
char-font *character*
These functions return the *code*, *bits*, and *font* attributes of their argument, as an integer.

code-char *code* &optional *bits font*
Returns a character with code attribute *code*, bits attribute *bits*, and font attribute *font*. *bits* and *font* default to 0. If it is not possible to construct such a character (possibly because NIL does not support code, bits, or font attributes of the values given) then nil is returned.

make-char *character* &optional *bits font*
Just like
```
(code-char (char-code character) bits font)
```

All characters have unique integer representations. The following two functions convert back and forth from integers to characters. They are what are used to provide that mapping for the MACLISP I/O functions which deal with integers rather than characters.

char-int *character*
Converts *character* to its integer representation. By definition, if *character* has no (i.e., 0) *bits* and *font* attributes, then char-int is the same as char-code. char-int can be used as a fast hash function on characters.

int-char *integer*
This is the inverse of char-int. If *integer* is in the range which might be returned by char-int on some character, then that character is returned; otherwise, nil is returned.

There is also a large number of predicates on characters for testing such things as alphabetic-ness, digit-ness, and for converting back and forth from digits to integer-weights. These are all documented in the NIL release notes and the COMMON LISP manual [1].

13. Sequences, Strings, and Arrays

The parts of NIL dealing with sequences, strings, and arrays, are gradually being brought up to COMMON LISP specifications. If more information is needed than is presented here, either the NIL Release Notes or a bootleg Common Lisp Manual should be consulted.

13.1 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*.

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.

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.

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,

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

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))
```

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*

...

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.

13.2 Strings

There is a large complement of string functions in NIL, only a few of which will be mentioned here. Mostly they are like those used in LISP MACHINE LISP.

string-equal *string1 string2* &optional *start1 start2 end1 end2*

This routine is the basic routine for doing case-independent string equality tests. That is,

```
(string-equal "foo" "Foo")  =>  t
(equal "foo" "Foo")         =>  nil
```

As usual, the starts default to 0, and the ends to the lengths of the strings. If the lengths of the specified substrings differ, then the strings are not string-equal. Note that this function takes positional rather than keyworded optional arguments, for historical reasons.

string-lessp *string1 string2* &optional *start1 start2 end1 end2*

Returns t if the specified substring of *string1* is "less than" that of *string2*. The comparison is case-independent; effectively, lowercase characters are compared as if they were uppercase. If the first substring is a prefix of the second, then it is "lessp".

substring *string start* &optional *end*

Like subseq (page 33). If the specified range is the entire string, the string may not be copied.

string-upcase *string*

Returns a copy of *string*, with all lowercase characters converted to uppercase.

string-downcase *string*

Returns a copy of *string*, with all uppercase characters converted to lowercase.

13.3 Arrays

NIL supports a large portion of the COMMON LISP array specification currently. The MACLISP compatible array routines have been lost during the conversion, and have not been re-implemented yet. What is described here is a simple subset of what is offered.

In NIL, a one-dimensional array is a vector. Some types of arrays have restrictions on the types of objects which can be stored in them. For example, arrays which can only hold string characters are string-char arrays, and if they are one-dimensional (and hence vectors), they are strings. An array which can hold any type of object is termed a general array. At this time, that is the most useful type of array (other than the very common special types string and bit-vector), so that is all that will be discussed here. Also, obscure, esoteric, and non-working features have been omitted.

In NIL, an array is an object all its own. They do not need to be associated with symbols or treated as functions in any strange way (as had been the case in MACLISP in the past).

make-array *dimensions*

This creates an array with the specified dimensions. *dimensions* may be either a single non-negative fixnum, or a list of dimensions. If it is a fixnum, then a one-dimensional array (vector) is created; otherwise, the created array will have a rank of the number of dimensions specified. Thus, a 10x10 array could be created with

```
(make-array '(10 10))
```

aref *array* &rest *indices*

Returns the element of *array* selected by the *indices*. The number of indices must equal the rank of the array, and each index must be a non-negative fixnum less than the size of the corresponding dimension.

To update an element of an array, use setf with aref:

