Revised NISP Manual

A tastefully wrapped version (0.2) of the classic from
August 1988
YALEU/DCS/RR #642

Drew McDermott

This research was supported in part by NSF grant IRI-8610241, by DARPA/BRL grant DAAA15-87-K-0001 and by Army-CSW grant DAAA10-86-C0604.

Contents

Chapter 1

Preliminaries

Note on 2004 edition: This manual needs to be completely rewritten, and it will be at some point. However, it's not going to happen today, or tomorrow. The current version, 2.9, differs from the version described here in one basic way: The NILS package has been replaced by the YTools package. They look very similar, but the latter is streamlined and made more consistent and elegant in various ways. The documentation for YTools is up-to-date; the manual is in the `ytools` directory. So, to read this manual, read the section on NILS, read the YTools manual, and then whenever a reference to a NILS construct is made in chapter 4, replace it with the corresponding YTools construct. `loop` becomes `repeat`, logical pathnames change from `foo/` to `%foo/`, and so forth.

Also, the goal of compatability with the T Lisp dialect has gone away, because T has gone away. I've thought about producing a Scheme version, but it would be a lot of work for a doubtful reward. The only target of Nisp now is Common Lisp. The goal has gone by the wayside of straddling several Lisp dialects by providing a moderate-sized set of facilities that can be implemented in all of them. YTools is a set of enhancements to Common Lisp, and Nisp programmers are now encouraged to use any Common Lisp construct they like. There are a few gaps left in the language where type declarations can't be integrated into Lisp syntax the way one might expect. If anyone stumbles into one, we will close it.

A big change not reflected in this manual at all is the availability of CLOS-style objects and generic functions. The document `nispobj.pdf` describes how it works.

A huge caveat in reading this documentation is that the capitalization conventions of Nisp have changed radically since it was written. I now use lowercase for ordinary symbols like `cdr` and `most-positive-fixnum`. I capitalize the first letter of types — including built-in types and classes defined by `defstruct`. Also, these are no longer just conventions. All my code has been updated so it runs in modern mode, to use the phrase coined by Franz to describe the case-sensitive mode that should have been adopted years ago by everyone. (Modern-mode code, if written with a bit of care, works just fine in ANSI Lisp; the pain for the devotee of modernism occurs when he tries to port someone else's ANSI code.) So, in reading what follows, adjust most upper-case letters down, and up-case the first letter of a lower-case name. In a construct such as `(LST float)` (in the old style), the current rule is to treat `LST` exactly the same as the type `float`. So the new version is `(Lst Float)`.

It's traditional to change the meaning of the acronym for Nisp whenever a sufficiently new version is introduced. It now stands for Numinous lISP. — Drew McDermott

NISP (Neutral lISP) is designed as a set of macros that can run in either Common Lisp or T, providing a concise and compatible interface to either. Common Lisp is the new Lisp standard, which is succeeding beyond (my) expectations in actually becoming standard. Its design is as elegant as possible for a dialect that consists of the assemblage of all previous Lisp dialects. T is the opposite, an effort to start from scratch and do things right. Both dialects are lexically scoped, but they differ in other respects. Common Lisp adopts the traditional identification of `NIL`, the empty list, and falsehood, as well as the traditional maintenance of two kinds of symbol value, for functions and arguments. T has just one kind of value, and `NIL` is just a variable whose value is falsehood. Common Lisp has extended the function call syntax in ways pioneered by Lisp Machine Lisp, with various kinds of keyword and whatnot. T retains traditional positional notation, plus the ability to pass arguments as a list. Common Lisp addresses namespace conficts with "packages" that allow multiple symbols with the same print name. T has just one namespace, but multiple variable-binding "environments."

As far as taste goes, T seems to be right in most cases, and Common Lisp wrong. Unfortunately, Common Lisp is the standard, in more ways than one. Thousands of person-hours are devoted to implementing, maintaining, and documenting it, whereas T is the work of a handful of eccentrics. Like Scheme, its parent, people who like T become fanatically devoted to it, but they have to put up with a lot, including a compiler that works correctly and efficiently only for stylistically fashionable code.

NISP is an effort to survive in both these worlds. (Getting the best of them would be too much to ask.) It looks more like Common Lisp than T, but implements T's object-oriented features, plus some macros that improve both languages. In addition, it provides compile-time type declarations that are more concise than Common Lisp's. A function that is written thus in Common Lisp

```
(DEFUN FOO (I X)
   (DECLARE (fixnum I) (float X))
   (FLOOR (* I X))   )
(PROCLAIM '(FTYPE (FUNCTION (fixnum float) integer) FOO))
```

can be written thus in NISP:

```
(DEFFUNC FOO - integer (I - fixnum X - float)
   (FLOOR (* I X))   )
```

NISP is written in three distinct layers:

1. **Core NILS**: The core NISP Implementation Language Subset, which is implemented via macros and function definitions in the host dialect.
2. **Full NILS**: Written entirely in Core NILS
3. **NISP Type System**: Written entirely in Full NILS

This manual reflects this division. First NILS is described, then the utilities that make up Full NILS, then the type system. The Core NILS system consists of a handful of files, amounting to about five thousand lines of Lisp code, depending on the dialect. When NISP is started, only this core system needs to be around. If a program depends on all of NILS or the type system, it should say so, and these components will be loaded.

Historical Note

This is the third incarnation of NISP. The first version ("Number Lisp") was an efficient number package for UCI Lisp, which eliminated consing of numbers; we hope the host Lisp compiler can now do that for us, given enough declarations. The second version ("Nifty Lisp") attempted to cover a wide range of Lisp dialects: UCI Lisp, Franz Lisp, Zetalisp, and T. Since these had varying external representations, it was necessary to provide a character-level translator, which is no longer necessary.

Over the years, many people have contributed ideas and code to Nisp, including Eugene Charniak, Ernie Davis, Denys Duchier-Proust, Jim Firby, Steve Hanks, Chris Riesbeck, and Larry Wright. The present manual was designed and edited by Larry Wright.

Common Lisp has solved many portability problems for us. NISP is still needed, to bridge the gap to T, and to provide compile-time types. However, wherever possible NISP has adopted Common Lisp conventions for function names, indexing conventions, etc.

Notational Conventions

Whenever a funoid or other feature is introduced, it appears on a special line indicating arguments (if any) and labeling it as one of the following:

Function: A procedure that takes zero or more arguments and returns zero or more results.

Magic: A reserved word or syntax extension.

Global Variable: A globally bound variable.

Type: NISP type.

Other: None of the above.

T and Common Lisp agree on most lexicographic issues. The default case is upper, and lower-case characters are changed to upper at read time. The escape character is "\". The glaring difference between them is the behavior of "`:`" — which behaves like an alphabetic to T, and is the impossible-to-disable package delimiter for Common Lisp. NISP attempts to avoid the use of "`:`" wherever possible.

In this manual, `typewriter` font is used for actual Lisp code, and *italics* are used for syntactic variables. An expression surrounded by hyphens indicates a series of expressions not enclosed in parens, as in `(COND` -*clauses*-`)`. The notation `[` a_1 | a_2 `...]` means the disjunction of the a_i. An empty disjunct may be used: `[` | a_1 | a_2 `]` means that a_1 or a_2 or neither may appear. With a single disjunct, an empty disjunct is implied, so that `[`*a*`]` means that *a* appears optionally.

1.1 NONOBVIOUS CHANGES:

For users of the old NISP, here is a brief indication of some of the more important changes which might not otherwise be obvious.

The biggest change is in the appearance of NISP code. The previous version used `:=` as a generalized setter. Alas, this identifier must be typed `\:=` in Common Lisp, which gets to be very tedious. The macro is now typed `!=`. All other colons have been eliminated as well. E.g., what used to be written `(: SL X)` is now written `(!_SL X)` or `!>X.SL`. The old forms are fully supported.

Type-declaration syntax has been changed so that type designators are preceded by hyphens, which makes it easier to extract them reliably. The old syntax is fully supported.

NISP now allows the use of `&REST` for an indefinite number of arguments, as in Common Lisp. Keyword and `&OPTIONAL` arguments are not supported, because the former are ugly, and the latter are not supported in T.

Functions such as `MEMBER` that previously used `EQUAL` as its equality test now use `EQL`, which tests for "visual equality" of atoms. Most uses of `MEMBER` were on lists of numbers anyway, so this shouldn't affect much code. The change was for Common Lisp compatibility.

Separate closures (`C\\`) are gone. Both T and Common Lisp have built-in lexical scoping, so NISP can now assume its existence.

`CHAR=` is no longer assumed identical to `EQ`.

`UNION` and `INTERSECTION` now take only two list arguments.

`/` is now defined as in T and Common Lisp.

NISP's own quasi-quote, written "!‘", is obsolete; just use the host quasi-quote, "`‘`".

The following functions are no longer part of NISP (although most are still defined to maintain compatibility with old code): `ASSQU`, `BOUNDP`, `CHR->STRING`, `CHR->SYM`, `CH=`, `CH<`, `CH>`, `CH>=`, `CH=<`, `CLOSEI`, `CLOSEO`, `CONSP`, `CMPL`, `CMPLQ`, `DE`, `DF`, `DIARECT`, `DM`, `ENTER`, `ENTQ`, `EQSTR`, `ERR-INTERCEPT`, `ERR-PASS`, `FIXP`, `FLOATP`, `LASTELEM`, `LIST-ELT-SET`, `LIST->INVISYM`, `MACRO-YIELD`, `NTHELEM`, `NTHELEM-SET`, `NTHCHR`, `NUMBERP`, `PROP-SET`, `QUARK`, `REPLAC...`, `RPLACA`, `RPLACD`, `SETPLIST`, `*SPLICE*`, `STRCONC`, `STRLEN`, `SUBSTRNG`, `*SUSP`, `SYMBOLFUN`, `SYMBOLP`, `SYM->FUN`, `TOPMAC`, <`C`.

Chapter 2

NILS — NISP Implementation Lisp Subset

NILS is written in the host Lisp dialect, and provides all of the constructs in which the rest of NISP is written.

In what follows, if a function f is marked *settable*, that means that a term `(`f `...)` stands for a storage-location-like entity whose value may be altered by writing `(SETF (`f `...) ...)` or `(!= (`f `...) ...)`. (See section 2.3.2.)

2.1 BASIC FUNCTIONS AND DATA TYPES

2.1.1 Trivial Functions

(**CR** *x*) [*Function*]

Returns x. Useful with functions which require a functional argument.

(**QUOTE** *x*) [*Magic*]

Returns *x* unevaluated. Usually input in the form `'`*x*.

(**GVAL** *e*) [*Function*]

Evaluates *e* and returns the value, using the *global* values of any variables contained in *e*.

(**PROG1** arg_1 arg_2 ... arg_n) [*Magic*]
(**PROG2** arg_1 arg_2 ... arg_n) [*Magic*]
(**PROGN** arg_1 arg_2 ... arg_n) [*Magic*]

Evaluates each argument in turn, returning the 1st, 2nd, or nth (last) argument.

2.1.2 Booleans and Predicates

There is no exclusively boolean data type in NISP.

T [*Global Variable*]
NIL [*Global Variable*]

In NISP, as in Lisp generally, there is a data object representing falsehood; every other object is taken to represent truth in contexts where a boolean is wanted.

The false object is the value of the variable `NIL`. Its "official" representation is `#F`, and this form is recognized when read. *However*, its printed representation differs between T and Common Lisp. In T, it prints as (and is `EQ` to) `()`, the empty list. In Common Lisp, it prints as (and is `EQ` to) `NIL` itself (which is also `EQ` to `()`!). NISP code should not depend on any of the three objects `#F`, `()` and `NIL` being equal or distinct.

For convenience, there is a canonical true value, whose read representation is `#T`, which is always the value of the variable T. In T, the value prints as `#T`; in Common Lisp, it prints as (and is `EQ` to) `T` itself. Again, no NISP code should depend on `T` being `EQ` to, or distinct from, `#T`. To refer to an arbitrary non-`#F` value, we will use TRUTH.

Because of the openness of the Boolean data type, it is hard to define "predicate" in NISP. "A function the falseness of whose value is often important" may be the best definition. Most predicates appear in sections specific to the characteristics they test. Those for equality and negation are more general, and are included here.

When it comes to predicate names, we eschew the querulous "?" of T (as in `SYMBOL?`) and the puerile "P" of Common Lisp (as in `SYMBOLP`) in favor of straightforward hyphenated present-tense constructions, as in `IS-SYMBOL`. There are a few predicates with names fitting none of these patterns, such as `ATOM`, `NULL` and the predicates below, where tradition has outweighed consistency.

"Equality" can mean a number of different things, and there are different predicates for different versions.

(**EQ** *x y*) [*Function*]

`#F` if *x* and *y* are "distinguishable objects" in the machine, else `#T`.

(= *x y*) [*Function*]

`#T` if and only if *x* and *y* are numerically equal numbers. That is, expect `(= 3 3.0)` to be `#T`.

(**EQL** *x y*) [*Function*]

`#T` if and only if one or more of the following is `#T`:

- `(EQ` *x y*`)`;
- *x* and *y* are the same kind of numbers and `=`; or
- *x* and *y* are characters and `CHAR=`.

(**EQUAL** *x y*) [*Function*]

`#T` if and only if one or more of the following is `#T`:

- `(EQL` *x y*`)`;
- *x* and *y* are strings with the same characters; or
- *x* and *y* are list structures whose `CAR`s and `CDR`s are `EQUAL`.

There is nothing particularly "appropriate" about `EQUAL`, although there might have been early in the history of Lisp. It might be better to define a version that recursed through any structured entity, not just lists. But neither T nor Common Lisp defines such a function; when someone wants something like that he usually can define something more specific; and `EQUAL` as defined here is occasionally useful for debugging and for elementary S-expression-hacking code.

(**NOT** *b*) [*Function*]

Returns `#T` if *b* is `#F`, else `#F`.

2.1.3 Numbers

Numbers are either floating-points or rationals. The latter are further divided into ratios and integers. An important subset of the integers are the fixed-point numbers, or "fixnums," which are implemented more efficiently than general integers, and cover most numbers that come up in practice, including all array and string subscripts. (The exact range covered by fixnums is implementation-dependent.) The ways these numbers print out is implementation-dependent, but floating-points (henceforth "floats") have decimal points, ratios have slashes (as in `3/4`), nonfixnum integers ("bignums") are long, and fixnums are not so long.

NISP follows the common-sense rule that generic functions have short names. So `+` is the name for generic addition with zero or more arguments.

(**IS-NUMBER** *x*) [*Function*]
(**IS-FLOAT** *x*) [*Function*]
(**IS-RATIONAL** *x*) [*Function*]
(**IS-RATIO** *x*) [*Function*]
(**IS-INTEGER** *x*) [*Function*]
(**IS-FIXNUM** *x*) [*Function*]

Predicates returning `#T` if *x* is a number, float, rational, ratio, integer, or fixnum respectively.

(**->INTEGER** *x*) [*Function*]
(**->FLOAT** *x*) [*Function*]

Take an arbitrary number, and return an integer or float whose value is `=` to the argument. If `->INTEGER` is given a number not `=` to an integer, it will return an integer "close" to its argument; exactly which one is implementation-dependent. See `FLOOR` and company, below.

(**IS-ODD** *x*) [*Function*]
(**IS-EVEN** *x*) [*Function*]

Test whether an integer is odd or even, returning `#T` or `#F`.

(< *x y*) [*Function*]
(> *x y*) [*Function*]
(>= *x y*) [*Function*]
(<= *x y*) [*Function*]
(=< *x y*) [*Function*]

Less than, greater than, greater than or equal, and less than or equal. Both versions of less-than-or-equal are supported, the common one and the visually appealing one.

(**MAX** arg_1 arg_2 ...) [*Function*]
(**MIN** arg_1 arg_2 ...) [*Function*]

The maximum or minimum of the arguments, of which there must be at least one.

(+ arg_1 arg_2 ...) [*Function*]

The sum of the *arg*s. `(+)` is 0.

`+` and the other arithmetic functions are generic, meaning that they take any kinds of numbers as arguments, and return a result of the simplest type possible (float if any argument is float, fixnum if all arguments are fixnum and the result is small enough, and so forth). To get non-generic arithmetic, rely on declarations, or use `FX+` and its kin, described below.

(− arg_1 arg_2 ...) [*Function*]

If given two or more arguments, − subtracts $arg_2 \ldots$ from arg_1. (− arg_1) returns the negation of arg_1.

(* arg_1 arg_2 ...) *[Function]*

The product of the *args*. (*) returns 1.

(/ x y) *[Function]*

The quotient of x and y. If x and y are integers, and y does not evenly divide x, then the result is a ratio $= x/y$.

(**FLOOR** x) *[Function]*
(**CEILING** x) *[Function]*
(**TRUNCATE** x) *[Function]*
(**ROUND** x) *[Function]*
(**FLOOR2** x y) *[Function]*
(**CEILING2** x y) *[Function]*
(**TRUNCATE2** x y) *[Function]*
(**ROUND2** x y) *[Function]*
(**QUOTIENT** i j) *[Function]*
(**REMAINDER** i j) *[Function]*
(**MOD** i j) *[Function]*

These functions are all related in that they have to do with integer division. `TRUNCATE2` takes two numbers x and y, and returns two values, x/y converted to an integer q, and a remainder r, with the property that $qy + r = x$. The integer q it picks is the integer closest to x/y that lies no further from 0 than x/y does. `FLOOR2` is similar, but picks the closest integer q no larger than x/y; `CEILING2`, the closest no smaller; and `ROUND2`, the closest integer, period. The remainder part is the simplest type possible.

The one-argument versions do the same thing, with y always taken as 1, and they return only the integer value, not the remainder.

`QUOTIENT` and `REMAINDER` operate only on integer arguments, returning the integer and remainder values that `TRUNCATE2` would return, respectively.

`MOD` operates only on integer arguments, and j must be a positive integer. It returns the second value that `FLOOR2` would return, which is always between 0 and j.

(**GCD** i j) *[Function]*

The greatest common divisor of i and j, which must be integers.

(**ABS** x) *[Function]*

Absolute value of x.

(**SIN** x) *[Function]*
(**COS** x) *[Function]*
(**TAN** x) *[Function]*
(**ASIN** x) *[Function]*
(**ACOS** x) *[Function]*
(**ATAN** x) *[Function]*
(**ATAN2** y x) *[Function]*

These are all defined as in Common Lisp. `(ATAN2` y x`)` means the same as `(ATAN` y/x`)` if x is positive; in general, `(ATAN2` y x`)` is the angle to the point x, y.

(**EXP** x) *[Function]*
(**EXPT** x y) *[Function]*
(**LOG** x) *[Function]*
(**SQRT** x) *[Function]*

Respectively, e^x, x^y, $\ln x$, $\sqrt{x}$. These are all floating point except for `EXPT`. `(EXPT` x y`)` is rational when x is rational and y is an integer, otherwise floating point unless x is negative and y is not an integer, in which case the result may be complex (undefined in Nisp).

(LOGAND *i j***)** [*Function*]
(LOGIOR *i j***)** [*Function*]
(LOGXOR *i j***)** [*Function*]
(LOGNOT *i***)** [*Function*]

Perform bitwise logical and, inclusive or, exclusive or, and not. Arguments must be integers. `LOGOR` is permissible as a synonym of `LOGIOR`.

(ASH *i d***)** [*Function*]

Arithmetic shift of i by d positions to the left ($-d$ to the right if $d < 0$).

(BIT-FIELD *i pos size***)** [*Function*]

Returns as an integer the *size* bits of integer i starting with the *pos*'th bit from the right. The rightmost bit is the zero'th. (Settable)

`(SETF (BIT-FIELD` *i pos size*`)` *new*`)` returns a new integer with the same bits changed to be the low-order *size* bits of *new*.

(FXRANDOM *ceiling***)** [*Function*]
(FLRANDOM *ceiling***)** [*Function*]

Returns a pseudo-random fixed-point or floating-point number between 0 (inclusive) and *ceiling* (exclusive).

(FX+ $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX- $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX* $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX/ $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX= $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX< $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX> $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX=< $fixnum_1$ $fixnum_2$**)** [*Function*]
(FX>= $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL+ $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL- $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL* $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL/ $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL= $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL< $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL> $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL=< $fixnum_1$ $fixnum_2$**)** [*Function*]
(FL>= $fixnum_1$ $fixnum_2$**)** [*Function*]

Specialized operators for fixnum and floating-point numbers. They take just two arguments each. These functions may be implemented as macros, in order to ensure that their arguments and results are properly declared. They are all equivalent to their generic kin (but more efficient), *except for* `FX/`, which is actually equivalent to `QUOTIENT`, not `/`.

If you use the NISP type-declaration package, it will automatically introduce these functions when required, so you don't need to use them.

2.1.4 Characters

To input a character, use #\ <*char*>, as in `#\A` or `#\a`. Some special characters have special representations:

#\SPACE [*Other*]
#\TAB [*Other*]
#\NEWLINE [*Other*]

These are the character objects for a blank space, tab and newline. During I/O, lines are delimited by a single `#\NEWLINE` character.

(**IS-CHAR** *x*) [*Function*]

Tests whether *x* is a character.

(**CHAR=** $char_1$ $char_2$) [*Function*]
(**CHAR>** $char_1$ $char_2$) [*Function*]
(**CHAR<** $char_1$ $char_2$) [*Function*]
(**CHAR>=** $char_1$ $char_2$) [*Function*]
(**CHAR=<** $char_1$ $char_2$) [*Function*]

Compare characters in the obvious ways.

(**CHAR-** $char_1$ $char_2$) [*Function*]

Subtracts numeric values of characters, yielding an integer.

(**CHAR+** *char int*) [*Function*]

Adds an integer to a character, returning a character.

(**IS-ALPHABETIC** *char*) [*Function*]
(**IS-DIGIT** *char radix*) [*Function*]
(**IS-WHITESPACE** *char*) [*Function*]
(**IS-UPPER-CASE** *char*) [*Function*]
(**IS-LOWER-CASE** *char*) [*Function*]

Type predicates for characters.

(**CHAR-UPCASE** *char*) [*Function*]
(**CHAR-DOWNCASE** *char*) [*Function*]

Coerce a character to upper or lower case.

(**CHAR->ASCII** *char*) [*Function*]
(**ASCII->CHAR** *num*) [*Function*]

Coerce characters to ASCII and back.

CHARFLOOR* [*Global Variable*]
CHARCEIL* [*Global Variable*]

Global variables are bound to one less than the smallest and one more than the greatest ASCII value.

2.1.5 Symbols and Property Lists

Symbols are manipulable objects, as well as serving as identifiers in programs.

(**IS-SYMBOL** *x*) [*Function*]

Returns `#T` iff *x* is a symbol, else `#F`.

(SYMBOL *-specs-*) [*Magic*]

Creates a symbol whose print name is built out of *specs*.

- An atomic or string *spec* is concatenated in.
- A *spec* of the form `(<` e_1 e_2 `...)` means that each e_i is to be evaluated; the values should be strings or symbols, and their characters are concatenated in.
- A spec of the form `(++` *e*`)` increments *e* and concatenates in its characters.
- Anything else is supposed to evaluate to a string, or be coercible to one; the result is concatenated in.

`SYMBOL` is usually used to create new symbols of the form `FOO1`, `FOO2`, ... To do this, you maintain a global counter `FOONUM*`, and just call

```
(SYMBOL FOO (++ FOONUM*))
```

(GENSYM) [*Function*]

Makes a new symbol, not `EQ` to any other.

Every symbol has a (possibly empty) *property list*, which can be manipulated by the functions below. Property lists are not as important in Lisp programming as they used to be, now that hash tables are available. Why write `(GET` *sym* `'`*ind*`)` when one can write `(TABLE-ENTRY` *ind-tab sym*`)`, letting a variable, *ind-tab*, play the role played by the quoted symbol *ind*, in a more controllable way? See section 2.1.12.

(GET *sym indicator*) [*Function*]
(PROP *indicator sym*) [*Function*]

If *sym* has a property value under the *indicator*, it is returned. Otherwise, `#F` is returned. (Settable)

(REMPROP *sym indicator*) [*Function*]

Makes the property attachment go away. The result is undefined.

(PLIST *sym*) [*Function*]

Returns the property list in the form (*ind val ind val* . . .) Important: Do not alter this list; use `!=` or `SETF`. (Settable, but be careful!)

2.1.6 List Structures

`CAR` and `CDR` of `()` are `()`, but it is considered bad style to depend on this fact. `CAR` and `CDR` are settable, unless their argument is `()`.

(IS-PAIR *x*) [*Function*]

`#T` iff *x* is not `()`, and has a `CAR` and `CDR`.

(ATOM *x*) [*Function*]

Same as `(NOT (IS-PAIR` *x*`))`.

(NULL *x*) [*Function*]

Same as `(EQ` *x* `'())`.

(LENGTH *list*) [*Function*]
(LEN *list*) [*Function*]
(LIST-LENGTH *list*) [*Function*]

The length of the list *list*.

(CAR *x***)** [*Function*]
(CDR *x***)** [*Function*]

Extracts the `CAR` or the `CDR`. (Settable)

(CADR *x***)** [*Function*]
(CADADR *x***)** [*Function*]

...up to the usual 4 `A`s and `D`s.

(CADD...DDR *x***)** [*Function*]

...up to 7 `D`'s.

(NTHELT *n list***)** [*Function*]
(LIST-ELT *list n***)** [*Function*]

The *n*th element (zero-based) of the list *list*. n=0 means `CAR`, and so forth. (Settable)

(NTHTAIL *n list***)** [*Function*]

The *n*th tail (`CDR`) of the list *list*. n=1 means `CDR`, and so forth.

(LASTELT *list***)** [*Function*]

The last element of *list*. (Settable.)

(LASTTAIL *list***)** [*Function*]

The last tail (`CDR`) of *list*.

(TAKE *n list***)** [*Function*]

A new list consisting of the first n elements of *list*, if n is non-negative. If n is negative, a new list consisting of the last $-n$ elements of *list*. If the magnitude of n is greater than the length of the list, the result is undefined.

(DROP *n list***)** [*Function*]

A new list consisting of the list *list* with the first n elements dropped, if n is non-negative. If n is negative, a new list consisting of the list *list* with the last $-n$ elements dropped. If n's magnitude is greater than the length of *list*, the result is undefined.

(LIST-SUBSEQ *list i j***)** [*Function*]

New list of elements starting with the *i*th and proceeding to just before the *j*th element of *list*.

(CONS *x y***)** [*Function*]

Returns a new list structure whose `CAR` is x and `CDR` is y.

(LIST arg_1 arg_2 ...**)** [*Function*]

Makes a list of the given elements.

(LIST-COPY *list***)** [*Function*]
(COPY-LIST *list***)** [*Function*]

Copies the top level of list *list*.

(COPY-TREE *list***)** [*Function*]

Copies the list structure *list* all the way down to its atoms.

(LIST-CONCAT *list*$_1$ *list*$_2$ ...) [*Function*]
(APPEND *list*$_1$ *list*$_2$ ...) [*Function*]

Makes a list whose elements are all the elements of the given lists, in order. `LIST-CONCAT` differs from `APPEND` (and resembles `VECTOR-CONCAT` and `STRING-CONCAT`) in that it copies all of its arguments, even the last.

(NCONC *list*$_1$ *list*$_2$ ...) [*Function*]

Makes the last `CDR`s of each argument but the last point to the next, thus stringing them together.

(LCONC *ptr list*) [*Function*]

Creates or modifies a "tconc pointer," a list structure whose `CDR` is the last tail of its `CAR`. If *ptr* is such a pointer, `LCONC` adds the elements of the *list* to the end of the list `(CAR` *ptr*`)`, and changes `(CDR` *ptr*`)` to be the new last tail. If *ptr* is `()`, `LCONC` returns a tconc pointer whose `CAR` is the *list*.

`LCONC` is usually used thus: A variable `P` is initialized to `'()`, and then repeatedly `(SETF P (LCONC *-*` *new-elements*`))` is executed. At the end, `(CAR P)` is the list of all the elements gathered so far. Except for the `CAR` step, this is the same as using `NCONC` instead of `LCONC`. The `LCONC` version is more efficient, because `LCONC` uses the `CDR` of `P` to find the end of `CAR P`.

(TCONC *x y*) [*Function*]

Same as `(LCONC` *x* `(LIST` *y*`))`. `TCONC` stands for "Tail concatenate."

(REVERSE *list*) [*Function*]
(DREVERSE *list*) [*Function*]

Returns a new list with the elements of *list* in reverse order. `DREVERSE` does this destructively.

(CAR-EQ *x y*) [*Function*]

Tests whether *x* is a pair whose `CAR` is `EQ` to *y*.

(IS-TAIL *list*$_1$ *list*$_2$) [*Function*]

`#T` if *list*$_1$ is a tail of *list*$_2$, that is, if *list*$_2$ is a non-`()` list, and *list*$_1$ is EQ to *list*$_2$, or is a tail of the `CDR` of *list*$_2$.

(LDIFF *list*$_1$ *list*$_2$) [*Function*]

If *list*$_2$ is a tail of *list*$_1$, returns a new list with all the tails of *list*$_1$ up to and not including *list*$_2$.

(SORT *list comparefn*) [*Function*]

Sorts *list* destructively using *comparefn*, which takes two objects and returns TRUTH if the first should come first.

(CONDENSE *x*) [*Function*]

Produces an S-expression showing a glimpse of *x* without blowing up if *x* is circular. If *x* is a list starting with `A`, we will get something like `(A --)`.

(CONSET *list x*) [*Magic*]

Same as `(SETF` *list* `(CONS` *x list*`))`, except that the *list* is evaluated only once.

(POP *list*) [*Magic*]

Same as

```
(PROG1 (CAR list)
       (SETF list (CDR list)))
```

except possibly more efficient.

(SERIES `[`*i* `1]` *n* `[`*k* `1])` *[Function]*

The list `(`$i\ \ i{+}k\ \ i{+}2k\ \ \ldots\ \ i{+}mk$`)`, where m is `(FLOOR` $n{-}i/k$`)`. With just one argument, the argument is taken to be n. With two, they are taken to be i and n. k must be > 0.

Searching and Editing S-expressions

These functions typically look through a list or tree for an object, then edit the list or tree, destructively or otherwise. They come in groups whose elements differ in how they do the search for the object. The default is to test for equality using `EQL`. Functions with names of the form "`<`*list-function*`>Q`" use `EQ` instead, while "`<`*list-function*`>=`" indicates a function whose first argument is a function to be used to test equality. Thus,

```
(MEMBER x list)  ≡  (MEMBER= #'EQL x list)
(MEMBERQx list)  ≡  (MEMBER= #'EQ  x list)
```

and similarly for the other function families. In the `REMOVE` family, the suffix `-IF` indicates that an arbitrary predicate is used to search. Functions with names of the form "`D<`*list-function*`>`" operate destructively.

(MEMBER *x list*) *[Function]*
(MEMBERQ *x list*) *[Function]*
(MEMQ *x list*) *[Function]*
(MEMBER= *eqtest x list*) *[Function]*

If *list* contains an element *y* such that *x* and *y* are equal, then returns the tail of *list* beginning with that element, else `#F`. (Tested using `EQL`, `EQ`, `EQ` and *eqtest*, respectively.)

(REMOVE1= *eqtest x list*) *[Function]*
(REMOVE-EVERY= *eqtest x list*) *[Function]*
(DREMOVE1= *eqtest x list*) *[Function]*
(DREMOVE-EVERY= *eqtest x list*) *[Function]*

Removes elements of *list* that are equal (according to *eqtest*) to *x*. If `D` is present, destroys *list*, else returns a brand-new list. If suffix `1` is present, remove first occurrence of *x*, else every occurrence.

(REMOVE1 *x list*) *[Function]*
(REMOVE-EVERY *x list*) *[Function]*
(DREMOVE1 *x list*) *[Function]*
(DREMOVE-EVERY *x list*) *[Function]*

Same as `([D]REMOVE[1|-EVERY]= #'EQL` *x list*`)`.

(REMOVE1Q *x list*) *[Function]*
(REMOVE-EVERYQ *x list*) *[Function]*
(DREMOVE1Q *x list*) *[Function]*
(DREMOVE-EVERYQ *x list*) *[Function]*

Same as `([D]REMOVE[1|-EVERY]= #'EQ` *x list*`)`.

(REMOVE1-IF *test list*) *[Function]*
(REMOVE-EVERY-IF *test list*) *[Function]*

(**DREMOVE1-IF** *test list*) [*Function*]
(**DREMOVE-EVERY-IF** *test list*) [*Function*]

Removes elements of *list* that satisfy *test*, a predicate of one argument. Otherwise same as `[D]REMOVE[1|-EVERY][|Q|=]`.

(**ADJOIN** *x list*) [*Function*]
(**ADJOINQ** *x list*) [*Function*]
(**ADJOIN=** *eqtest x list*) [*Function*]

If *list* contains an element equal to *x*, returns *list*; otherwise, returns (`CONS` *x list*).

(**ASSOC** *x list*) [*Function*]
(**ASSOCQ** *x list*) [*Function*]
(**ASSQ** *x list*) [*Function*]
(**ASSOC=** *eqtest x list*) [*Function*]

list must be a list of non-atoms. If any element of *list* has a `CAR` equal to *x*, that element is returned. Otherwise, `#F` is returned. (Tested using `EQL`, `EQ`, `EQ` and *eqtest*, respectively.)

(**UNION** $list_1$ $list_2$) [*Function*]
(**UNIONQ** $list_1$ $list_2$) [*Function*]
(**UNION=** *eqtest* $list_1$ $list_2$) [*Function*]

Makes a list whose elements are all the elements of the given lists, in no particular order, with duplicates removed. (The management is not responsible for duplicates in the original lists.)

(**INTERSECTION** $list_1$ $list_2$) [*Function*]
(**INTERSECTIONQ** $list_1$ $list_2$) [*Function*]
(**INTERSECTION=** *eqtest* $list_1$ $list_2$) [*Function*]

Returns a list each of whose elements is equal to some element in each of $list_1$ ....

(**IS-SUBLIST** $list_1$ $list_2$) [*Function*]
(**IS-SUBLISTQ** $list_1$ $list_2$) [*Function*]
(**IS-SUBLIST=** *eqtest* $list_1$ $list_2$) [*Function*]

`#T` iff every element of $list_1$ is equal to some element of $list_2$.

(**COMPLEMENT** $list_1$ $list_2$) [*Function*]
(**COMPLEMENTQ** $list_1$ $list_2$) [*Function*]
(**COMPLEMENT=** *eqtest* $list_1$ $list_2$) [*Function*]

A new list consisting of all the elements of $list_1$ not equal to any element of $list_2$.

(**NODUP** *list*) [*Function*]
(**NODUPQ** *list*) [*Function*]
(**NODUP=** *eqtest list*) [*Function*]

Makes a new list with duplicates removed.

(**DNODUP** *list*) [*Function*]
(**DNODUPQ** *list*) [*Function*]
(**DNODUP=** *eqtest list*) [*Function*]

Destructively removes duplicates from the *list*.

(**SUBST** *new-elem old-elem tree*) [*Function*]
(**SUBSTQ** *new-elem old-elem tree*) [*Function*]
(**SUBST=** *eqtest new-elem old-elem tree*) [*Function*]

Returns a copy of *tree* with all occurrences of *old-elem* replaced by *new-elem*.

2.1.7 Backquote

Backquote is what philosophers call “quasi-quote.”

‘*exp* [*Magic*]

‘*exp* evaluates to *exp*, with parts marked by “`,`” and “`,@`” substituted with their values. For example:

`‘(A B C)`	is the same as `’(A B C)`
`‘(A ,X C)`	is the same as `(LIST ’A X ’C)`
`‘(A B . ,X)`	is the same as `(CONS ’A (CONS ’B X))`
`‘(A ,@X C)`	is the same as `(CONS ’A (APPEND X ’(C)))`

In general, things marked with comma will be evaluated; and things marked with comma-atsign will be evaluated, and the resulting list spliced into the result. The commas and comma-atsigns may appear at any level, sparing you complex analysis of what gets evaluated when. This is especially useful when creating s-expressions to be evaluated later, as macros do.

If there are no marked subparts, “‘” behaves like “’”. (In most implementations, it will literally quote the following expression, but don’t depend on this.)

If a backquote appears within a backquote (a so-called “nested” backquote), then it is possible, with sufficient ingenuity, to work out the meaning of the resulting expression. However, there is seldom any good reason to inflict this puzzle on the reader of your code. If you find yourself wanting to nest backquotes, lie down until sanity returns, and use `LIST` instead.