```
(setf (aref a i j) new-value)
```

array-rank *array*

Returns the rank (number of dimensions) of *array*.

array-dimension *array dimension-number*

Returns the size of the *dimension-number* dimension of *array*. The dimensions are numbered zero-origined; dimension-number must be non-negative, and less than the rank of *array*. If an array had been created by (make-array '(2 3 4)), then its 0 dimension is 2, its 1 dimension 3, etc.

array-dimensions *array*

Returns a list of the dimensions of *array*. This could then be given to make-array to make another array with the same dimensionality.

14. Input and Output

14.1 Streams

I/O in NIL uses *streams*. The operations performed are implemented by sending messages to these streams; there are various levels of protocol involved, and it sometimes gets quite complicated. However, for most ordinary applications, there are functions which may be used instead, providing a simpler interface, and some argument defaulting.

There are several variables whose values are the streams used as the defaults in various contexts. If the name of the variable ends with "-input", then it is only expected to be used for input; "-output", then only for output; and "-io", then both.

terminal-io *Variable*

This is the stream which reads from and writes to the terminal.

standard-input *Variable*

This is the "default input stream", used as a default by functions which can default a stream argument and expect to use it for input. Normally, it is bound to a stream which indirects to the value of terminal-io. Some contexts may bind it, however; when one is compiling a file, or loading an interpreted lisp file, it will be bound to a stream which reads from that file.

standard-output *Variable*

This is the "default output stream", used similarly to standard-input, but for output. Normally, it is bound to a stream which indirects to the value of terminal-io; it may sometimes be bound to something else, however, potentially even a stream which "broadcasts" to multiple other streams.

query-io *Variable*

This is a stream used by functions which query the user, such as yes-or-no-p (page 41). Normally it indirects to the value of terminal-io, although it could conceivably by a stream which logs the interaction.

Most of the stream functions (with a couple exceptions) also allow the stream to be one of the symbols t or nil. t is taken to mean the value of terminal-io, and nil the default stream for the operation; the value of either standard-input or standard-output, depending on whether an input or output operation is implied. The main exceptions are the (MACLISP compatible) cursorpos function (which is not described here), which defaults to using terminal-io, and the format function (page 41). Also, some operations do not inherently have any directionality, so cannot determine the meaning of nil as a stream (none of them are described here anyway).

14.2 Basic I/O

14.2.1 Input

Most input functions take (possibly in addition to some other arguments) optional *stream* and *eof-value* arguments. If the stream argument is null or not supplied, the value of standard-input is used; if it is t, the value of terminal-io is used. The *eof-value* argument determines the behavior of the function if it is reading from a stream which can run out of data, as a disk file might when the end of the file is reached. If no *eof-value* argument is supplied, then the function does not handle the end-of-file condition: an error is signalled. Otherwise, that value is returned by the function, in place of whatever object it would have read. Note, however, that end-of-file detected after the start of an object is always an error; thus, read will always error if there are (say) too few close-parentheses in the expression, but will return its *eof-value* if end-of-file is detected before the next LISP expression starts.

read-char &optional (*stream standard-input*) *eof-value*
Reads one character from *stream*.

tyi &optional (*stream standard-input*) *eof-value*
Reads one character from *stream*, and returns its integer representation. This is provided only for MACLISP compatibility.

peek-char &optional (*stream standard-input*) *eof-value*
Returns the next character to be read from *stream*, but does not "advance the pointer". That is, repeated calls to peek-char with no intervening calls to read-char will return the same result, as will a single following call to read-char.

tyipeek *etc etc etc*
Tries hard to simulate MACLISP-compatible tyipeek. May not always succeed.

read &optional *stream eof-value*
Reads one LISP expression from *stream*.

readline &optional *stream eof-value*
Reads one line of text from *stream*, and returns it as a string. The newline is passed over, but is not part of the returned string.

14.2.2 Output

For these functions, a *stream* argument of nil is the same as specifying the value of standard-output, and a *stream* argument of t the same as specifying the value of terminal-io. This provides some semblance of MACLISP compatibility. (Also a nice abbreviation over using terminal-io.)

write-char *character* &optional (*stream standard-output*)
Writes *character* to *stream*. This *must* be an object of type **character**; a fixnum or a string are not acceptable.

tyo *integer-character* &optional (*stream standard-output*)
This is provided for MACLISP compatibility. *integer-character* must be the integer representation of a character (see **int-char**, page 32); it is converted to a character and given to **write-char**.

princ *object* &optional (*stream standard-output*)
Prints *object* to *stream*. If *object* is a string, then only the characters contained in the string are output; if it is a symbol, then only the characters of the print name are output.

prin1 *object* &optional (*stream standard-output*)
This is the basic entry to print a LISP object in such a way that it can be read back in again, hopefully. For the most part, this means that strings have doublequotes put around them, and maybe have backslashes preceding contained doublequotes and backslashes, and symbols have their package printed as a prefix and may have vertical bars surrounding them if there are any characters in the print name which would cause the symbol to not be read in as such.

print *object* &optional *stream*

```
(defun print (object &optional (stream standard-output))
  (prog2 (terpri stream)
         (prin1 object stream)
         (write-char #\space stream)))
```

terpri &optional *stream*
Does a newline on stream.

fresh-line &optional *stream*
If the "cursor" of *stream* is not at the beginning of a line, this does a **terpri**. Thus, it can be used to guarantee that subsequent output starts at the beginning of a line, while not producing superfluous blank lines.

pretty-prin1 *object* &optional *stream*
"Pretty-prints" the *object*. The object is assumed to be LISP code; i.e., lists which start with special function names like **defun**, **let**, and **cond** are treated specially, and **quote** and **function** forms are inverted to print out using the ' and #' prefixes. Note that this does not start the output on a new line; this allows one to print a prefix first.

This pretty-printer goes over the structure it is printing to detect circularities, and displays them using COMMON LISP labelling syntax (which should be fairly obvious when seen). This labelling is not understood by the reader, but stops the printer from dying.

pretty-print *object* &optional *stream*
pretty-prin1, with a terpri first.

format *destination control-string* &rest *arguments*
format provides ways to produce simple to fancy text output. A simple call might look like

```
(format t "~&The ~A count was ~D.~%" 'cat 10)
```

The t tells format its text to the default output stream (just like giving print no *stream* argument). Within the *control-string*, a tilde introduces a format directive. The directives in the above string mean (respectively) go to a new line if needed, princ an argument (taken from the *arguments*), print an argument as a decimal number, and output a newline (terpri). The output from that thus looks like

```
The CAT count was 10.
```

format differs from the other output functions in how it interprets its *destination* argument; t means use the value of standard-output, and nil means collect the output and return it as a string.

format is almost entirely compatible with the format used in MACLISP and LISP MACHINE LISP. In fact, it uses the same source as the one used in both PDP-10 and MULTICS MACLISP (and has some of the same bugs). The only things missing in the NIL version are the operations which deal with floating-point numbers. More details on format may be found in either the Maclisp Extensions Manual [5] or the Lisp Machine Manual; the former documents this version of format more accurately.

14.3 Interacting with the User

Some functions of potential interest.

y-or-n-p &optional *message stream*
Prints the string *message* to the stream, which defaults to the value of query-io, and then reads a character. y-or-n-p requires that the character be either y or n, and returns t if it is y or nil if it is n. It is unusual to specify the stream argument to y-or-n-p, as the default will almost always suffice.

yes-or-no-p &optional *message stream*
Somewhat like y-or-n-p. The input, however, is an entire line, which must be either yes or no. yes-or-no-p is oriented towards unexpected questions which may have significant consequences or require thought: it flushes all typeahead, and beeps.

format-y-or-n-p *format-string* &rest *format-args*
Like y-or-n-p, but uses format to display *format-string* with arguments *format-args*.

format-yes-or-no-p *format-string* &rest *format-args*
Like yes-or-no-p, but uses format like format-y-or-n-p.

readline-with-prompt *prompt-string* &rest *read-args*

This is a convenient way to read a line of text from the terminal, with a prompt string. It is better than printing the prompt yourself and calling **readline**, because the input processor knows about the prompt string and can reprint it as necessary. *read-args* are interpreted like **read** does; typically, none need to be given.

read-with-prompt *prompt-string* &rest *read-args*

Similar to **readline-with-prompt**, but uses **read**.

15. Running and Debugging Programs

15.1 Loading Programs

load *pathname* &rest *hairy-options*

load is the general function for loading a file of either LISP source or compiled LISP code into NIL. If no file-type (extension) is specified in *pathname*, then a file type of `vasl` (VMS extension `vas`) is looked for first, and if that is not found, a file type of `lisp` (VMS extension `lsp`) is tried. The manner of loading the file is determined from the file itself, as are various attributes about the file (such as its readtable and package).

load sets the default pathname it uses every time it is called. Therefore, (load "") effectively repeats the previous call to load.

15.2 The Interactive Debugger

Generally when errors are detected, the error message is printed out, and the *interactive debugger* is entered. This is a program which takes (mostly) single-character commands, and can be used to poke around the LISP stack and execution environment. You can reconize this, because it always prompts with >*n*>dbg>, where *n* is the recursion-level of the debugger. The ? command prints what other commands are available, and a brief description of what they do. The debugger has a "current frame" which it displays. Many commands, such as those to display local variables and arguments, operate with respect to that frame. Commands may be given arguments by typing the decimal digits before the command character. The most useful commands are

- D Examine the next frame down (less recent invocation).
- U Examine the next frame up (more recent invocation).
- A Print an argument (these are zero-origined). Because the arguments are printed with the frame, this is only really useful if the argument desired was not printed due to truncation of the printout.
- E Evaluate an expression (which is prompted for). If the expression is a symbol, then it is not actually evaluated; rather, some information about it is printed, which includes its current value. The subtle difference is to aid in debugging in a lexical environment. If E is given a numeric argument, then it will give a more detailed analysis of the variable.
- R Prompts for and reads a form, evaluates that in the current lexical environment, and returns that value as the value of the error function call (that is, the call to error, ferror, or cerror).

debug

The debugger may be entered directly from code by evaluating (debug).

It is worth noting that the debugger's idea of "down" and "up" is different from the normal interpretations of stack direction. This will someday be fixed, and confuse everybody.

15.3 Stepping

The NIL evaluator contains a hook with which every call into the evaluator may be trapped. *Stepping* is when one gets to interact with each such call, to follow what is happening during the evaluation process.

step *form* *Macro*

This evaluates *form* with stepping turned on. At each internal **eval** call, control returns to the stepper; typically, the stepper then prompts for a command, and continues (or not) depending on that command. There are some heuristics for handling certain forms which cause certain actions, including not waiting for a command. The most useful commands are

- C Continue stepping (steps into the current form, if that is possible).
- V Don't step "into" this form, just evaluate it and show the resulting value (and then continue as before).
- N Don't step "into" this form, don't even show its value; just evaluate it and continue.
- B Like V; and then enters a break loop. This is not so useful in a lexical environment; it is inherited from a MACLISP stepper.
- ? Lists all the defined stepper commands.

15.4 Tracing and Breaking

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.

trace &rest *trace-specs* *Macro*

With no arguments, **trace** returns a list of all currently traced functions. Otherwise, it traces each of the functions in the manner specified by *trace-specs*. In the simplest case, a *trace-spec* is simply the name of a function. Tracing then will show the function and the arguments it receives when it is called, and the value(s) it returns when it returns.

A *trace-spec* may also be a list of a function name, and options. For instance,