Backquote notation should be used to construct S-expressions, not lists to be used as records (i.e., to be altered). This is because ‘ may try to quote as much of the expression as possible, so it is hard to tell exactly how much structure will be shared between two evaluations of the same ‘ expression.

(**INCLUDE-IF** *test exp*) [*Function*]

Useful inside a backquote. `,@(INCLUDE-IF` *test exp*`)` behaves like `,`*exp* if *test* is true, else as if it weren’t there.

2.1.8 Vectors and Arrays

Vectors are just 1-dimensional arrays.

(**MAKE-VECTOR** *size*) [*Function*]

Makes a vector of this size, with undefined elements.

(**VECTOR** *-elements-*) [*Function*]

Makes a vector with these elements.

(**IS-VECTOR** *x*) [*Function*]

Tests if *x* is a vector.

(**VECTOR-ELT** *vec n*) [*Function*]
(**VREF** *vec n*) [*Function*]

Returns the *n*th element of *vec* (zero-based). (Settable)

(**VECTOR-LENGTH** *vec*) [*Function*]

The number of elements in the argument.

(**VECTOR-COPY** *vec*) [*Function*]

A new vector with the same elements as the argument.

(**VECTOR-SUBSEQ** *vec i j*) [*Function*]

Returns a new vector starting with *i*th element and proceeding to just before *j*th (zero-based).

(**VECTOR-CONCAT** v_1 v_2 ... v_n) [*Function*]

A new vector consisting of the elements of v_1, the elements of v_2, etc., in that order.

(**VECTOR->LIST** *vector*) [*Function*]
(**LIST->VECTOR** *list*) [*Function*]

Convert a vector to a list with the same elements, or vice versa.

(**MAKE-ARRAY** *dimension-list*) [*Function*]
(**INITIALIZED-ARRAY** *dimension-list initial-element*) [*Function*]

Make an array with as many dimensions as the length of the *dimension-list*, where the *i*th dimension is given by the *i*th element of that list. `MAKE-ARRAY` leaves the elements undefined; `INITIALIZED-ARRAY` initializes them to the given element.

(**IS-ARRAY** *x*) [*Function*]

Test whether *x* is an array. Note that vectors are arrays.

(**AREF** *array* i_1 ... i_n) [*Function*]

Element of *array* specified by subscripts i_1 ... i_n. (Settable)

(**ARRAY-DIMENSIONS** *array*) [*Function*]

A list of the dimensions.

(**ARRAY-DIMENSION** *array dim*) [*Function*]

The *dim*th element of that list.

2.1.9 Strings

Strings are vectors of characters, but the host dialect may not allow vector operations on them. They print using delimited double quotes.

Use `EQUAL` to compare strings.

(**IS-STRING** *x*) [*Function*]

Returns `#T` if *x* is a string.

(**STRING-COPY** *string*) [*Function*]
(**STRING-CONCAT** $string_1$ ... $string_n$) [*Function*]

Return a new string, containing the same characters as — but not sharing any structure with — the original string(s).

(**STRING-SUBSEQ** *string i j*) [*Function*]

Returns a new string starting with the *i*th character and proceeding to just before *j*th (zero-based).

(**STRING-ELT** *string i*) [*Function*]

The character at pos *i* of *string*, zero-based.

(**STRING-LENGTH** *string*) [*Function*]

The length of (number of characters in) *string*.

(STRING-UPCASE *string*) [*Function*]
(STRING-DOWNCASE *string*) [*Function*]

Coerces *string* to upper/lower case, and returns the new string.

2.1.10 Sequences

There are no generic sequences in NISP. For *seq*=`LIST`, `STRING`, or `VECTOR`, we have *seq*-`COPY`, *seq*-`LENGTH`, *seq*-`SUBSEQ`, *seq*-`ELT`, and *seq*-`CONCAT`.

2.1.11 Character coercions

You can go back and forth between symbols, characters, lists of characters, and strings, by using the functions below.

(SYMBOL->LIST *symbol*) [*Function*]

Returns a list of the characters in the print name of *symbol*.

(SYMBOL->STRING *symbol*) [*Function*]

Returns a string of those characters.

(STRING->LIST *string*) [*Function*]

Returns a list of the characters in *string*.

(STRING->SYMBOL *string*) [*Function*]

Returns a symbol whose print name is the string *string*.

(LIST->SYMBOL *list*) [*Function*]

Returns a symbol whose print name consists of the characters in the list *list*.

(LIST->STRING *list*) [*Function*]

Returns a string of those characters.

`SYMBOL->LIST` and `SYMBOL->STRING` are guaranteed to work on numbers, but `LIST->SYMBOL` and `STRING->SYMBOL` never produce numbers.

(NUMBER->STRING *n*) [*Function*]

Produces a string from a number.

(STRING->NUMBER *s*) [*Function*]

Produces a number from a string that looks like a number.

(CHAR->STRING *char*) [*Function*]
(CHAR->SYMBOL *char*) [*Function*]

Returns a string with the single character *char*, or a symbol with such a string as its name.

2.1.12 Hash Tables

Hash tables provide an efficient mechanism for associating *key*s with *value*s.

(MAKE-EQ-HASH-TABLE) [*Function*]

Makes a hash table whose entries are keyed on subscripts that are `EQ`-tested.

(**TABLE-ENTRY** *hashtable key*) [*Function*]

Returns entry associated with *key* in *hashtable*. (Settable)

(**IS-HASH-TABLE** *x*) [*Function*]

Tests whether *x* is a hash table.

(**WALK-TABLE** *fn tab*) [*Function*]

Applies function *fn*, of two arguments, to every *key,value* pair in hash table *tab*.

(**FRESH-TABLE** *tab*) [*Function*]

Returns a new table obtained by clearing hash table *tab* (if supported) else by building a brand-new hash table.

2.1.13 Mappers

Mappers are functions that transform a list by applying a function to each of its elements or tails, then combining the results, often into a new list. The most common mapper is the one (traditionally called `MAPCAR`) that conses the returned values into a list. In NILS, it is called `MAPELTLIST`.

For every mapper that applies to elements of a list, there is a version that applies to the tails. The former has `ELT` as the middle of its name; the latter, `TAIL`. So `MAPTAILLIST` is a function that applies a function to every tail of a list, and makes a list of the results. Note that in this context the list counts as a tail of itself, and the final `()` at the end does not.

Most of the mappers take any number of list arguments. The single functional argument must be able to handle as many arguments as there are lists. For example, `(MAPELTLIST LIST '(A B C) '(1 2)) => ((A 1) (B 2))`. The lists don't have to be the same length; the mapper stops when one list runs out. In what follows, I use the term "cross section" of the argument lists to refer to a collection of elements (the N'th of each list) to which the function is applied. The cross sections of `(A B C)` and `(1 2)` are `A,1` and `B,2`. In the tail versions, the cross sections are groups of tails rather than elements.

Here are all the mappers, plus some non-mappers that seem to belong here. These funoids behave as functions, but may for efficiency be implemented as macros, a fact irrelevant in almost all situations.

(**MAPELTLIST** *fun -lists-*) [*Function*]
(**MAPTAILLIST** *fun -lists-*) [*Function*]

Applies *fun* to each cross section of the argument *lists*, and make a list of the results. (Traditional names: `MAPCAR`, `MAPLIST`.)

(**MAPELTAPPEND** *fun -lists-*) [*Function*]
(**MAPTAILAPPEND** *fun -lists-*) [*Function*]

Applies *fun* to each cross section of the argument *lists*, and `APPEND` the results.

(**MAPELTCONC** *fun -lists-*) [*Function*]
(**MAPTAILCONC** *fun -lists-*) [*Function*]

Apply *fun* to each cross section of the argument *lists*, and `NCONC` the results. (Traditional names: `MAPCAN`, `MAPCON`.)

(**MAPELTAND** *pred -lists-*) [*Function*]
(**MAPTAILAND** *pred -lists-*) [*Function*]

Applies *pred* to successive cross sections of the argument *lists*, and returns `#F` if the predicate ever returns `#F`. If the end of one list is reached, returns TRUTH. (Traditional name: `EVERY`)

(**MAPELTOR** *pred -lists-*) [*Function*]
(**MAPTAILOR** *pred -lists-*) [*Function*]

Applies *pred* to successive cross sections of the argument *lists*, and returns the remaining tails of all the list arguments as soon as the predicate returns TRUTH. (That is, if there is just one *list*, its tail is returned, else the tails of all the lists are returned as multiple values. If the end of a list is reached without satisfying the predicate, return `#F`, or, more precisely, as many `#F`'s as there are list arguments.

(**MAPELTSOME** *pred -lists-*) [*Function*]
(**MAPTAILSOME** *pred -lists-*) [*Function*]

Like `MAP...OR`, but returns the value of the *pred* rather than the tails of the *lists*. (Traditional name: `SOME`)

(**MAPELTCOLLECT** *pred -lists-*) [*Function*]
(**MAPTAILCOLLECT** *pred -lists-*) [*Function*]

Makes a list of the elements of the last argument list for which *pred* returns TRUTH on the corresponding cross section. (Traditional name: `SUBSET`)

(**MAPELTDO** *fun -lists-*) [*Function*]
(**MAPTAILDO** *fun -lists-*) [*Function*]

Applies the function to successive cross sections of the argument lists, discarding the results. The result of `MAP...DO` is undefined. (Traditional names: `MAPC`, `MAP`)

(**MAPELTREDUCE** *fun ident -lists-*) [*Function*]
(**MAPTAILREDUCE** *fun ident -lists-*) [*Function*]

If any list argument is `()`, returns *ident*. Otherwise, it replaces *ident* with the value of *fun* applied to *ident* and the first cross section of the *lists*, replaces the lists with their `CDR`s, and repeats. That is,

```
(MAPELTREDUCE fun ident
              '(e11 e12 ... e1K)
              '(e21 e22 ... e2K)
              ...
              '(eN1 eN2 ... eNK))
=>
(fun ... (fun (fun ident e11 e21 ... eN1)
                  e12 e22 ... eN2)
     ...
             e1K e2K ... eNK)
```

2.2 FUNCTIONS, MACROS, ETC.

The word "function" is often used fairly loosely in the Lisp literature to mean something that can come after a left paren. In this manual, we will be careful to reserve that term for an entity that is passed argument values and returns a result (actually, zero or more results). We will use the term *magic word* for any other kind of callable entity, such as `COND` or `QUOTE`. Magic words do not necessarily take "arguments" as such; they can be used to extend the syntax of the language in arbitrary ways. Some magic words, like `IF`, can be considered to take arguments, but may not evaluate all of them. Magic words are either defined as source-level transformations, in which case they are called *macros*; or in some other mysterious way, known only to the implementors, in which case they are called *special forms*. Users can define their own macros.

The word "funoid" will be used to mean "function or magic word."

2.2.1 Defining and Manipulating functions

There are three entities associated with a function:

1. Its name (optional): E.g. `BAZ`
2. Its definition: E.g. `(LAMBDA (X) (LIST X X))`
3. Its procedure: E.g. `#{Procedure BAZ}`

The procedure is the "function value" of its name, which in T is just the ordinary value, and in Common Lisp is something else. NISP code should never depend on an identifier's having distinct symbol and function values.

Named functions are created using `DEFUN`:

(DEFUN *name* (*-args-* [&REST *var*]) *-body-*) [*Magic*]

The primitive definer of functions. If the function takes an indefinite number of arguments, this is indicated by ending the *-args-* with `&REST` *var*, and *var* will be bound to a list of all the remaining arguments.

Functions may be defined without being named. An anonymous function is written

(LAMBDA (*-vars-* [&REST *var*]) *-body-*) [*Other*]

Any free variables in the *body* get their bindings from the current lexical environment.

Warning: `LAMBDA` is neither a function nor a magic word. A `LAMBDA`-expression denotes a function in two contexts: when it appears in functional position, and when it appears in an expression of the form `([FUNCTION | FUNKTION] (LAMBDA ...))`. The optional `&REST` clause works as in `DEFUN`, but cannot be used with `(LAMBDA ...` in functional position (due to incompatibility with T). `FUNDEF->LAMBDA` (see p. 23) provides a way around this problem.

Functions appearing anywhere except functional position must be quoted:

(\\ (*-vars-*) *-body-*) [*Magic*]

Is the same as

(FUNCTION (LAMBDA (*-vars-*) *-body-*)).

It evaluates to an anonymous procedure with the given formal arguments. You cannot use the `\\` notation in functional position.

(FUNCTION [*name* | *lambda-exp*]) [*Magic*]

This form may be abbreviated #'[*name* | *lambda-exp*] What follows is either a symbol-name with a function-value, or a lambda expression:

(FUNCTION *name*)
#'*name*
(FUNCTION (LAMBDA (*-vars-*) *-body-*))
#'(LAMBDA (*-vars-*) *-body-*)

In T, (FUNCTION *x*) is the same as *x*.

(FUNKTION [*name* | *lambda-exp*]) [*Magic*]

This form may be abbreviated !'[*name* | *lambda-exp*]. With an atomic argument, `FUNKTION` evaluates to an object that is always the current procedure bound globally to *name*, even if the name has been redefined.

(FUNKTION *name*)
!'*name*
(FUNKTION (LAMBDA (*-vars-*) *-body-*))
!'(LAMBDA (*-vars-*) *-body-*)

`FUNCTION` and `FUNKTION` are magic. They do not evaluate their argument. They may be thought of as a sort of `QUOTE` for globally defined named functions. In compiled code, `(FUNKTION symbol)` means the same as `(FUNCTION symbol)`.

`(`**APPLY** *fun -args- list*`)` `[`*Function*`]`

Not a mapper. Calls the given function with the *args* followed by the elements of *list* as arguments.

`(`**FUNCALL** *fn -args-*`)` `[`*Function*`]`

Is the only way to call an evaluated procedure, that is, a function returned as the value of the expression *fn*.

`(`**IGNORE** *-vars-*`)` `[`*Magic*`]`

Is used in both named and anonymous functions to tell the compiler not to worry about an unused argument. The `IGNORE` form should be put at the front of the *body*.

2.2.2 Defining Macros

Macros are defined using `DEFMACRO`:

`(`**DEFMACRO** *name* `(`*-vars-* `[&REST` *var*`])` *-body-*`)` `[`*Magic*`]`

Associates a macro definition with *name*. Uses `&REST` notation analogously to `DEFUN`. Alternatively, you can just write "`.`" instead of `&REST`. A macro makes itself felt thus: whenever `(`*name -stuff-*`)` is seen in an evaluable position, it is transformed by binding the *args* to the corresponding pieces of *stuff*, evaluating the expressions in *body*, and letting the value of the last be the transformed code.

2.2.3 Manipulating Funoids

Here are some functions for manipulating funoids and their names.

`(`**IS-FUN-NAME** *sym*`)` `[`*Function*`]`

Tests whether *sym* is the name of a globally defined named function or magic word. It returns `#F` iff it is not.

`(`**SYMBOL->FUN** *sym*`)` `[`*Function*`]`

Returns the procedure corresponding to *sym*. Returns `()` if *sym* has no function definition (or if the *sym* is magic). `SYMBOL->FUN` does not get the value of *sym*, but its global function value, which may be different.

`(`**GET-FUNDEF** *sym*`)` `[`*Function*`]`

Gets the funoid definition of *sym*, or its procedure if the definition is unavailable. (Settable, or use `PUT-FUNDEF`.) GET-FUNDEF returns `()` if and only if the *sym* has no definition. If the text of the definition is available, GET-FUNDEF returns one of the following:

- `(LAMBDA` *args -body-*`)`: If the symbol names a function.
- `(MACRO` *args -body-*`)`: If the symbol names a macro.
- `(NLAMBDA (`*arg*`)` *-body-*`)`: If the symbol is an interpreted magic word (a "fexpr"). In some implementations, there aren't any of these.

These forms have the property that replacing `LAMBDA`, `MACRO`, or `NLAMBDA` respectively with "`DEFUN` *name*," "`DEFMACRO` *name*," or "`DF` *name*" will produce a correct definition, which, if evaluated, would make the definition of *name* be the same as that of *sym*.

If the value returned by `GET-FUNDEF` is none of these three or `()`, then its meaning is implementation-dependent. If such a thing is printed, it probably cannot be read back in, even to the same LISP dialect.

(PUT-FUNDEF *sym fundef*) [*Function*]

Where *fundef* is as returned by `GET-FUNDEF`, defines *sym* to be *fundef*. This is guaranteed to work when *fundef* is any value of `GET-FUNDEF`, including `()`.

(FUNDEF->FUN *fundef*) [*Function*]

Returns a function with the given definition. Works only if fundef is of the form `(LAMBDA ...)`.

(FUNDEF->LAMBDA *fundef*) [*Function*]

Not all expressions of the form `(LAMBDA ...)` can occur in functional position, because `&REST` is illegal in T. This function converts a legal NISP `(LAMBDA ...)` definition into something that can occur in functional position.

(FUN-NAME *funoid*) [*Function*]

Returns the symbol-name of the funoid if it can find one, else `()`.

(IS-MACRO *x*) [*Function*]

Tests if *x* is a symbol defined as a macro.

(IS-MAGIC *x*) [*Function*]

Tests if *x* is magic, i.e., is a callable something that does not expect its arguments to be evaluated and passed to it exactly once. (Macros count as magic.) *x* may be either a symbol or a value returned by `GET-FUNDEF`

(ONE-MACRO-EXPAND *exp*) [*Function*]
(MACRO-EXPAND-EXP *exp*) [*Function*]

If *exp* is a form beginning with a symbol having a macro definition, `ONE-MACRO-EXPAND` expands the macro call once and returns the result; otherwise, *exp* is returned unchanged. `ONE-MACRO-EXPAND` expands the call repeatedly until the form no longer begins with a macro.

2.3 CONTROL STRUCTURES

2.3.1 Binding Variables

We assume lexical scoping. It may be overridden using `BIND`.

(LET ((*var val*) (*var val*) ...) *-body-*) [*Magic*]

Binds variables lexically, then evaluates each expression in the body, returning the last value. To leave a variable uninitialized, just say *var* instead of (*var val*). Well ..., you can't really leave it uninitialized, but not giving it an explicit initial value indicates that its initial value is unimportant.