```
(trace (fact :break))
```

will trace the function fact, and enter a break loop when entering and exiting each call to fact. On entrance, in this break loop, the variable *trace-break-args* is set to the list of arguments given to the function; on exit, to the list of returned value(s). Each option may also be a list of the option, and a predicate; the predicate receives arguments about the call, and determines what happens with that option. This is not developed here.

untrace &rest *fns* *Macro*

With no arguments, this undoes tracing on all traced functions. Otherwise, it untraces only those functions specified. The function names are not evaluated.

trace-break-args *Variable*

When a break loop is entered during function tracing due to the :break option, this holds a list of the arguments to the function being traced (if the break was on entrance), or the list of value(s) returned (if the break was on exit). Changing this list will change the arguments given or values returned.

15.5 Exhibiting

exhibit *object*

Runs an interactive structure editor on *object*. "?" gives a list of single-character commands.

15.6 Printing Definitions

pp &optional *name* *Special Form*

If *name* is defined as a function, then this pretty-prints the form used to define it. If it has a (dynamic) value, then a setq form for that assignment is pretty-printed. If *name* is not supplied, then it defaults to the name given to the previous call to pp.

grindef &optional *name* *Special Form*

Same as pp.

16. Steve, the Editor

ed &optional *something*

Enters the editor. Currently, *something* may only be a pathname; that file will be read into a buffer (if that has not been done already), and that buffer selected. (ed) simply enters the editor, with the same current buffer as the last time it was used.

The NIL editor (which is named STEVE for a not particularly convoluted reason that will not be explained here) is mostly modeled after ITS/TOPS-20 EMACS [*ref*]. To the extent possible in the differing programming and operating system environment, it attempts to be compatible with the EMACS provided on TOPS-20. (For those familiar with the ITS emacs, this is to allow control-C to be reserved for interrupt usage under VMS; control-Z is used as the control-meta prefix.) People familiar with this EMACS should be able to skip the remainder of this chapter except for section 16.7; we are attempting to be compatible, but it is still experimental.

When STEVE is entered, it turns on *passall mode*, effectively disabling all special interpretations of input characters by VMS (see chapter 3, page 5, in particular page 7). This allows it to make as full use of the limited characters available with an ASCII keyboard as possible.

STEVE is a *realtime editor*. That is, inserts text into the buffer, and updates the display of the buffer contents, as it is entered. Most "normal" ASCII characters are set to simply insert themselves into the buffer; most "control" characters are set to do something other operation.

The commands to STEVE are based on an extended character set, similar to the virtual extended character set in EMACS. That is, in principal there may be different commands on each of the characters *control-A*, *meta-A*, *super-A*, *hyper-A*, *control-meta-A*, etc. Because practically none of these characters can be typed on a normal ASCII keyboard (the exceptions being only the *subset* of the control characters which get specially mapped into the extended character set as it is), STEVE defines certain of the simple control characters to be prefixes which add the extra bits to the following character. The character #\altmode (escape), for example, is *prefix meta*; typing the sequence *altmode a* is the same as typing *meta-a*. Note that the case of the basic character is irrelevant.

Here are the prefix and dispatch characters defined:

Control-↑

This is *prefix control*. It can be used to type in a control character which is not part of the ASCII character set, such as *control-space*.

Altmode

Prefix meta. The next character is interpreted with the *meta* bit added. Thus, *altmode altmode* is like typing *meta-altmode* (which is the command to enter the minibuffer).

Control-Z

Prefix control meta. The next character is interpreted with *both* the control and meta bits added.

Control-X

This prompts for another character and dispatches off of it, as opposed to being a bit-prefix.

There are no prefix characters defined for the *super* or *hyper* bits yet, nor are there any commands defined on such characters.

16.1 Files and Buffers

STEVE is capable of editing multiple files "at the same time"; that is, of maintaing the state of several editors at once. Each file being edited is in a buffer of its own, independent of all the others. The most common commands needed to edit and save files are:

C-X C-F — Find File
This command prompts for a filename. If STEVE already has a buffer with a pathname which matches the pathname you specify, that is selected (it becomes the "current buffer", the one you are editing). If not, a buffer name is generated from the pathname, that buffer is created (you are asked whether to reuse one if a buffer with that name already exists), and the file is read into that buffer.

C-X C-S — Save File
This writes the file to the *buffer filename*, iff the buffer has been modified. There is a little "*" which appears as the rightmost piece of information in the modeline, if the buffer has changed since it was last read or written.

C-X B — Select Buffer
Prompts for a buffer name (displaying the default, which is what a null line will select), and selects that buffer, creating it if it does not exist already.

C-X C-B — List Buffers
Displays a list of all the buffers STEVE knows about, and some information about them.
The above are usually all that are needed as file commands for simple editing uses.

16.2 Arguments

Many commands take signed numeric arguments. Typically (but not always) they are used as repeat counts for searches, kills, movement, etc. There are a few different ways in which in which arguments may be accumlated for commands.

Standard argument commands accumulate when used consequetively, and give the following command an argument of the "decimal" representation of the accumlated characters. Both *control-* and *control-meta-* digits and hyphen (minus sign) are assigned this way. Thus, the sequence *C-M-2 C-M-3 C-N* runs the control-N command with an argument of 23; presumably, this will move the cursor down 23 lines. *C-M-8 **** will insert "********" into the buffer, and *C-5 5* will insert "55555".

auto argument commands are a slight extension. They allow multiple-digit arguments to be typed without having to use argument commands for any digits but the first. The *meta* digits and hyphen (minus sign) are all auto-arguments. Thus, to give *C-N* an argument of 23, only *M-2 3 C-N* need be typed. The effect of this is that one only needs to type *altmode* (i.e., *escape*), then the digits, then the command (or character to be repeated).

The *universal argument*, which is assigned to *C-U*, is like auto-argument, and then some. A numeric argument 23 could be typed as *C-U 2 3*. With no digits (or minus sign), *C-U* is an argument of *4*; with or without, subsequent consequetive *C-U*s multiply the argument by 4. Thus *C-U C-U C-N* gives *C-N* an argument of 16, *C-U C-U C-U C-N* gives it an argument of 64, etc.

16.3 Movement Commands

C-A — Beginning of Line
Goes to the beginning of the line.

C-E — End of Line
Goes to the end of the line.

C-F — Forward Character
Moves forward one character (or *argument* characters).

C-B — Backup Character
Backspace — Backup Character
Moves backwards one character (or *argument* characters).

M-F — Forward Word
Moves forward one "word" (or *argument* words).

M-B — Backward Word
Moves backward one "word" (or *argument* words).

C-N — Down Real Line
Moves down a line (or *argument* lines).

C-P — Up Real Line
Moves up a line (or *argument* lines).

C-M-S — Forward S-Expression
Moves forward one (or *argument*) LISP expressions.

C-M-P — Backward S-Expression
Moves backwards one (or *argument*) LISP expressions.

C-M-A — Beginning of Defun
Moves to the beginning of the current "defun" (toplevel s-expression).

C-M-E — End of Defun