(BIND ((*var val*) (*var val*) ...) *-body-*) [*Magic*]

Like `LET`, but binds dynamically. `BIND` cannot be used to bind an unbound variable. It must be previously `DEFVAR`'ed first.

A variable bound dynamically is traditionally said to be "special," and Common Lisp upholds this tradition. Such a variable can be accessed in a piece of code where it is unbound, provided it is declared special. In the file where the variable is `DEFVAR`'ed, such a declaration happens automatically. In any other file, you must write `(PROCLAIM '(SPECIAL` *-special-vars-*`))` before the first binding or use of the variable.

(DEFVAR *symbol exp*) [*Magic*]

Used only globally. Proclaims variable special; initializes its value, but if executed again may leave value undisturbed (in Common Lisp; in T this behavior is impossible to obtain, so don't count on it).

(FLABELS (*-local-function-defs-*) *-body-*) [*Magic*]

Each *local-function-def* is of the form

(*name* (*-args-*) *-body-*)

Each *name* is locally defined as a function in the obvious way, and the *body* is executed with those definitions in effect. (Note that the local functions are called without using `FUNCALL`.)

(PROG (*-vars-*) *-tags-and-statements-*) [*Magic*]

Binds the *vars*, then execute the *statements*. If an expression of the form `(GO` *tag*`)` is executed anywhere in the lexical scope of the `PROG`, control will jump to the statement following that *tag*.

2.3.2 Side Effects

(SETQ *var value*) [*Magic*]

Sets an already-bound variable.

(GSET *symbol value*) [*Function*]

Sets the global value of the *symbol* to the *value*. That is, if the value of `X` is `Y*`, `(GSET X 5)` sets `Y*` to be `5`.

(SETF *exp value*) [*Magic*]

Sets a settable expression. If *exp* is a macro call, not itself settable, then it is expanded and `SETF` tries again to make sense of it.

Important: `SETQ` and `SETF` are to be executed purely for effect, and return no reliable value.

(DEFSETF *accessor setter*) [*Magic*]

Tells NISP to transform expressions of the form `(SETF (`*accessor* `...) ...)` into `(`*setter* `......)`. The *setter* should take one more argument than the accessor. E.g., if `RPLACA` were the name of a function to alter the `CAR` of a dot-pair, we could write `(DEFSETF CAR RPLACA)` to tell NISP to treat `(SETF (CAR` *x*`)` *y*`)` as `(RPLACA` *x y*`)`. Note that `DEFSETF` is followed by two unevaluated funoid names; the process is like macro definition, not like "telling an accessor what its setter is," or something fancy like that.

2.3.3 Conditionals

(COND ($test_1$ -$body_1$-) ($test_2$ -$body_2$-) ... ($test_n$ -$body_n$-)) [*Magic*]

Evaluates each test $test_i$ until one is TRUTH. The corresponding *body* is then evaluated, and the last value is returned. The value is undefined if no test comes out TRUTH. Empty bodies are not allowed.

(IF *test true-exp* [*false-exp*]) [*Magic*]

Alternative form of conditional for faddists who have tired of `COND`.

(AND e_1 e_2 $\ldots e_n$) [*Magic*]

`(AND)` is equivalent to `#T`.
`(AND` *e*`)` is equivalent to *e*.
`(AND` e_1 e_2 $\ldots e_n$`)` is equivalent to

```
(COND (e1
        (AND e2 ... en))
      (T '#F)  )
```

(OR e_1 e_2 $\ldots e_n$) [*Magic*]

`(OR)` is equivalent to '`#F`.

`(OR` $e_1 \ldots$ e_n`)` is equivalent to

```
(LET ((v1 e1))
   (COND (v1 v1)
         (T (OR e2 ... en))  ))
```

(SELQ *exp* -((-*vals*-) -*body*-)- [(T -*default-body*-)]) [*Magic*]

Evaluates *exp*, then evaluates the *body* whose *vals* contain an element `EQ` to the value of exp. If just one *val*, you can omit the parens. If no such body is found, then the *default-body* is evaluated instead. If there is no default and no clause whose *vals* contain an element `EQ` to the value of *exp*, the value of the `SELQ` is undefined.

2.3.4 Loops

(DO (-*var-bindings*-) (*test* -*result-body*-) -*body*-) [*Magic*]

Where the *var-bindings* are each of the form (*var init next*). Binds the variables to their initial values, does the *test*, executes *body*, bumps the variables, does the *test*, and so forth, until the *test* comes up TRUTH, when it evaluates the *result-body* and returns the value of the last expression in it. "Bumping the variables" means evaluating each *next* expression and setting the corresponding variable to it, all in parallel.

(LOOP [FOR (-*variable-specs*-)] -*statements*-) [*Magic*]

variable-spec forms are:

symbol

(*sym init-val* [*bump*])

(*sym* IN *list*)

(*sym* = *init* [BY *incr*] [TO *final*])

statement forms are:

`WHILE` *test*

`UNTIL` *test*

`RESULT [IS]` *value*

action

Meaningless keywords such as `REPEAT` can be sprinkled anywhere in a `LOOP`, and they will be ignored.

Semantics: The variables are initialized, and statements are executed. If a test indicates termination, then the next `RESULT` expression is evaluated and returned as the value of the loop. The default value is `#F`. At the end of the statements, if no test has succeeded, the variables are bumped and the statements are re-evaluated.

Bumping a variable takes place one of 3 ways:

1. If it was bound as `(`*var init new*`)`, then *var* is set to the value of *new* at the end of the loop.
2. If it was bound as `(`*var* `IN` *list*`)`, then *var* is set to the next element of the list.
3. If it was bound as `(`*var* `=` *init* `[TO` *final*`] [BY` *incr*`]))`, then the obvious Algolish thing happens.
   - If both *incr* and *final* are omitted, *var* is only initialized.
   - If only *incr* omitted, defaults to 1.
   - If only *final* is omitted, *var* is incremented but not tested.

   WARNING: Positive increment assumed unless *incr* is negative constant!!

2.3.5 Mapping Loops

(FOR *-var-clauses-* `[(WHEN` *test*`)] ([SAVE|SPLICE|FILTER]` *exp*`))` [*Magic*]

Where each *var-clause* is of the form `(`*var* `IN` *list*`)`, is equivalent to a `MAPELTLIST`, `MAPELTCOLLECT`, or `MAPELTCONC` in the following way:

- Let *vars* be the variables from the *var-clauses*, and *lists* be the lists.
- If `WHEN` is present, then pretend *lists* consists of just the elements that pass the *test*.
- Now the meaning of `FOR` depends on whether the last keyword is `SAVE`, `SPLICE`, or `FILTER`:
- `SAVE`: Collect values of *exp* for each binding of the *vars* to elements of *lists*. Make a new list of them.
- `SPLICE`: Ditto, but destructively splice them.
- `FILTER`: As with `SAVE`, but `#F` values are discarded.

The `FOR` macro is due to Chris Riesbeck.

(FORALL *-var-clauses- test*`)` [*Magic*]

Where the *var-clauses* have the same syntax as for `FOR`. Equivalent to

`(MAPELTAND (\\ (`*-vars-*`)` *test*`)` *-lists-*`)`

where the *vars* and *lists* are as defined in the definition of `FOR`.

(EXISTS *-var-clauses- test*`)` [*Magic*]

Where the *var-clauses* have the same syntax as for `FOR`. Equivalent to

`(MAPELTOR (\\ (`*-vars-*`)` *test*`)` *-lists-*`)`

where the *vars* and *lists* are as defined in the definition of `FOR`.

2.3.6 Nonlocal Jumps

(INTERCEPT *label -body-*`)` [*Magic*]

Evaluate *body*. Return last thing. A `PASS` to the *label* aborts execution. The label is not a variable; it can be used only by appearing in a `PASS` within the *dynamic* scope of the `INTERCEPT`.

(**PASS** *label value*) [*Magic*]

value is evaluated. Surrounding `INTERCEPT` with that *label* is exited with that value. The label is not evaluated in `INTERCEPT` and `PASS`.

(**UNWIND-PROTECT** *exp -cleanup-exps-*) [*Magic*]

Evaluates *exp*, then evaluates all *cleanup-exps* even if there is a nonlocal exit (i.e., a `PASS`, or error abort) out of *exp*.

2.3.7 Multiple Values

Functions can return more than one value by making sure that the last thing they evaluate is an expression of the form (`VALUES` v_1 ...v_n). When this occurs, whoever called the function must be expecting as many values as were returned. Unlike Common Lisp, NISP does not conveniently discard extra values. You can use `ONE-VALUE` to do this.

(**VALUES** a_1 ... a_n) [*Function*]

Basic multiple-value construct.

(**ONE-VALUE** *x*) [*Function*]

The first value returned by *x*. It is an error to use this form if *x* returns zero values.

(**LIST->VALUES** *list*) [*Function*]

Converts list to multiple values.

(**MULTIPLE-VALUE-LIST** *form*) [*Magic*]

Makes a list of the values returned by *form*

(**MULTIPLE-VALUE-LET** (*-vars-*) *form -body-*) [*Magic*]

Binds the *vars* to the values returned from *form*. There must be exactly as many values as variables.

(**MULTIPLE-VALUE-CALL** *receiver form*) [*Magic*]

Evaluates *form* and call function *receiver* with the values returned. `MULTIPLE-VALUE-LIST` could be defined as `(MULTIPLE-VALUE-CALL #'LIST ...)`

(**MULTIPLE-VALUE-SETQ** (*-vars-*) *form*) [*Magic*]

Evaluates *form* and sets the variables to the values returned. There must be the same number of *vars* as values. The value of the whole expression is undefined.

2.3.8 Data-Driven Programming

Data-driven programming is a simple kind of object-oriented programming, in which the objects are S-expressions operated on by procedures whose behavior depends on symbols found in the `CAR`s of those S-expressions. This kind of programming is ubiquitous in Lisp systems programs, such as pretty-printers, but is also common in non-systems programs, such as syntax checkers for predicate calculus. Rather than write such procedures as large `COND`s that check for every expected symbol, we associate a separate small procedure with every symbol, and have our master procedure look for it. The following macro makes that chore easy.

(**DATAFUN** *indicator symbol what-to-do*) [*Magic*]

Associates *what-to-do* with the *symbol* under the *indicator*, typically by storing it on the property list of *symbol*. *what-to-do* is one of three things:

1. An expression of the form `!'`*function*: in which case, that function will be associated with the symbol.
2. A symbol: in which case the action associated with *symbol* is to be the same as the action already associated under *indicator* with this symbol.
3. A function definition with the name elided: in which case a function named *"symbol-indicator"* will be defined and associated with this symbol.

The default method of association is to put *what-to-do* on the property list of *symbol* under *indicator*. A function can then execute `(PROP '`*indicator* `S)` to find the action associated with `S`. A typical master procedure might look like

```
(DEFUN MASSAGE (X)
   (COND ((OR (ATOM X) (NOT (IS-SYMBOL (CAR X))))
          ;; What to do in unusual or base case
          ...)
         (T
          (LET ((FN (PROP 'MASSAGE (CAR X))))
             (COND (FN
                    ;; Found function — call it.
                    (FUNCALL FN X ...))
                   (T
                    ;; Default behavior
                    ...)   )))   ))
```

And a typical call to `DATAFUN` would look like:

```
(DATAFUN MASSAGE OR
   (DEFUN (X ...)
      -code-for-massaging-things-beginning-with-OR-))
```

which would define a function named `OR-MASSAGE` to massage expressions of the form `(OR ...)`.

To override the convention that functions are stored in the property lists of the symbols they are associated with, you must tell NISP how to attach functions to symbols. Put on the property list of *indicator*, under the property `ATTACH-DATAFUN`, a function of three arguments, say `IND`, `SYM`, and `FNAME`, that associates the function named `FNAME` with `SYM` under `IND`. The simplest way to make this "meta-association" is by writing

```
(DATAFUN ATTACH-DATAFUN indicator
   (DEFUN (IND SYM FNAME)
         ;; IND is presumably just indicator again
         Attach (SYMBOL->FUN FNAME) to SYM in appropriate way
))
```

Now that property lists are out of fashion, you may want to use a hash table to associate symbols with procedures. Use

(**DATAFUN-TABLE** *table-name indicator*) [*Magic*]

If you write `(DATAFUN-TABLE M-TAB* MASSAGE)`, then `(DATAFUN MASSAGE` *sym* `...)` will store the function (named *sym*-`MASSAGE`) as `(TABLE-ENTRY M-TAB* '`*sym*`)`, for your code to retrieve.

2.4 INPUT/OUTPUT

I/O is based on entities called *streams* that yield or absorb characters and larger objects. The ones that yield things are called *input streams*; the ones that absorb things are called *output* *streams*.

2.4.1 Reading and Printing Conventions

Lists read and print in the standard way.

The escape character is *backslash*. Modern LISP dialects have supplanted the slash character somewhat for symbols with more than one funny character in their names. An arbitrary string of characters may be made into a symbol by enclosing it in vertical bars (|...|). To put a slash or vertical bar into such a symbol's name, slashify it. So `|A \\c \||` is the way to write a symbol whose print name has six characters: A, space, backslash, lower-case C, space, vertical bar. Some dialects will print this symbol this way, others as `A\ \\\c\ \|` or `#[Symbol "A \c |"]`.

Comment character: `;` — Everything from here to the end of the line will be treated as one whitespace.

The macro character `#` is reserved to the host dialect. Vectors are read and printed as `#(`-*elements*-`)`. We also have

- `#\` character
- `#'` `FUNCTION` abbreviation
- `#+` Common Lisp read-time conditionalization (ignored by T)
- `#-` Common Lisp read-time conditionalization (ignored by T)

The macro characters `!` and `?` are reserved to NISP.

- `!'` `FUNKTION` abbreviation
- `!_` Slot access
- `!>` Slot access
- `!D` Dialect (host Lisp dialect) specific
- `!S` System (host operating system) specific
- `!@` Piece of a match pattern to be evaluated (see below)
- `?` Match variables

The macro characters `#`, `?` and `!` are inactive when they appear inside identifiers.

(READMAC *char fun***)** [*Function*]

Attaches *fun* to *char* so that when *char* is seen as the initial character of a read expression, *fun* is called and its value counts as the object read. *fun* takes one argument, a stream. So quote might have been defined by

```
(READMAC #\' (\\ (S) `(QUOTE ,(SRMREAD S))  ))
```

2.4.2 Streams

Standard streams: standard input, standard output, and error output. These are given when the process is started, but may be rebound by NISP code. Of course, they are usually all bound to the terminal-io stream.

(STDIN) [*Function*]
(STDOUT) [*Function*]
(ERROUT) [*Function*]

Return these three streams. (Settable)

(STDIN-SET *s***)** [*Function*]
(STDOUT-SET *s***)** [*Function*]
(ERROUT-SET *s***)** [*Function*]

Reset them. (Or just use `SETF`.)

(**REBIND-STDIN** *s -body-*) [*Magic*]
(**REBIND-STDOUT** *s -body-*) [*Magic*]
(**REBIND-ERROUT** *s -body-*) [*Magic*]

Rebind a standard stream during a body.

TTYIN* [*Global Variable*]
TTYOUT* [*Global Variable*]

Global variables bound to terminal input and output (hopefully). These should never change.

(**OPENO** *filename*) [*Function*]
(**OPENI** *filename*) [*Function*]

Creates an input or output stream. The *filename* may be a pathname or something coercible to a pathname.

(**CLOSE** *stream*) [*Function*]

Closes a stream. Streams do not close by themselves, so you probably want to use the following two magic words instead of the explicit openers and closers.

(**WITH-INPUT-FROM-FILE** *sym filename -body-*) [*Magic*]

Binds *sym* to an input stream from the given file, executes *body* (returning its last element) and closes the stream.

(**WITH-OUTPUT-TO-FILE** *sym filename -body-*) [*Magic*]

Is similar, but does output. Both of these constructs are "unwind-protected," in that the streams get closed even if there is an abnormal exit from the *body*.

(**WITH-INPUT-FROM-STRING** (*var string*) *-body-*) [*Magic*]

Binds *var* to stream that yields characters of *string* one by one, then executes *body*.

(**WITH-OUTPUT-TO-STRING** *var -body-*) [*Magic*]

Binds *var* to a stream that collects characters into a string, then executes *body*. An optional pair of parens can surround the *var*. The resulting string will be returned.

Most of the remaining functions described in this section have two versions, one beginning with `SRM` (which takes an explicit stream argument) or `STD` (which uses the standard input or output).

(**IS-EOF** *x*) [*Function*]

Except as indicated, all of the functions that try to read something will, when the end of a stream (= "end of file") is seen, return an object for which `IS-EOF` returns `#T`. There is only one such object.

From the terminal, these read functions may print an irritating prompt string.

(**SRMREAD** *s*) [*Function*]
(**STDREAD**) [*Function*]

Reads an S-expression.

(**SRMREADC** *s*) [*Function*]
(**STDREADC**) [*Function*]

Reads a character.

(SRMPEEKC *s***)** [*Function*]
(STDPEEKC) [*Function*]

Peeks at a character.

(SRMLINEREAD *s***)** [*Function*]
(STDLINEREAD *s***)** [*Function*]

Returns a list of the S-expressions appearing on the next line. May or may not prompt. Returns `()` if an empty line is seen (including end of file).

NEWLINE* [*Global Variable*]

Bound to the character used to end lines.

(SRMREAD-LINE *s***)** [*Function*]
(STDREAD-LINE) [*Function*]

Reads a line of input and return it as a string. Last line in file does not need to end in a newline.

(READ-OBJECTS-FROM-STRING *string***)** [*Function*]

A list of objects readable from the *string*.

(CLEAR-INPUT *s***)** [*Function*]

Clears input buffer.

(LISTEN *s***)** [*Function*]

`#T` if there is a character ready to read on *s*.

(SRMPRINT *x s***)** [*Function*]
(STDPRINT *x***)** [*Function*]

Prints an S-expression. No carriage returns.

(SRMDISPLAY *x s***)** [*Function*]
(STDDISPLAY *x***)** [*Function*]

Prints without slashifying.

(SRMBPRINT *x s***)** [*Function*]
(STDBPRINT *x***)** [*Function*]

Pretty-prints *x*, starting in current column. Returns final column.

(SRMPRINLEV *x d s***)** [*Function*]
(STDPRINLEV *x d***)** [*Function*]

Prints to a depth *d*.

(SRMPRINTC *c s***)** [*Function*]
(STDPRINTC *c***)** [*Function*]

Prints a character.

(SRMNEWLINE *s***)** [*Function*]
(STDNEWLINE) [*Function*]

Prints a `NEWLINE*`.

(SRMSPACES *n s***)** [*Function*]
(STDSPACES *n***)** [*Function*]

Prints n spaces.

(SRMTAB *n s***)** [*Function*]
(STDTAB *n***)** [*Function*]
Goes to column n (the leftmost column is numbered 1). If past it, does a `NEWLINE` first.

(SRMLINES *n s***)** [*Function*]
(STDLINES *n***)** [*Function*]
Prints n blank lines. If n is 0, does a `NEWLINE` only if not at beginning of line already.

(SRMCURRCOL *s***)** [*Function*]
(STDCURRCOL) [*Function*]
If the implementation supports it (and Common Lisp proper does not), returns the column number (1-based) where the next character will be printed. If the implementation does not support it, returns 1, i.e., guesses that there is a lot of room to the right.

(SRMLINELENGTH *s***)** [*Function*]
(STDLINELENGTH) [*Function*]
The length of the line on an output stream.

(FORCE-OUTPUT *s***)** [*Function*]
Forces buffered output to be actually sent.

(PRINTWIDTH *x***)** [*Function*]
(DISPLAYWIDTH *x***)** [*Function*]
The number of characters it would take to print x, slashified and unslashified, respectively.

2.4.3 User Non-hostile Constructs

(IN *-specs-***)** [*Magic*]
General input macro. Each *spec* is one of the following:

`(FROM` *stream*`)`: Tells what stream following *specs* are from. If omitted, use standard input.

`OBJ` (or `READ`, `T`, or `OBJECT`): Read a readable object, typically an S-expression.

`CHAR`: Read a character.

`PEEK`: Peek at a character.

`LINESTRING`: Read a line as a character string.

`LINELIST`: Read a line as a list of S-expressions.

The objects read are collected and returned as multiple values. Example: `(IN CHAR OBJ)` returns a character and the following object.

(OUT *-specs-***)** [*Magic*]
General output macro. Each *spec* is one of the following:

- A positive number: Skip that many spaces.
- A negative number: Skip that many lines (after negating it).
- Zero: Be at beginning of line.
- A T: output a new line.
- A string: Print it without quotes or slashes (`DISPLAY` it).

- A list of one of the forms:
  - `(TO` *stream*`)`: Shift output to that stream. Initially standard output.
  - `(T` *num*`)`: Tab to that column.
  - `(D` *-exps-*`)`: Evaluate and `DISPLAY` the *exps*.
  - `(S` *-exps-*`)`: Evaluate the *exps*, and interpret numbers as spacing commands. Everything else is `DISPLAY`ed.
  - `(PP` *e*`)`: Pretty-print *e*.
  - `(E` *-exps-*`)`: Evaluate the *exps* and discard values.
  - `(Q` *-clauses-*`)`: Each clause is of the form `(`*test -out-stuff-*`)`. Each test is evaluated, and `OUT` processing resumes on the *-out-stuff-* of the first true one.
- Anything else: Evaluate it and `SRMPRINT` it to whatever stream is being used.

(MSG ...) [*Magic*]
(STDMSG ...) [*Magic*]
(SRMMSG *s* ...) [*Magic*]
(TTYMSG ...) [*Magic*]

The first two are just synonyms for `OUT`. `SRMMSG` is the same, except that the first argument is interpreted as meaning `(TO` *s*`)`. `TTYMSG` sends to interactive terminal. (`TTYMSG` differs from `(OUT (TO TTYOUT*) ...)` in that it uses `FORCE-OUTPUT` to make the characters come out in real time.)

2.4.4 Pretty Printing

(SRMBPRINT *x stream*) [*Function*]
(STDBPRINT *x*) [*Function*]

The two pretty-printers (see previous section). They already know to print things like `(QUOTE` *x*`)` as `'`*x*. To tell them how to print something whose `CAR` is the symbol *sym*, do

```
(DATAFUN BP sym
   (DEFUN (X TR COL)
      ...))
```

(See `DATAFUN`, p. 27.) This function will be passed `X` when `X` is of the form `(`*sym* `...)`. `COL` will be the current print column. The standard output will be the stream pretty-printing is going to. `TR` is a "size tree," a data structure giving the print sizes of all pieces of `X`. Although it can be more efficient to make use of this, it is simpler just to ignore it. The function should return either the new current column when `X` is printed, or `()` if you want the default print routine to take over and print `X`.

2.4.5 Files and Filenames

We adopt the Common Lisp *pathname* datatype. A pathname is a special data object with six fields — host (file system), device, directory, name, type, and version. Pathname objects should not be confused with their printed representations, which may not display all component values, nor with *namestrings* such as "foo/bar.lisp". The precise way such a namestring is represented in a pathname object depends on the specific implementation.

(PATHNAME-HOST *pathname*) [*Function*]
(PATHNAME-DEVICE *pathname*) [*Function*]
(PATHNAME-DIRECTORY *pathname*) [*Function*]
(PATHNAME-NAME *pathname*) [*Function*]

(**PATHNAME-TYPE** *pathname*) [*Function*]
(**PATHNAME-VERSION** *pathname*) [*Function*]

Fields are null (= `()`) if absent from the pathname data object. The fields are often strings, but don't count on it: lists of strings, symbols, numbers and other objects can also appear. The only rule is that if a value came from a given field of a pathname, it's legal to use it as the value of that field in a new pathname.

In addition to the above, NILS implements its own "logical names." A symbol may have a `LOGICAL-NAME` property, which should be a pathname. Whenever the symbol appears where a file name is supposed to be, and is terminated by a colon or slash, it stands for that pathname. For instance, if `FOO` has a `LOGICAL-NAME` which is a pathname corresponding to directory "~/phou/", then the string `"FOO/baz.t"` corresponds to "~/phou/baz.t". (Note that, because `FOO` occurs inside a string, it must appear in upper case, unless it is the symbol `|foo|` that has the `LOGICAL-NAME` property.)

(**CONS-PATHNAME** [*host device directory name type version*]) [*Function*]

Makes a pathname based on the specified fields. Any omitted arguments may default to `()`, or may be given implementation-specific defaults.

(**->PATHNAME** *something*) [*Function*]

Converts string or symbol to pathname, obeying NISP logical-name convention. If *something* is already a pathname, it is returned. If *something* is a symbol representing just the name of the file, its case may be switched, depending on the host file system. In particular, on a Unix system, `(->PATHNAME 'FOO)` will return a pathname with `NAME "foo"`. This switch will *not* occur with more complicated symbols; `(->PATHNAME 'FOO.NSP)` returns a pathname with `NAME "FOO"`.

(**IS-PATHNAME** x) [*Function*]

Tests whether it's a pathname.

(**PATHNAME->STRING** *pathname*) [*Function*]

Converts to a string.

(**MERGE-PATHNAMES** *pathname defaults*) [*Function*]

Constructs a new pathname, with the same fields as *pathname*, with null values filled in from *defaults*, which is also a pathname.

(**PROBEF** *filename-or-pathname*) [*Function*]

Tests whether the file is there.

(**EVALFILE** *filename-or-pathname*) [*Function*]

Reads, evals, but does not print the things in the given file.

(**LOADOREVAL** *filename-or-pathname*) [*Function*]

If *filename* names an object file, loads it. Otherwise, `EVALFILE`s it.

(**FILESPECS->PATHNAMES** *filespecs*) [*Function*]

filespecs is a list of strings and symbols, each corresponding to a pathname. Some of these describe complete filenames, and others describe directories, hosts, or the like. `FILESPECS->PATHNAMES` scans through the *filespecs* in order, collecting the incomplete filenames, and merging their pathnames with those of the later complete filenames, returning the latter as pathnames. E.g., if `FOO` has a `LOGICAL-NAME` property as described above, `(FILESPECS->PATHNAMES '(FOO/ BAZ "blech.nsp"))` will return a list of two pathnames, one for "~/phou/baz" and the other for "~/phou/blech.nsp".

(DSKLAP `[-A] [-F]` *-filespecs-*) [*Magic*]

Loads in the indicated files. The *filespecs* are as for `FILESPECS->PATHNAMES`, which is used to parse them into pathnames. If a pathname specifies both the name and type of a file, that file is the indicated one. Otherwise, if the type is unspecified, DSKLAP will do some thinking. It wants to load the object version of the file if possible, so it uses the strings in the list `OBJECT-SUFFIXES*` to try to complete the pathname. For instance, on the TI Explorer, the only element of this list is `"XLD"`. But it also uses the strings in the list `SOURCE-SUFFIXES*` to find a source file as well. The first element of this list is normally `"NSP"`, and there are usually other elements, such as `"T"`, `"L"`, or the like. If `()` is an element of the list, that means "no extension."

If `DSKLAP` finds an object file and no source file, the object file is loaded. But if there is a source file and no object file, or a source file is found to be newer than the object file (not all implementations can detect this), then, depending on the value of the global variable `DSKLAP-COMPILE*`, it will consider compiling the source file and loading the resulting object file. The value of `DSKLAP-COMPILE*` is either `COMPILE`, `SOURCE`, `OBJECT`, or `ASK`. If it is `COMPILE`, an old object file is always overwritten with a freshly compiled one; if it is `SOURCE`, the source file is always loaded; if it is `OBJECT`, the object file is loaded if it exists. If the value is `ASK`, the user is told about the uncompiled source file and prompted with `"Compile it now?  "`. He can type `y`, `n`, `+`, or `-`. The first two responses have the obvious meaning, while `+` means "Set `DSKLAP-COMPILE*` to `COMPILE`," and `-` means "Set it to `SOURCE`." If the user types `n`, then he is further prompted for whether to load object or source, and whether to remember this response if the file is encountered again. The `-A` flag will reset `DSKLAP-COMPILE*` to `ASK`.

There is one other complication. If `DSKLAP` has already loaded a file with a given `NAME` field, then it will not load that file or any other file with the same name, even from another directory. To override this convention, just use the `-F` flag as the first argument to `DSKLAP`. This flag forces all the files to be loaded, even if they have been loaded before.

As a special case, `(DSKLAP)` with no arguments just retries the previous `DSKLAP`. `(DSKLAP -F)` retries the previous one with the `-F` flag on.

(DEPENDS-ON *system-symbol*) [*Magic*]
(DEPENDS-ON [*flag*] *-DSKLAP-style-filespecs-*) [*Magic*]

The `DEPENDS-ON` macro is used, normally near the top of a file, to declare other files or systems that this one depends on. It comes in two forms. In the first, a single symbol follows `DEPENDS-ON`, and this symbol has a `DEPENDS-ON` property that consists of a form to be evaluated whenever this file is loaded. The form typically loads in a supporting system, and does some other chores. The most common example is the form `(DEPENDS-ON NISP)`, which must appear at the front of every file that uses NISP types (Chapter 4), and causes various type-related things to happen when the file is loaded.

The other form is used to indicate what files need to be loaded (using `DSKLAP`) when this file is. The *flag* is either `AT-RUN-TIME` or `AT-COMPILE-TIME`. `AT-COMPILE-TIME` means that the following specified files contain code that must *run* when this file is *compiled*. A file needed `AT-COMPILE-TIME` will be `DSKLAP`'ed in when this file is compiled, or loaded before compilation. It will not be loaded when the compiled file is loaded.

`AT-RUN-TIME` means that the code in the specified files will not be executed until some code in this file is executed. The files will be loaded when this file is loaded, even if this one has been compiled. At compile time, the specified files are not loaded, but they are *slurped*. "Slurping" means going through each file, and loading essential information about the contents of the file. This information includes macros, and it may include other things, notably the NISP declarations found in the file. (See Chapter 4.)

If the flag is omitted, the filespecs will be loaded for both running and compilation. This is rare, and if you think it's necessary, what you probably really want is `NEEDED-BY-MACROS`.

(**NEEDED-BY-MACROS** *-forms-*) [*Magic*]

Appears at top level of file, and has no effect on the evaluation of the forms. (It's as if they appeared in the file unbracketed.) However, if some other file `DEPENDS-ON` this one `AT-RUN-TIME`, then the forms will be evaluated when that other file is compiled.

Here is the typical place where this is useful:

```
File 1:
  (DEFMACRO MAC (...)
      ... (AUXFUN ...) ...)

  (NEEDED-BY-MACROS
  (DEFUN AUXFUN (... ) ...)
  )

File 2:
  (DEPENDS-ON AT-RUN-TIME FILE1)

  (DEFUN FOO ( ... )
     ... (MAC ...) ...)
```

File 2 depends on the macro `MAC`, defined in File 1. Since `MAC` calls `AUXFUN`, it must be surrounded by `NEEDED-BY-MACROS` to make sure that it is defined when File 2 is compiled (and `MAC` is run).

2.5 CREATING AND COMPILING FILES

A NILS or NISP program consists of one or more files. Each file should start like this:

```
;;; -*- Mode:Common-Lisp; Package:NISP; Base:10 -*-
(HERALD filename (READ-TABLE NISP-READ-TABLE*)
                 (SYNTAX-TABLE NISP-SYN*))
(IN-PACKAGE 'NISP)

[(DEPENDS-ON [NISP | NILS | ... ])]

(DEPENDS-ON AT-RUN-TIME -various-other-files-)

[(OVERDRIVE)]

– all the code –
```

The first few lines are an attempt to tell every possible T or Common Lisp system what read table, syntax table, package, etc. are to be used. The first `DEPENDS-ON` is necessary if the file uses any part of NISP beyond the NILS kernel. In particular, if you use the type system, described in Chapter 4, you must provide a `(DEPENDS-ON NISP)`. If you use only NILS and its utilities, you must write `(DEPENDS-ON NILS)`.

If this file requires other files to be loaded at run time or compile time, express those dependencies with another `DEPENDS-ON`.

If, when compiled, the file is to be optimized for speed, with safety unimportant, put `(OVERDRIVE)` early in the file. This should be done only when the file is well debugged.

(**NISCOM** `[-F]` *-filespecs-in-DSKLAP-format*) [*Magic*]

Compile all the given files. This funoid is the authorized method for compiling any NISP or NILS file.

`NISCOM` will not compile a source file that appears to be as old as the object file that would be generated. To override this convention, use the `-F` flag to force compilation.

2.6 ERROR HANDLING

(**EARROR** *function value -msgs-*) [*Magic*]

Simulates an error. It prints the *msgs* (in `OUT` format), then enters a read-eval-print loop.

`EARROR` is often parasitical on the host error system. Such systems often have a notion of aborting versus resuming from an error. Aborting is usually done by hitting control-something, or by evaluating something like `(RESET)`. Resuming is done by typing `OK` or `(RET)`. Sometimes the user has the option of resuming with a value or resuming without a value. In the former case, this value will be returned as the value of `EARROR` and execution will continue. In the second case, the second argument to `EARROR` will be evaluated, and that value used instead.

In most implementations, `EARROR` tries hard to allow the following: To proceed with a value, type `RETURN` *val* (with *no* parens); to proceed with the default, type `OK`. In some dialects, it is necessary to tell the system to proceed first, after which it will prompt you for whether or not you want to supply a value. In some of those systems, you then type `OK` or `RETURN` ...; in other systems, something else entirely happens.

These multifarious conventions can lead to confusion, because the *-msgs-* in an `EARROR` call will often say things like

```
"Type ‘RETURN num’ to proceed with corrected data"
```

and it is important to remember that you must issue the “resume” command first.

2.7 HOST LANGUAGES & SYSTEMS

While NISP is designed to allow portable code, ignoring differences between the host Lisp dialects and machine characteristics, it is sometimes necessary to take such differences into account. Two global variables are used to reflect the current configuration:

HOST-DIALECT* [*Global Variable*]

A constant bound to the current host Lisp dialect (currently either T or COMMON).

HOST-SYS* [*Global Variable*]

A constant bound to the current host operating system (e.g. UNIX, AEGIS, VMS, HP, SYMBOLICS, TI).

Two read macros are defined allowing expressions to be read or ignored conditionally, depending on the values of these two variables:

`!D([-]` *-dialects-*`)` *expression*
`!S([-]` *-systems-*`)` *expression*

When the reader encounters `!D` or `!S`, the following list of host dialects or operating systems is compared with the current value of `HOST-DIALECT*` or `HOST-SYS*`, respectively. If there is no match, the following *expression* is ignored (by the reader; that is, nothing will be read). Otherwise, the expression is read as usual. “Matching” is defined as you might expect. If the list doesn’t start with a hyphen, then it must include `HOST-DIALECT*` or `HOST-SYS*`; if the list does start with a hyphen, then it must *not* include the host dialect or system. For example:

```
!S(UNIX)(CONVERT-FILENAME-TO-LOWERCASE ...)
!S(- UNIX)(TRY-OTHER-FILENAME-OPTIONS ...)
```

will result in only one of the two expressions being read, depending on whether or not the current operating system is UNIX.

To facilitate customization for specific Common Lisp implementations, the `#+` and `#-` Common Lisp read macros can be used directly, and are both equivalent in T to `!D(- T)`.

Chapter 3

NILS Utilities

3.1 BETTER SETTERS

(**!=** *exp val*) [*Magic*]

Makes *exp* equal to *val*, and returns *val*. Equivalent to `SETF`, except that on the right-hand side of an assignment, the symbol `*-*` stands for the left-hand side. So, to add 1 to the variable `TOTAL`, write `(!= TOTAL (+ 1 *-*))`.

Note that absolutely nothing clever happens with `*-*`; it simply gets replaced by a copy of the left-hand side. If the left-hand side is expensive or has side effects, you lose.

A special case is (!= < *v1 v2 ...* > *e*) which assigns the variables to the multiple values returned by *e*. If there is more than one *e*, then a `VALUES` is wrapped around them. Hence two variables can be swapped by saying (!= < v_1 v_2 > v_2 v_1).

Another special case of `!=` is (!= (< *v1 v2 ...* >) *list*) which assigns the variables to successive elements of a list.

(**!=/** *exp val*) [*Magic*]

Like `!=`, except that it returns a list showing the "condensed" version of the previous value and new value of the expression. For example, if `G*` has value `(A B C)`, `(!=/ G* '(D E F))` will return `((WAS (A --)) (NOW (D --)))`. If there was no previous value, it returns the expression itself. `!=/` is mainly useful at the top level for setting variables with unprintable values. (Courtesy of E. Davis.)