Moves to the end of the current "defun" (toplevel s-expression).

M-<— Goto Beginning
Goes to the beginning of the buffer.

M->— Goto End
Goes to the end of the buffer.

Movement by screen:

C-V — Next Screen
Goes forward one (visual) screenful. A couple lines of context are left. Giving an argument to this command makes STEVE adjust the displayed portion of the buffer forward

that many lines.

M-V — Previous Screen
Similar, only in the other direction.

C-L — New Window
Clears and completely redisplays the screen.

16.4 Deleting Text

Rubout — Backward Delete Hacking Tabs
Deletes one (or *argument*) character backwards. Tabs are treated as if they were spaces, so you don't have to know which were in the buffer.

C-Rubout — Backward Delete Character
Deletes one (or *argument*) character backwards. Tabs are *not* turned into spaces before being deleted.

C-D — Delete Character
Deletes on (or *argument*) character forwards.

M-Rubout — Kill Word Backward
Deletes one (or *argument*) word, backwards.

M-D — Kill Word
Deletes on (or *argument*) word, forwards.

C-M-Rubout — Backward Kill S-expression
Deletes one (or *argument*) LISP expressions, backwards.

C-M-K — Kill S-expression
Deletes one (or *argument*) LISP expressions, forwards.

C-K — Kill Lines
Deletes lines. With no argument, kills from the cursor to the end of the line, unless the cursor is at the end of the line, in which case the newline itself is killed. With an argument, kills that many whole lines.

When text (other than single characters) is deleted, it is saved on a *kill ring*. Consequetive deletion commands (no other intervening commands) are saved together, as if only one kill had been performed.

C-Y — Unkill
Restores the text most recently killed.

M-Y — Un-Kill Pop
If what you restored with C-Y was not what you wanted, M-Y will replace it with the previous block of text killed. This may be repeated some number of times.

16.5 Searching

C-S — Incremental Search
Reads text from the terminal, and searches for it and updates the display, as you type in the text. The search may be repeated by re-typing C-S, or reversed by typing C-R. *Rubout* may be used to delete characters being searched for; the search then backtracks. *Altmode* exits the search, leaving the cursor where it was last shown. Any "command" character also exits the search, then executes that command.

C-R — Reverse Incremental Search
Do the same, only backwards.

16.6 Other Stuff

Regions, marking, transposition, appending kills, etc. etc. etc. See [*ref*]...

16.7 Interacting with NIL

C-M-Z — Exit Editor
C-X C-Z — Exit Editor
Return from the editor (to its caller). This is currently always the caller of **ed**; one of these commands, or some other, should eventually be usurped to return directly to the VMS CLI.

M-Z — Zap to Lisp
The current **defun** (actually, toplevel s-expression the cursor is contained within or immediately after) is evaluated. This is convenient to selectively evaluate a fixed-up form in a file which has already been loaded.

M-X Compile-Sexp
Similar to M-Z, but *compiles* the form instead.

M-X Compile-This-File
Compiles the file being edited (offering to save it first if necessary).

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. This can be copied for the curious.

3. Weinreb, D., and Moon, D., *Lisp Machine Manual*, MIT Artificial Intelligence Laboratory, Cambridge, Mass., (July 1981). A newer edition of this is available from the AI Publications Office.

4. Burke, G. S. ,and Moon, D., *Loop Iteration Macro*, TM-169, MIT Laboratory for Computer Science, Cambridge, Mass., (January 1981). Available from the LCS Publications Office.

5. Bawden, A., Burke, G. S., and Hoffman, C. W., *Maclisp Extensions*, MIT Laboratory for Computer Science, Cambridge, Mass. TM-203, July 1981. Available from the LCS Publications Office.

6. Moon, D. A., *MACLISP Reference Manual*, MIT Laboratory for Computer Science, Cambridge, Mass., (1974). Reprints of portions of this are available from the LCS Publications Office.

7. Burke, G. S., Carrette, G. J., and Eliot, C. R., *NIL Notes for Release 0.5*, MIT Artificial Intelligence Laboratory, Cambridge, Mass., Working Paper, in preparation.

Index