(**SWITCH** *exp1 exp2*) [*Magic*]

Sets *exp1* to *exp2* and *exp2* to *exp1* simultaneously. The value is undefined. (Courtesy of E. Davis.)

(**MATCHQ** *pattern form*) [*Magic*]

Turns into LISP code to test if *form* matches *pattern* and, if so, set the variables of *pattern*. For instance, `(MATCHQ (A !@B . ?X) VV)` becomes

```
(AND (IS-PAIR VV)
     (EQ (CAR VV) 'A)
     (IS-PAIR (CDR VV))
     (EQ (CADR VV) B)
     (PROG1 T (!= X (CDDR VV))))
```

or something equivalent and uglier. (The macro produces code that compiles efficiently, but may interpret inefficiently.) Anything marked with an `!@` is unquoted, so in the example `A` means the symbol `A`, but `!@B` means the value of variable `B`. Anything marked with `?` is a variable to be set to the part of the form that it winds up in correspondence with. So if `B` is `(P Q)`, then `VV` = `(A (P Q) ZIP ZAP)` will match and set `X` = `(ZIP ZAP)`, while `VV` = `(A P Q ZIP)` will fail to match.

If the `?` is followed by `()`, then it will match anything without setting a value.

Any subexpression of the pattern of the form `?(&` *-pats-*`)` will match if all of the *pats* match. Similarly for `?(\|` *-pats-*`)`, which matches if any of the *pats* match. If a variable is repeated, as in `(A ?X ?X)`, then it gets the last value it is matched against; it does not have to match the same thing every time it occurs. So `(A ?X ?X)` matches `(A B C)` with `X` set to `C`. If the match fails, the values of the pattern variables are undefined.

`(`**MATCH-VARS-BIND** *-body-*`)` `[`*Magic*`]`

Equivalent to `(LET` *match-vars -body-*`)`, where *match-vars* is a list of all the symbols *v* such that `?`*v* occurs somewhere in *body*. The search for match variables in the body is not at all sophisticated, so this construct is not that useful if quoted variables occur in the body.

`(`**MATCH-COND** *x -clauses-*`)` `[`*Magic*`]`

Behaves like

```
(LET ((MATCH-DATUM x))
   (MATCH-VARS-BIND (COND -clauses-)))
```

with two extra features:

1. Any clause of the form

   ```
   ?(pat ...)
   ```

   is transformed into the form

   ```
   ((MATCHQ pat MATCH-DATUM) ...)
   ```

   so they are in the same form as the second type.

2. Any occurrence of

   ```
   (MATCHQ pat)
   ```

   (i.e., `MATCHQ` without its second argument) is treated as

   ```
   (MATCHQ pat MATCH-DATUM).
   ```

So

```
(MATCH-COND (BLAT V)
   ?((FOO ?X) (TTYMSG "FOO " X T))
   ((MATCHQ (BAR ?Y) (CAR V))
    (TTYMSG "(BAR " Y ")" T))
   (T (TTYMSG "NO MATCH"))  )
```

is the same as

```
(LET ((MATCH-DATUM (BLAT V)))
   (LET (X Y)
      (COND ((MATCHQ (FOO ?X) MATCH-DATUM)
              (TTYMSG "FOO " X T))
             ((MATCHQ (BAR ?Y) (CAR V))
              (TTYMSG "(BAR " Y ")" T))
             (T (TTYMSG "NO MATCH"))  )))
```

3.2 MAGIC MAPPERS

For those who think APL is too verbose, we provide a concise set of abbreviations for the mapping functions:

(<**#** [\.] *-function-spec- -listargs-*) [*Magic*]
(<**$** [\.] *-function-spec- -listargs-*) [*Magic*]
(<**!** [\.] *-function-spec- -listargs-*) [*Magic*]
(<**&** [\.] *-function-spec- -listargs-*) [*Magic*]
(<**V** [\.] *-function-spec- -listargs-*) [*Magic*]
(<**?** [\.] *-function-spec- -listargs-*) [*Magic*]
(<\ [\.] *-function-spec- -listargs-*) [*Magic*]
(</ [\.] *-function-spec- -listargs-*) [*Magic*]
(<< [\.] *-function-spec- -listargs-*) [*Magic*]

These are macros beginning with < that abbreviate MAPELTLIST and company. In general, they have the following syntax:

(<*char* [\.] *-function-spec- -listargs-*)

The *function-spec* is *not* evaluated; the *listargs* are evaluated. The most common *function-spec* is the name of a function or an expression of the form

(\\ (*-vars-*) *-body-*)

However, there are other possibilities, described below.

For example, the concise version of MAPELTLIST is called <#, as in:

```
(<# REVERSE '((A B) () (D) (P Q R)))
```

which means the same as

```
(MAPELTLIST #'REVERSE '((A B) () (D) (P Q R)))
```

and has value ((B A) () (D) (R Q P)).

The optional \. after the name of the mapper specifies TAIL mapping instead of ELT. So

```
(<# \. REVERSE '((A B) () (D) (P Q R)))
```

is the same as

```
(MAPTAILLIST #'REVERSE '((A B) () (D) (P Q R)))
```

and evaluates to

```
=> (((P Q R) (D) () (A B))
    ((P Q R) (D) ())
    ((P Q R) (D))
    ((P Q R)))
```

Here is a table of all the concise mappers (and a couple of relatives):

<#	MAP...LIST
<$	MAP...APPEND
<!	MAP...CONC
<&	MAP...AND
<V	MAP...OR
<?	MAP...COLLECT
<\␣	MAP...DO
</	MAP...REDUCE
<<	APPLY

Note that the `DO` equivalent has a space in its name, after the `"\"`.

A construct with similar syntax is

(RMV-IF `[D] [A]` *pred list*) [*Magic*]

Produce a new list with the elements of the old list satisfying the predicate removed. If the `D` is present, do it destructively. If the `A` is present, remove all the elements, else just the first. Note that it is important in the destructive case to store the value returned. That is, say `(!= X (RMV-IF D ... X))`. Otherwise, if the first value in the list is one of those removed, the result will be wrong.

NEG [*Other*]
IS [*Other*]
!_ [*Other*]

All of the macros described in this section (except <<) allow some useful extensions in specifying the *function-spec*. For example, `NEG` in front of the function turns it into `(\\ (X) (NOT (`*function* `X)) )`. So, if you write `(<? NEG ATOM L)`, you will get all the non-atoms in `L`. When using the NISP type system, you can also write `(<& IS` *type-desig* `L)`, or `(<# !_(`*type-desig slot*`) L)`. You can catenate these things, getting, e.g., `(<? NEG IS symbol L)`.

These work because `NEG`, `IS`, and `!_` have

MAPMAC [*Other*]

properties. The value of this property is a function that takes the list beginning with the symbol so flagged, and returns a list of the form `((FUNCTION` *fun*`)` *-listargs-*`)`. `(`*mapper sym* `...)` is then equivalent to `(`*mapper fun -listargs-*`)`. For instance, `NEG`'s mapmac function returns

```
((FUNCTION (LAMBDA (X) (NOT (ATOM X)) )) L)
```

in the case given above.

3.3 LAZY LISTS

A *generated list* is a list-like object whose elements are computed on demand, "lazily," as the expression goes. Such a list may be stepped through using `SAR` and `SDR` instead of `CAR` and `CDR`. In NISP, a generated list is implemented as an ordinary list some of whose elements are flagged as *generators*, corresponding to functions that can be called to make more elements. If you step through such a list with `CAR` and `CDR`, you will actually get the generators. If you use `SAR` and `SDR`, the generators will be called as they are encountered, and you will see the elements they generate.

(SAR *generated-list*) [*Function*]
(SDR *generated-list*) [*Function*]

`SAR` returns the next element in a generated list, or `()` if there are no more. `SDR` returns a new list whose first element is the next element after the first; or `()` if there are fewer than two elements in the list.

The "S" in the names of these functions stands for "stream."

(*GEN *closure*) [*Function*]

Creates a generator. The *closure*, of no arguments, will generate more things when called.

You usually call `*GEN` indirectly, through the `LAZYLIST` macro:

(LAZYLIST *-body-*) [*Magic*]

Evaluates to a g-list whose `CAR` is a generator that will generate body.

Lazy lists can be stepped through using the `GEN` construct in `LOOP` , which corresponds to `IN` for ordinary lists:

```
(LOOP FOR ((var [GEN | GENERATED-BY] gl)) ...)
```

(EXTRUDE *n gl*) [*Function*]

Forces the generators in *gl* to cough up at least *n* objects. Note that `EXTRUDE` alters and returns the original stream, including remaining generators if any. If *n* objects cannot be generated, just returns the list with all generators expanded.

(NORMALIZE *gl*) [*Function*]

Called by `SAR` and `SDR` to force *gl* to either start with a non-generator or be `()`. You can't really tell whether an un-`NORMALIZE`d g-list is empty.

Example:

```
;; Generate all the atoms in an S-expression X
(DEFUN ATOMS (X)
   (COND ((ATOM X) (LIST X))
         (T (NCONC (ATOMS (CAR X))
                   (LAZYLIST (ATOMS (CDR X)))))   ))
```

Now

```
(LOOP FOR ((A GEN (ATOMS '((A . B) ((C) . D) (E (F . G) . H)))))
         UNTIL (NULL A)
         (OUT A T)   )
```

prints

```
   A
   B
   C
```

(<#S *closure gl*) [*Function*]

Like `MAPELTLIST (<#)` for lists, returns a g-list of *closure* applied to each element of *gl* in turn.

(<!S *closure gl*) [*Function*]

Like `MAPELTCONC (<!)` for lists. The *closure*, applied to an element of the g-list *gl*, must return a g-list. So (`<#S` *closure gl*) would return a g-list of g-lists. (`<!S` *closure gl*) returns a g-list containing (eventually) every element of every g-list of the g-list of g-lists (just as (`<!` *function list*) returns a list containing every element of every list in the list of lists (`<#` *function list*)).

3.4 OBJECTS AND OPERATIONS

NISP has T-style objects, abstract entities that respond to *operations*, which are syntactically identical to functions. Defining an object is just specifying how it responds to various operations.

This whole area is in a state of flux, and you can expect extensions to the facilities described here as things like CLOS (Common Lisp Object System) mature.

(**DEFOP** *name* (*ob* -*args*-) -*body*-) [*Magic*]

Defines an operation. A form (*name x* ...) will be evaluated by first attempting to have *x* handle the operation; that is, if *x* is an object or member of a class that knows about operation *name*, then the code associated with *x* is run. Otherwise, just as for an ordinary function call, *ob* and the other *args* are bound, and the *body* is evaluated. The *body* is allowed to be empty, in which case an error is signaled if the first argument cannot handle the operation.

(**MAKE-OBJECT** *clauses*) [*Magic*]

Returns an object that handles operations as specified by the *clauses*. Each *clause* is of the form (*operation* (-*args*-) -*body*-), and defines a procedure to be run when that operation is applied to this object.

(**DEFCLASS** *name clauses* -*slotnames*-) [*Magic*]

Defines a globally-defined object class where *slots* is a list of slot names, and *clauses* are as for `MAKE-OBJECT`. (A class is not a type in the NISP sense (Chapter 4). To define types corresponding to classes, see section 4.4.2.)

After evaluating a `DEFCLASS`, you can make instances of the class by calling the constructor, (`MAKE-`*name* -*slotcontents*-). It takes as many arguments as there are slots, in the same order. There are then two kinds of thing you can do with a class instance: access and set its slots, and perform operations on it. The slot accessors are called *name*-*slot*. To change the contents of a slot, write (`SETF` (*name*-*slot* ...) ...).

Operations are handled as spelled out by the *clauses*, which are in the same format as for `MAKE-OBJECT`.

Note: Each clause begins with an operation name, which in general must have been defined using `DEFOP`, but there are some exceptions. In T, you may use any system-defined operation (although of course code using such an operation won't be portable). In both T and Common Lisp, you can use `PRINT` as if it were an operation, even though `PRINT` is not actually part of NISP at all. Nevertheless, it can appear in the *clauses* of `MAKE-OBJECT` or `DEFCLASS`, and will get control when a value of the `MAKE-OBJECT` expression or an instance of the `DEFCLASS` is printed. It takes two arguments, the object to be printed and the stream to print it on. E.g., one can write things like:

```
(DEFCLASS PEAR ((PRINT (X STREAM)
                   (OUT (TO STREAM)
                        "#<PEAR " (PEAR-I X) ", "
                                  (PEAR-J X) ">")))
    I J)
```

and then if `P1` is set to `(MAKE-PEAR 5 6)`, it will print out as `#<PEAR 5, 6>`.

`DEFCLASS` defines a test function for instances, called `IS-`*name*. To test whether an object *x* is an instance of a class, call (`IS-`*name x*).

For more on objects and operations, see section 4.4.2.

Chapter 4

NISP Type System

Modern programming languages are built around mechanisms for *abstraction,* concealment of implementation details of abstract data types. Lisp dialects include tools like structures and flavors for this purpose. NISP integrates these tools into a coherent package for

1. Defining abstract data types.
2. Declaring variables of those types.
3. Checking for type violations.

For example, suppose we wanted to define a new abstract data type, "Cartesian points in two-space." Here is how we might do that:

```
(DEFTYPE cpoint (STRUCTURE X Y - float))
```

This definition is entirely analogous to a `DEFSTRUCT`; indeed, in Common Lisp it will expand into a `DEFSTRUCT`. However, using it allows us to define more concisely functions that manipulate `cpoint`s. For example:

```
(DEFFUNC MAGNITUDE - float (P - cpoint)
   (SQRT (+ (* (!_X P) (!_X P))
            (* (!_Y P) (!_Y P))))   )
```

This code defines a function `MAGNITUDE` that returns a `float` value given a `cpoint` argument. It uses the formula for distance from the origin to find the magnitude of `P`. The notation `(!_X P)` means to get the contents of the `X` slot of `P`. Because `P` has been declared to be of type `cpoint`, the `X` slot can be determined at compile time to be a certain position in the vector used to implement `P`. Furthermore, the system automatically infers that `(!_X P)` is of type `float`, and hence can open-compile the multiplications. The code above is analogous to the Common Lisp

```
(DEFSTRUCT cpoint (X 0.0 :TYPE FLOAT) (Y 0.0 :TYPE FLOAT))

(DEFUN MAGNITUDE (P)
   (DECLARE (TYPE cpoint P))
   (SQRT (+ (* (CPOINT-X P) (CPOINT-X P))
            (* (CPOINT-Y P) (CPOINT-Y P))))   )

(PROCLAIM '(FTYPE (FUNCTION (CPOINT) FLOAT) MAGNITUDE)))
```

but much clearer and more concise.

To use the type system, all you have to do is (a) use macros like `DEFFUNC` and `DEFTYPE` to define functions and types; (b) put `(DEPENDS-ON NISP)` at the front of your file (to make sure the file is slurped before it is compiled). Note that, just as in Common Lisp, type declarations are used only at compile time.[1] At run time, the code looks the same as ordinary Lisp code, except possibly for the presence of more efficient object code.

4.1 EXPRESSION TYPES

In normal Lisp, the type of an object is simply a predicate that it satisfies. An object can be of several types simultaneously, although for convenience one of them may be considered to be "the" type of the object. A variable's type is just the type of its value, and this can change.

With compile-time typing, we get a whole new sense of the word "type." The type of an expression can be considered to be the narrowest class of objects such that all the values it will ever have fall into that class. We will use the phrase *expression type* for this sense; in the usual phrase "type of a variable," the expression type is meant.

In code like

```
(DEFFUNC MAGNITUDE - float (P - cpoint)
   (SQRT (+ (* (!_X P) (!_X P))
            (* (!_Y P) (!_Y P))))   )
```

the identifiers `float` and `cpoint` are *type designators*. The designator denotes the type, which is an abstract object associating slots with access functions. For instance, the type denoted by `cpoint` associates with `!_X` and `!_Y` functions to extract the first and second slots of a certain structure. Most of the time users will not have to worry about the distinction between types and their designators. In the current implementation of NISP, the association between a designator and its type is global; there are no local type definitions. In any case, the associations exist purely at compile time, and have nothing to do with variable bindings; `cpoint` does not have an abstract type as its value.

"`!_`" is a macro that inspects its arguments and expands into the appropriate accessing function. The general form of `!_` is `(!_(`*type slot*`)` *x*`)`, in which both the type and slot of x are specified, but usually NISP can infer the type from declarations, and the *type* and parens can be omitted.

Most slots are settable. That is, you can write `(!= (!_X P)` *new*`)`, and from then until the next such setting, `(!_X P)` will have the new value.

An alternative syntax, for those who like C, is `!>`*object*`.`*slot*, as in "`!>P.X`". (If dots bother you, you can also write "`!>P>X`".) This notation can be iterated, so that

`(!_`s_n `(!_... (!_`s_2 `(!_`s_1 `e))))`

may be written

`!>e.`s_1`.`s_2`.....`s_n`.`

or, if you prefer, as

`!>e>`s_1`>`s_2`>···>`s_n`.`

Many types are associated with functions for testing for membership in the type. If a type t has such a function, then you can write

(IS t x**)** [*Magic*]

[1] Some implementations will insert run-time checks to verify type declarations when the value of `SAFETY` is high.

to test whether x is of that type. This construct is analogous to Common Lisp's `(TYPEP` *x* `'`*t*`)`, except that t is not evaluated. Thus we write `(IS fixnum N)`, and this will be turned into an expression to test if `N` is a fixnum. Note that the test occurs at run time, and has nothing to do with whether `N` is declared to be of type `fixnum`. [2] It is possible to test for t-hood in this way only if type t has an "IS-tester" associated with it; such a type is said to be *IS-testable*.

For some types, especially user-defined types (section 4.4.2), new objects of that type are constructed using

(**MAKE** *type* ...) [*Magic*]

The arguments depend on the type, and are typically initial values of some or all of the slots of the new object. Most built-in types are not "*MAKE-able*" in this way because they don't have slots. To construct, e.g., a string, you just use the normal string-constructing functions.

4.2 BUILT-IN TYPES

Here are the types built in to NISP. Where possible, they have the same names as the corresponding Common Lisp types. Atomic type designators are written the opposite case from the default, but this is only a convention. It makes code more readable, and is strongly recommended.

4.2.1 Simple Types

In this section we list the atomic-named types.

obj [*Type*]

Anything is an `obj`.

void [*Type*]

Nothing is a `void`. The main purpose of `void` is to serve as a placeholder for the value of a function that is executed for effect.

null [*Type*]

Only `#F`, the boolean false value, is of this type. In all implementations to date, however, `()`, the empty list, is `EQ` to `#F`.

boolean [*Type*]

`#T` or `#F`. But anything can be considered a `boolean`, so it isn't IS-testable.

symbol [*Type*]

An atomic symbol. (IS-testable)

string [*Type*]

A string. (IS-testable)

char [*Type*]

A character. (IS-testable)

number [*Type*]
float [*Type*]
rational [*Type*]
ratio [*Type*]

[2] Warning: Some compilers will optimize a `COND` clause away if its test can be deduced to evaluate to false based on type information.

integer [*Type*]
fixnum [*Type*]

A `number` is either a `float` (floating-point) or `rational`. `Rational`s are further divided into `ratio`s and `integer`s. A `fixnum` is a "small" `integer` (implementation dependent); nonfixnum integers are known as "bignums." (all are IS-testable)

sexp [*Type*]

An "S-expression": a non-circular list structure whose leaves are symbols, numbers, characters, strings, or null.

form [*Type*]

An executable expression. Two slots `FUN` and `ARGS`.

lambda-exp [*Type*]

An `sexp` of the form `(LAMBDA` *bvars* `.` *body*`)`. Two slots `BVARS` and `BODY`. (IS-testable)

macro [*Type*]

A symbol defined to be a macro (IS-testable)

stream [*Type*]

An I/O stream. (IS-testable)

pathname [*Type*]

A file name. Slots `HOST`, `DEVICE`, `DIRECTORY`, `NAME`, `TYPE`, and `VERSION`. None of the slots are settable. (IS-testable)

4.2.2 More Complex Types

Types can have nonatomic designators, in which case they are of the form `(`*type-constructor -stuff-*`)`. The simple ones are described here. Complex structured types are described in Section 4.4.2.

`(`**LRCD** *a* `.` *d*`)` [*Type*]

A cons cell whose `CAR` is of type *a* and whose `CDR` is of type *d*. If *d* is another `LRCD`, the `LRCD` can be dropped. So `(LRCD symbol integer . float)` means the same as `(LRCD symbol LRCD integer . float)`, an object whose `CAR` is a `symbol`, `CADR` is a `integer`, and `CDDR` is a `float`. (The name of this type constructor stands for "List ReCorD," an unfortunate historical accident.)

`(`**LST** *t*`)` [*Type*]

A list of elements of type *t*. IS-testable if *t* is.

`(`**GLST** *t*`)` [*Type*]

A generated list of elements of type *t*. See Section 3.3.

`(`**ARY** *type rank*`)` [*Type*]

An array of *rank* dimensions containing elements of the given *type*. If the rank is unknown at compile time, it may be written as `*` or omitted.

`(`**VCT** *type*`)` [*Type*]

A synonym for `(ARY` *type* `1)`.

(RCD *-types-***)** [*Type*]

A record containing as many slots as there are *types*, named <1>, <2>, ... For instance, type (RCD integer float) has two slots <1> of type integer and <2> of type float. So you can write

```
(SPECDECL (R1 (MAKE (RCD integer float) 4 5.0))
          - (RCD integer float))
```

followed by (!= (!_<1> R1) 6), and so forth.[3] RCDs are typically implemented as vectors. See also structures, described in Section 4.4.2.

(HTB *valtype***)** [*Type*]

A hash table containing elements of type *valtype*. (Currently, keys can be of any type, but in future the key type may be made explicit.)

(MLV *-types-***)** [*Type*]

The "type" returned by a function that returns multiple values of the corresponding types.

(FUN r (a_1 ... a_n) b**)** [*Type*]

A function that takes arguments of types a_1 ... a_n and returns a value of type r. If the a_i list terminates in a type designator instead of (), then starting with the argument in that position the function takes an indefinite number of arguments of that type. b is non-() iff the function has side effects. (See Section 4.3.1.)

/ is of type (FUN number (number number) ()).

+ is of type (FUN number number ()), because it takes any number of args.

A magic word does not have a type, because it does not denote a function at all.

(CONST c_1 ... c_n**)** [*Type*]

One of these constant S-expressions. (IS-testable)

(EITHER t_1 ... t_n**)** [*Type*]

The union of these types. (IS-testable if all the t_i are.)

(~ t) [*Type*]

Like type (EITHER t null), except that it inherits all the slots, is-testers, and the like from t. Useful for declaring variables that will normally be of type t, but in "degenerate" or exceptional cases are allowed to have value #F.

4.3 DECLARATIONS

4.3.1 Defining and Declaring Procedures

To define a procedure, declare its type, and declare the types of the variables inside it, use:

(DEFFUNC *name* - *rtype* (*-type-var-list-*) *-body-***)** [*Magic*]
(DEFPROC *name* - *rtype* (*-type-var-list-*) *-body-***)** [*Magic*]
(DEFOPFUNC ...) [*Magic*]
(DEFOPPROC ...) [*Magic*]
(FUNC ...) [*Magic*]
(PROC ...) [*Magic*]

[3] The syntax !>R1.<1> is not allowed.

(OPFUNC ...) [*Magic*]
(OPPROC ...) [*Magic*]

Define functions (or operators), declaring name to be of type (FUN *rtype* (-*argtypes*-) *sw*), where the *argtypes* are extracted from the *type-var-list* (see below), and the *sw* is () for DEFFUNC and TRUTH for DEFPROC. (DEFFUNCs have no side effects, and DEFPROCs do. FUNC and PROC are alternative names.) Within the *body* of a function defined with one of these constructs, variables are declared as specified in the *type-var-list*.

I will use the term *type-var list* for the declaration-and-binding specifications found in these constructs and elsewhere. The syntax is as follows:

var var ... - *type*
var var ... - *type*
...

which declares each group of *var*s to be of the following *type* (flagged by a hyphen). Older versions of NISP had variables and types interleaved without hyphens, and this syntax is still supported. In fact, the types can come *before* the variables, in the form

type var var ... *type var var* ...

Of course, you have to be consistent within a single type-var-list. In similar fashion, the hyphens before the *rtype*s in function definitions are a new feature, and may be omitted. (Although strange things can happen with undefined *rtype*s.)

Type-var-lists have a standard syntax throughout NISP, but it varies in obvious ways. For instance, when a variable is being bound, we must be able to supply an initial value, as described below.

In a function definition, we allow the occurrence of the keyword &REST. The variable after the &REST must be declared to be of type (LST *eltype*). If it is declared to be of any other type *t*, that is taken to be an implicit declaration of (LST *t*). So the following are equivalent:

```
(DEFFUNC FOO - baz (&REST X - (LST baz)) ... )
(DEFFUNC FOO - baz (&REST X - baz) ... )
(DEFFUNC FOO baz ((LST baz) &REST X) ... )
(DEFFUNC FOO baz (&REST baz X) ... )
```

In all cases, FOO is of type (FUN baz baz ()). (See definition of FUN, p. 49.)

What does the "side-effect" flag, the choice between DEFFUNC and DEFPROC, mean? Currently, it's only documentation.

I will use the phrase *declaration context* to refer to a context in which NISP declarations are in effect, such as in the body of a DEFFUNC. (Other declaration contexts occur in SPECDECLs and in DEFTYPE clauses, Section 4.4.2.) If you have no other way to create such a context, use:

(DECL (-*type-var-list*-) -*body*-) [*Magic*]

Evaluates *body* with the variables declared as indicated. Note that an uninitialized variable is not bound at all; the DECL form serves simply to declare it. (DECL (X (Y 5.0) - float) ...) allocates and declares Y, but only declares X (which had better be bound to a float by someone else beforehand).

4.3.2 Declaring Variables

This section describes mechanisms for binding and declaring typed variables.

(SPECDECL -*type-var-list*-) [*Magic*]

Declares and allocates variables globally, as in `(SPECDECL (PI 3.14159) - float)`. The variabless are automatically `DEFVAR`'d, or, if there is no initial value, `PROCLAIM`ed `SPECIAL`. (Exception: If a variable is declared to be of type `(FUN ...)`, it will not be declared special automatically. In this case NISP assumes you are declaring the type of a function identifier, not a global variable, and in Common Lisp these are two different things.)

Within a declaration context, the variable-binding constructs `LET`, `BIND`, `PROG`, `FOR`, `LOOP`, `LAMBDA`, `FLABELS`, and `MAKE-OBJECT` allow type-var-lists where their bound variables go. That is, you can say things like

```
;; Bind I to 5 and declare it integer
(LET ((I 5) - integer) ...)

;; Step I from 5 to 10, and declare it integer.
(LOOP FOR ((I = 5 TO 10) - integer) ...)

;; Step I through elements in L, and declare it  integer.
(FOR (I IN L) - integer ...)

;; Let FOO be of type (FUN float (integer)).
(FLABELS ((FOO - float (I - integer) ...)) ...)
```

Note that all the constructs obey the same consistent syntax. Unfortunately, consistency isn't everything. You may prefer the following syntax for `LOOP` and `FOR`:

```
(LOOP FOR (integer (I = 5 TO 10)) ...)

(FOR (integer I IN L) ...)
```

The first of these was already allowed above; the second is a `FOR` idiosyncrasy.

Suppose that no types at all appear in a type-var-list. In older versions of NISP, this syntax was equivalent to declaring all the variables of type `obj`, which was entirely equivalent to not declaring them at all. (I.e., NISP would never complain that they were the wrong type; see below.) In current versions of NISP, the situation is slightly different. If in a type-var list *none* of the variables are declared explicitly, then variables with initial values are declared to have the same type as the initial value; other variables are of type `obj`. So in

```
(LET (Y (X 5)) ...)
```

`Y` is of type `obj` and `X` is of type `integer`. This rule applies in most circumstances where NISP looks like it ought to be able to deduce the type of an expression.

Inside a declaration context, if a `COND` or `IF` clause begins `((IS` *type var*`) ...)` or `((IF-IS` *type var* `...) ...)`, then within that clause, the variable is declared to be of that type. (Idea courtesy of E. Charniak.)

Within a declaration context, the functions `MEMBER=`, `ADJOIN=`, `ASSOC=`, `UNION=`, `INTERSECTION=`, `IS-SUBLIST=`, `COMPLEMENT=`, `[D]REMOVE[1|-EVERY]=`, `NODUP=` and `DNODUP=` have an extended syntax. If the *eqtest* argument is replaced by a type designator, then the system looks for the slot `!_(`*type* `=)` for that type, and uses what it finds. If the *eqtest* argument is omitted altogether, then it defaults to the appropriate type: For `MEMBER`, `ADJOIN`, `ASSOC`, and `[D]REMOVE`...), this is the type of the first argument; for `UNION`, `INTERSECTION`, `IS-SUBLIST`, `COMPLEMENT`, `NODUP`, and `DNODUP`, it's the element type of the first list argument.

(**EQU** [*type*] *x y*) [*Magic*]

Synonymous with `(!_(`*type* `=)` *x y*`)`, but more readable. *type* should be the expression type of *x* and *y*, and may be omitted inside a declaration context. If the function `!_(`*type* `=)` is `EQ`, means the same as `EQ`; if `!_(`*type* `=)` is `EQUAL`, means `EQUAL`.

Often a single type instance is to have several of its slots inspected. To avoid having to rewrite the type and instance repeatedly, we can write

(WITH [*type*] *object* ...) [*Magic*]

Within the "...," in any occurrence of `!>`*object*`...`, the *object* can be omitted. Otherwise, the "..." behaves like a `PROGN` body; it is a list of expressions that is evaluated, with the value of the last being returned. For example, if `FOO` is declared to be of type `form`,

```
(LIST !>FOO.FUN 1 !>FOO.ARGS)
```

may be written

```
(WITH FOO (LIST !>.FUN 1 !>.ARGS))
```

or as

```
(WITH FOO (LIST !>>FUN 1 !>>ARGS))
```

The extra dot or bracket may be dropped in unambiguous cases (i.e., uniterated slot references), so this could be written

```
(WITH FOO (LIST !>FUN 1 !>ARGS))
```

But in general `!>.`s_1`.....`s_n cannot be written `!>`s_1`.....`s_n, because this notation means `(!_`s_n `(...(!_`s_2 s_1`)))`.

These examples have omitted the *type* argument to `WITH`; the type of the *object* is inferred from declarations. Supplying the type [re]declares the object within the scope of the `WITH`. The object expression does not have to be an identifier. It will be evaluated just once, so feel free to use it even if its evaluation has side effects.

(IF-IS *type x* ...) [*Magic*]

The very common idiom

```
(COND ((IS type x)
       (WITH type x ...))
      (T '#F)  )
```

may be abbreviated using `IF-IS`, as in

```
(IF-IS lambda-exp FOO (TTYMSG !>BVARS !>BODY)  )
```

4.4 USER-DEFINED TYPES

In this section we describe how to define new type designators, plus some complex types that are used in such definitions.

4.4.1 Defining New Types

(DEFTYPE *name type-desig -patches-*) [*Magic*]

This makes *name* designate a new type that behaves just like the *base type* designated by *type-desig*, as modified by the *patches*. A "patch" defines a new slot as a procedure for accessing its virtual contents (and possibly a procedure for setting those contents).

Example:

```
(DEFTYPE client (LST number)
   (SUM - number (C - client)
      (<< + C)   )
   (MAIN - number (C - client) (CAR C))   )
```

defines a new type `client` that consists of a list of numbers. Elements of such a list may be accessed with `CAR`, `LIST-ELT`, etc., but there are also two new slots, `SUM` and `MAIN`, defined as the sum of all the numbers, and the first number, respectively.

In general, each patch is of the form

```
([ | SET | ACCESS | TYPE | BOTH | ALL] slotname
        [- type]
        [*INTEGRABLE]
        [-function-definition-])
```

The first thing in the patch says whether the setter, accesser, or the type of the slot is being specified. `BOTH` means the setter and accesser are both being specified. `ALL` means the setter, accesser, and type are all being specified. (The first thing is optional; if omitted, `ACCESS` is assumed.) A patch may be used to override the accessor, setter, or type of an existing slot as well as to create a new one.

The slotname is any symbol, but the atoms `IS` and `CONSER` (or `IS-TEST` and `CONSTRUCTOR`) are assumed to be for testing membership in the type and constructing new members; and the atom "`=`" is assumed to be the equality test for the type.

The *type* is the type of the objects occupying the slot.

A function definition is of the form `(`*-type-var-list-*`)` *-body-*, just as for ordinary functions. The first argument should be declared to be of this very type, except for the constructor and is-tester. The constructor will take an arbitrary number of arguments; the equality tester will take two arguments. In general, accessors take one argument and setters take two, but this is not essential. If a slot accessor takes extra arguments, you write `(!_`*slot obj -additional-arguments-*`)`. The setter (when there is one) presumably takes as many arguments plus one.

***INTEGRABLE** *[Other]*

Normally a `DEFTYPE` patch gives rise to a new function definition. Any reference to the corresponding slot expands into a call to that function. If the flag `*INTEGRABLE` is put before the argument list, then instead the corresponding lambda expression will occur in-line everywhere the slot is accessed. In the example, if the last patch had been

```
(MAIN - number *INTEGRABLE (X) (CAR X))
```

then any expression of the form `(!_MAIN C)` would be transformed into `(CAR C)`.

^^ *[Other]*

Within a `DEFTYPE` patch, the symbol ^^ stands for the base type. So `(MAKE ^^ ...)` means "Make an object of that type, using the original argument order." Typically this construct is used in the definition of the `CONSER` for the derived type, so that the arguments to it can differ from those of the base type (usually by eliminating some). You can also use `(IS ^^ ...)`, `(!_(^^ slot) ...)`, and so forth.

A type must be defined before it is used. You can say `(DEFTYPE` *type* `FORWARD)` as a placeholder for the actual definition. If two defined types t_1 and t_2 refer to each other, and the definition of t_1 comes first, then you must say `(DEFTYPE` t_2 `FORWARD)` before that definition. In a file with many type definitions, you might as well put `FORWARD` definitions for all of them at the beginning of the file, and then not worry about circularities.

(AUGTYPE *type-desig -patches-*`)` *[Magic]*

Adds slots to a type. The *patches* are in the same format as for `DEFTYPE`, and have the same effect. This enables you to break up large `DEFTYPE`s into pieces.

4.4.2 Structures

The following complex type designators rarely occur outside of a `DEFTYPE`:

(STRUCTURE `[()]` *-type-var-list-* `[(HANDLER` *-clauses-*`)])` [*Type*]

A structured object whose slots are stored explicitly and given names. The *type-var-list* describes the slots and their types. If the `()` *flag* is present, then the structure is implemented as a vector with as many slots as there are variables in the *type-var-list*. Such a structure type is said to be *anonymous*, and membership in it is not "IS-testable." If the flag is absent, then the internal representation of the type is implementation-dependent, and membership in it is IS-testable. If the `HANDLER` is present, then it is followed by clauses of the kind accepted by `DEFCLASS` (see Section 3.4). Objects of this type will be able to handle the operations as specified. We will neglect this feature until section 4.5.

Here is an example:

```
(DEFTYPE employee
    (STRUCTURE LASTNAME - string
               PAY BENEFITS - integer
               DEPENDENTS - (LST person)))
```

This code defines an `employee` as a structure with four slots: a `string` `LASTNAME`, two `integer`s `PAY` and `BENEFITS`, and a list of `person`s `DEPENDENTS`. (Presumably `person` has been defined by the user already.) The slots are all settable as well as accessible. The constructor for this type takes four arguments, and returns a structured object with the four slots initialized to those four arguments. The type is IS-testable, because of the absence of the `()` flag.

For anonymous structures, it is officially guaranteed that `(STRUCTURE ()` *-x-*`)` expands into exactly the same thing as `(RCD` *-y-*`)` if *y* is *x* with all the slot names replaced by their types. E.g,

```
(STRUCTURE () FOO BAZ - integer Z - float)
```

is the same as

```
(RCD integer integer float).
```

This guarantee is nullified if the structure contains a `HANDLER`.

(LSTRUCTURE `[ | () | &FLAG` *symbol*`]` *-type-var-list-*`)` [*Type*]

Is like `STRUCTURE`, except that a `HANDLER` is not allowed; and it is guaranteed to be implemented with list structures in every implementation. That is, you can depend on a particular correspondence of the slots with `CAR-CDR` compositions. For instance, if we had used `LSTRUCTURE` instead of `STRUCTURE` in the previous example, that would have guaranteed that instances of the type were represented as lists of the form

`(employee` *name pay benefits dependents*`)`

`LSTRUCTURE` slightly extends the flag conventions of `STRUCTURE`. If `()` appears as the first argument, then instances of the type are anonymous list structures. Otherwise, the `CAR` of each instance is an identifying flag. If `&FLAG` *symbol* is present, the *symbol* is the flag. Otherwise, the type name is used (e.g., `employee` above).

The guaranteed correspondence between slots and `CAR-CDR` compositions is what you would expect. In particular, if the *type-var-list* has no extra layers of parentheses, then the slots become the `CAR`, `CADR`, `CADDR` etc. If the *type-var-list* ends in " &REST *slot-name*," then the last slot is a `CDDD...DDR` composition rather than a `CADD...DDR` composition.

If there are extra parentheses, then they are significant unless just one slot occurs within them. For instance, in this case:

```
(LSTRUCTURE () A - symbol
```

```
(B C - integer)
(D - float)
(E &REST F - symbol)
G - symbol)
```

We get the following correspondences:

```
A   CAR
B   CAADR
C   CADADR
D   CADDR
E   CAADDDR
F   CDADDDR
G   CADDDDR
```

Note that the parens around `B` and `C`, and around `E` and `F`, are significant, but that those around `D` are not; `D` is the `CADDR`, not the `CAADDR`.

`LSTRUCTURE` is to `LRCD` as `STRUCTURE` is to `RCD`. That is, it is guaranteed that `(LSTRUCTURE () -`*x*`-)` expands into exactly the same thing as `(LRCD -`*y*`-)` if *y* is *x* with all the slot names replaced by their types.

It is legal to use `STRUCTURE` and `LSTRUCTURE` outside a `DEFTYPE`. If you do, they will be anonymous even if the `()` flag is omitted, except in the case of an `LSTRUCTURE` with an explicit `&FLAG`..

4.4.3 Types Built on Property Lists

(**NAMED** *type-desig* [*flag* `DATA`]) [*Type*]

Designates the type of objects implemented as symbols, with the actual data stored on the property list, under the indicator *flag*. The resulting type has all the slots that *type-desig* has, but no constructor or is-tester. To add these things, use `DEFTYPE` patches, as described below.

For example, `(NAMED (LST integer) NUMS)` is a type whose elements are symbols with lists of numbers under the indicator `NUMS`.

(**SYMPLIST** *-types-and-vars-*) [*Type*]

Designates a data type consisting of symbols whose slots are implemented as good old-fashioned property-list entries. The conser and is-tester are unspecified.

4.4.4 Examples of `DEFTYPE`

Some examples of `DEFTYPE`, `STRUCTURE`, etc.:

```
(DEFTYPE employee (STRUCTURE
                    LASTNAME - string
                    PAY BENEFITS - integer
                    DEPENDENTS - (LST person))
   (GROSS - integer (E - employee)
      (WITH E (+ !>PAY !>BENEFITS))  )
   (SET LASTNAME (E - employee NEW - string)
      (IGNORE NEW)
      (EARROR LASTNAME-SETTER NIL
         "Can't set LASTNAME slot of employees"))  )
```

This defines a new type `employee` that behaves like the given `STRUCTURE`, except that it has one more slot `GROSS`, defined to be an integer, which, when accessed, returns the sum of `PAY` and `BENEFITS`; and it modifies the definition of `LASTNAME` so that the slot is read-only.

Another example:

```
(SPECDECL (PTNO* 0) - integer)

(DEFTYPE cartesiansym (SYMPLIST X Y - float)
   (CONSER (X Y - float)
      (LET ((PT (SYMBOL PT (++ PTNO*))))
         (DECL (PT - cartesiansym)
            (!= (!_X PT) X)
            (!= (!_Y PT) Y)
            PT   ))))
```

defines `cartesiansyms` to be symbols with `X` and `Y` coordinates stored as property-list entries. The user supplies a conser, so that `(MAKE cartesiansym 1.2 0.7)` will create the appropriate symbol, with a name of the form PTn.

And another:

```
(DEFTYPE part (LSTRUCTURE
                 LEN WID - float
                 NAME - string
                 SUPPLIERS - (LST supplier))
    (CONSER (LEN WID - float NAME - string PRIM - supplier)
       (MAKE ^^ LEN WID NAME (LIST PRIM))   )
    (LENGTH float *INTEGRABLE (P - part) !>P.LEN)
    (WIDTH float *INTEGRABLE (P - part) !>P.WID)
    (ALL PRIMARY-SUPPLIER - supplier (X - part)
       (CAR !>X.SUPPLERS)  )
    (SET SUPPLIERS (part X exp L)
        (!= (CDR (!_(^^ SUPPLIERS) X))
            (REMOVE1 (CAR (!_(^^ SUPPLIERS) X)) L)))   )
```

This definition describes a data type consisting of list structures whose `CAR`s are the identifying symbol `part`, and whose `CDR`s consist of a list containing the length, width, name, and suppliers in order. The first supplier is special, and is called the "primary supplier." The structures are to be consed ("`MAKE`d") by giving the length, width, name, and primary supplier. In the definition of the `CONSER`, `(MAKE ^^ ...)` is short for `(MAKE (LSTRUCTURE ...)  ...)`. In this case, we could have said `(LIST 'part ...)` instead of `(MAKE ^^ ...)`, but in other cases the use of `MAKE ^^` is the only way to refer to the procedure for constructing instances of the base type.

The next two patches define `LENGTH` and `WIDTH` to be synonyms for `LEN` and `WID`. Any reference to `(!_LENGTH X)` will be translated into `(!_LEN X)`, in-line, and similarly for `WIDTH`.

The last two patches define `PRIMARY-SUPPLIER` to be the first in the list of suppliers. Then the `SUPPLIERS` setter must be redefined not to disturb the `PRIMARY-SUPPLIER` or duplicate it. Because `ALL` precedes the symbol `PRIMARY-SUPPLIER`, `DEFTYPE` assumes that `(!= (!_PRIMARY-SUPPLIER` p`)` s`)` means `(!= (CAR (CDDDDR` p`))` s`)`.

4.5 OBJECT-ORIENTED PROGRAMMING

NISP provides rudimentary facilities for object-oriented programming, in which computing occurs by passing messages to objects. These facilities are in a state of flux, and will expand to provide inheritance, separate methods, etc., in the future. For now, there are three basic facilities for doing message passing. First, you must define the messages, which are called *operations*. Do this with `DEFOPFUNC` and `DEFOPPROC`, which are used exactly like `DEFFUNC` and `DEFPROC`, except that an operation definition may have an empty body.

Now you define the objects, in one of two ways. One is with `(MAKE-OBJECT` *clauses*`)`, which returns an object that responds to the operations as specified by the *clauses*. This is the same funoid defined in section 3.4, except that in a declaration context the clauses are allowed to specify the types of their arguments and results.

The other way to define objects is via the `HANDLER` feature of `STRUCTURE` types. Without a handler, a `STRUCTURE` instance may be thought of as a kind of vector; with the handler, it becomes something more: an object that can respond to operations as well as having slots. To create such objects, define a type *obtype* using a `STRUCTURE` with a `HANDLER`, then just execute `(MAKE` *obtype* `...)`.

Here is an example, a simple "lazy vector" package. A lazy vector is an object that responds to the operation `ELEMENT` by yielding an element, which may or may not be computed on demand.

```
(DEFTYPE lazyvec FORWARD)

(DEFOPFUNC ELEMENT - obj (V - lazyvec I - integer)
   ;; Default: Just assume V is a real vector
   (DECL (V - (VCT obj))
      (VREF V I)   ))

;; Change an element:
(DEFOPPROC SET-ELEMENT - void (V - lazyvec I - integer NEW - obj)
   (DECL (V - (VCT obj))
      (!= (VREF V I) NEW)   ))

;; Allow (!= (ELEMENT ...) ...)
(DEFSETF ELEMENT SET-ELEMENT)
```

Now for the definition of the basic type:

```
(DEFTYPE lazyvec
         (STRUCTURE ELEMENTS
                    - (VCT (EITHER (CONST *UNCOMPUTED) obj))
                    METHOD
                    - (FUN obj (integer)) ; method for computing
                                          ; elements on demand
             (HANDLER
                (ELEMENT - obj (V - lazyvec I - integer)
                   (COND ((EQ (VREF (!_ELEMENTS V) I)
                               '*UNCOMPUTED)
                          (LET ((NEW (FUNCALL (!_METHOD V) I)))
                             (!= (VREF (!_ELEMENTS V) I) NEW)
                             NEW   ))
                         (T (VREF (!_ELEMENTS V) I))   ))
                (SET-ELEMENT - void (V - lazyvec I - integer
                                     NEW - obj)
                   (!= (VREF (!_ELEMENTS V) I) NEW)   )
                (PRINT - void (V - lazyvec S - stream)
                   (OUT (TO S) "#<LAZYVEC " (!_ELEMENTS V) ">")   )))
    (CONSER (METHOD - (FUN obj (integer)))
       (MAKE ^^ (INITIALIZED-ARRAY '(10) '*UNCOMPUTED)
                ;; All lazyvecs have ten elements!
```

```
                    ;; Fixing this is left as an exercise.
                    METHOD)   ))
```

Now we can make a lazyvec by writing things like

```
(!= V1 (MAKE lazyvec (\\ (I - integer) (* 2 I) )))
```

after which `(ELEMENT V1 4)` returns 8, unless we have done something like `(!= (ELEMENT V1 4) 'FOO)` first.

But we have more flexibility than this. Suppose we wanted a function to add two lazy vectors, creating a new lazy vector that always computes values on demand, never allocating storage for them:

```
(DEFFUNC LAZYVEC+ - lazyvec (V1 V2 - lazyvec)
   (MAKE-OBJECT
      ((ELEMENT - number (ME - lazyvec I - integer)
           (IGNORE ME)
           (+ (ELEMENT V1 I) (ELEMENT V2 I))   )
       (SET-ELEMENT - void (ME - lazyvec I - integer NEW - obj)
           (IGNORE ME NEW)
           (EARROR SET-ELEMENT NIL
              "Attempt to set element " I
              " of the sum of two lazy vectors"))
       (PRINT - void (ME - lazyvec S - stream)
           (IGNORE ME)
           (OUT (TO S) "#<LAZYVEC +>")   ))))
```

Note that the first argument to a clause function of `MAKE-OBJECT` will be bound to the object itself, which we usually don't need to access.

Now `(!= V2 (LAZYVEC+ V1 V1))` returns an object that prints as `#<LAZYVEC +>`, such that `(ELEMENT V2 3)` is 12, and so forth.

Please note the distinction between operations and slot names. A `DEFTYPE` can associate slot names with an arbitrary piece of code to be executed when the slot is accessed, but this association exists only at compile time. The association between `ELEMENT` and the appropriate code in the example above is determined at run time. If the lazyvec was returned as the value of `LAZYVEC+`, then the sum is computed (after two recursive calls to `ELEMENT`); if it was created using `MAKE lazyvec`, then it is looked up or computed and stored; if neither case applies, then `VREF` is used. This added flexibility costs something, but is worth it when we want to create an abstract class of objects that are to appear uniform under a group of operations, but must be implemented in a diversity of ways.

Our example is a little misleading. `LAZYVEC+` claims to return a `lazyvec`, but of course the object it returns will not have an `!_ELEMENTS` slot, and `(IS lazyvec V2)` will be `#F` (because `V2` is not a structure of the right type). What we really want is to be able to create types that inherit properties from the type `lazyvec`, so that `LAZYVEC+` could return an object that inherited the property of being a `lazyvec`. But for now, if you want to be able to test whether something is a `lazyvec`, you should define an operation `IS-LAZYVEC` with default value `#F`, and provide clauses to make genuine lazyvecs return TRUTH.

4.6 TYPE CHECKING

NISP provides mechanisms for checking whether expressions are of allowed types in the contexts they appear in. The variable `TYPE-CHECK*` controls whether these mechanisms are on or off.

If it is `#F`, then no type checking is done. If it is `WARN`, then an error message is generated whenever an expression is encountered in a context where something of its type is not allowed. If `TYPE-CHECK*` is `BARF`, then a read-eval-print loop will be entered at that point, giving you an opportunity to correct the problem. The default is `BARF`.

The basic rule is that if an expression of type *e* is expected, and an expression of type *t* occurs, then *t* must be a subtype of *e*. Normally function declarations explain what type each of their arguments is expected to be. If `FOO` is defined using `(DEFFUNC FOO integer (X - integer L - (LST float)) ...)`, then the second argument to `FOO` is expected to be a subtype of `(LST float)`.

When a type discrepancy is detected, NISP will print an error message, and, if `TYPE-CHECK*` is `BARF`, will print a message of the form

```
While defining function name
While compiling expression of wrong type --

Expression exp cannot be coerced from type
to expected type.
...
To proceed type OK or RETURN '<correct coercion>
```

If you resume from this break point (see section 2.6) with `OK`, the type discrepancy will be ignored. If you resume and type `RETURN` *e*, then *e* will be evaluated and substituted for the *expression.*

To see more of the context surrounding the error, use `DCLSTACK`. `(DCLSTACK [`*n* `3])` shows the stack of expressions that are being processed, in a hopefully clear way. The hope is that you will see something like this:

```
(FOO (BAZ A)
     (BLECH (RACD (*>* (ZOO X))))
```

The part marked with `*>*` is the part where the error occurred. If the expression that caused the error is not a subexpression of the next guy on the stack (usually because of macro expansion), then a somewhat different format is used. Suppose `(ZOO X)` is the result of expanding `(BUFFALO X)`. Then the display would show

```
(FOO (BAZ A)
     (BLECH (RACD (*>* (BUFFALO X)))))
?>*
(*>* (ZOO X))
```

instead.

In the rest of this section, we will look at ways of understanding — and overriding — the type checker. The easiest way to override it for a given variable is to declare the variable to be of type `obj`; then it can never cause an error message. If all the formal parameters of a function are left undeclared, then they will all be implicitly declared of type `obj`. Uninitialized and undeclared local variable are also of type `obj`. (Initializing them implicitly declares them to be of the type of their initial value.)

Often a type is used repeatedly in contexts where the programmer knows it is all right, but the type system does not. The usual example is where there are two types, *super* and *sub*, such that technically *sub* is a subtype of *super*, but often you want to use a variable of type *super* where one of type *sub* is expected. If there are not too many occurrences, then you can change each occurrence of *var* to be `(BE` *sub var*`)`. This form is not executable, but simply signals the type system to treat *exp* as if it were of the designated type.

(**BE** *typ exp*) [*Other*]

The value of this expression is the value of *exp*, but the system is informed that *exp*'s value will be of type *typ* at this point. This is analogous to Common Lisp's `THE` construct. If the symbol `*` appears instead of a type, that means, "Treat *exp* as the desired type in this context," which can save some typing if the desired type is a lengthy expression.

You can use `(BE [`*type* `| *]` *exp*`)` on the left-hand side of an assignment (`!=`). .

In some cases, however, the use of a variable of type *super* where one of type *sub* is so frequent that it gets to be a real nuisance remembering to put a `BE` around it. The solution is to tell the system not to notice such violations any more, by executing

```
(DECLARE-TYPE-ACCEPTABLE 'super 'sub)
```

(DECLARE-TYPE-ACCEPTABLE *got want*) [*Function*]

Tells the type system that an expression of type *got* should never cause an error in situations requiring something of type *want*. This only works for types with atomic names.

`DECLARE-TYPE-ACCEPTABLE` should be used only when every expression of type *got* that is used in a context where *want* is expected will actually be of type *want* at run time. Be kind to your host compiler.

Some types are so vacuous that nothing should ever cause an error by appearing where they are expected.

(DECLARE-TYPE-ACCEPTABLE `'#F` *vacuous-type*) [*Function*]

Tells the system about such a type.

It is important to understand how the type-checker thinks in cases where it seems annoyingly stupid. One such case is where a constant symbol is used in a context requiring a typed expression. For example, suppose that `PS` is declared of type `procstate`, Then `(!= PS 'IDLE)` will give an error message, because `(!=` *e v*`)` requires that *v*'s type be a subtype of *e*'s. `PS` is of type `procstate`, while `'IDLE` is of type `symbol`.

One way to avoid the error is to change the assignment statement to

```
(!= PS (BE procstate 'IDLE))
```

or

```
(!= PS (BE * 'IDLE))
```

But NISP, anticipating this situation, will suppress the error message all by itself if it can verify that the datum `IDLE` is a `procstate`. If a `procstate` is nothing but a symbol from a prechosen list, then the type should have been defined thus:

```
(DEFTYPE procstate (CONST IDLE RUNNING ...))
```

The IS-tester for the `CONST` type will verify that `IDLE` is a legal value.

If `procstate`s are not this simple, then the user can make sure his own IS-tester is in effect at compile time thus:

```
(AUGTYPE procstate
         (IS (X)
            (OR (EQ X 'IDLE) ...)))
```

This tactic will work only if the IS-tester is executable at compile time, and if the file containing the `AUGTYPE` form is loaded at that time. Usually the file containing the occurrence of `'IDLE` depends on the file containing the `AUGTYPE` only at run time; wrap a `NEEDED-BY-MACROS` around the AUGTYPE or DEFTYPE for procstate in this case to make sure that the form is evaluated when its file is "slurped."

Another class of bugs derives from ambiguities in the types of list-building expressions. In some contexts an expression like (LIST 'A 5) could be thought of as building an object of type (LRCD symbol fixnum). In other contexts it could be thought of as building an object of type (LST sexp). Nisp solves the ambiguity by taking it as the latter type. Hence if an object of the former type is expected, the use of the (LST ...) expression will cause a type-check error. The solution is to use a synonym of LIST, LRECORD, which Nisp transforms into an ordinary LIST, but which always builds an object of type LRCD.

The type boolean behaves somewhat strangely. There is no IS-tester for this type, because no object would fail the test. But it is considered wrong to use an object of type stream, say, where a boolean is expected. That's because a stream can never be #F, and so you should have used '#T, an explicit boolean constant. In general, the rule is that an expression is acceptable as a boolean only if #F (i.e., ()) could be one of its values, in other words, if it is of type boolean, null, (LST *t*), (GLST *t*), or (~ *t*), or a subtype of one of these things.

Index