The PSL Users Manual

Version 4.2

by

Herbert Melenk and **Winfried Neun**

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Division of Symbolic Computing
Takustrasse 7
D–14195 Berlin-Dahlem

URL: http://www.zib.de

based on earlier Versions

by

The Utah Symbolic Computation Group
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

and

Hewlett-Packard Company
Computer Research Center

(M 3004.02)

ABSTRACT

This manual describes the primitive data structures, facilities and functions present in the Portable Standard LISP (PSL) system. It describes the implementation details and functions of interest to a PSL programmer. Except for a small number of hand-coded routines for I/O and efficient function calling, PSL is written entirely in itself, using a machine-oriented mode of PSL, called SYSLisp, to perform word, byte, and efficient integer and string operations. PSL is compiled by an enhanced version of the Portable LISP Compiler, and currently runs on many platforms, from personal computers up to super computers.

Contents

1.1 Opening Remarks 1
1.2 Scope of the Manual 1
1.2.1 Typographic Conventions within the Manual 2
1.3 Hints on Using the PSL System 2
1.3.1 Loading Optional Modules 2
1.3.2 Error and Warning Messages 3
1.4 Switches and Globals 4
1.5 Compilation Versus Interpretation 4
2.1 Data Types 4
2.1.1 Data Types and Structures Supported in PSL 4
2.1.2 Predicates Useful with Data Types 9
2.1.3 Converting Data Types 13
3.1 Numbers and Arithmetic Functions 16
3.1.1 Big Integers 16
3.1.2 Conversion Between Integers and Floats 17
3.1.3 Arithmetic Operators 17
3.1.4 Arithmetic Functions 20
3.1.5 Functions for Numeric Comparison 23
3.1.6 Bit Operations 24
3.1.7 Various Mathematical Functions 25
4.1 Introduction 33
4.2 Identifiers and the Id Hash Table 35
4.2.1 Identifier Functions 35
4.3 Property List Functions 37
4.3.1 Functions for Flagging Ids 40
4.3.2 Direct Access to the Property Cell 40
4.4 Value Cell Functions 41
4.5 System Global Variables, Switches and Other "Hooks" 45
4.5.1 Introduction 45
4.5.2 Setting Switches 45
4.5.3 Special Global Variables 47
4.5.4 Special Put Indicators 47
4.5.5 Special Flag Indicators 48

5.1 Introduction to Lists and Pairs . . . 49
5.2 Basic Functions on Pairs . . . 50
5.3 Functions for Manipulating Lists . . . 53
5.3.1 Membership and Length of Lists . . . 55
5.3.2 Deleting Elements of Lists . . . 59
5.3.3 List Reversal . . . 61
5.3.4 Functions for Sorting . . . 61
5.4 Functions for Building and Searching A-Lists . . . 62
5.5 Substitutions . . . 64
6.1 Characters . . . 65
6.2 Strings . . . 70
6.2.1 String Creation and Copying . . . 70
6.2.2 About the Basic String Operations . . . 71
6.2.3 The Operations . . . 71
6.3 Common LISP String Functions . . . 72
6.3.1 String comparison: . . . 72
6.3.2 String Concatenation: . . . 74
6.3.3 Transformation of Strings: . . . 74
6.3.4 Type Conversion: . . . 75
6.3.5 Other: . . . 76
6.3.6 Substring Comparison . . . 76
6.3.7 Searching for Strings . . . 77
6.3.8 Reading and Writing Strings . . . 77
7.1 Introduction . . . 81
7.1.1 Conditionals . . . 81
7.2 Case and Selectq Statements . . . 83
7.3 Sequencing Evaluation . . . 85
7.3.1 Iteration . . . 88
7.3.2 Mapping Functions . . . 98
7.4 Non-Local Exits . . . 102
8.1 Function Definition in PSL . . . 107
8.1.1 Function Types . . . 107
8.1.2 Notes on Code Pointers . . . 108
8.1.3 Functions Useful in Function Definition . . . 108
8.1.4 Function Definition in LISP Syntax . . . 110
8.1.5 BackQuote . . . 114
8.1.6 MacroExpand . . . 116
8.1.7 Function Type Predicates . . . 117
8.2 Wrappers . . . 118
8.2.1 Notes on Writing Wrappers . . . 118
8.2.2 Exported Functions . . . 120
8.2.3 Examples . . . 121

8.3 Variables and Bindings . . . 124
8.3.1 Binding Type Declaration . . . 125
8.3.2 Binding Type Predicates . . . 126
8.4 User Binding Functions . . . 126
9.1 Evaluator Functions Eval and Apply . . . 129
9.2 Support Functions for Eval and Apply . . . 133
9.3 Special Evaluator Functions, Quote and Function . . . 134
9.4 Support Functions for Macro Evaluation . . . 135
10.1 Introduction . . . 137
10.1.1 Organization of this Chapter . . . 137
10.2 Printed Representation of LISP Objects . . . 138
10.3 Functions for Printing . . . 141
10.3.1 Basic Printing . . . 141
10.3.2 Whitespace Printing Functions . . . 142
10.3.3 Formatted Printing . . . 142
10.3.4 The Fundamental Printing Function . . . 144
10.3.5 Additional Printing Functions . . . 144
10.3.6 Printing Status and Mode . . . 145
10.4 Functions for Reading . . . 147
10.4.1 Reading S-Expressions . . . 147
10.4.2 Reading Single Characters . . . 148
10.4.3 Reading Tokens . . . 149
10.4.4 Reading Entire Lines . . . 150
10.4.5 Read Macros . . . 150
10.4.6 Terminal Interaction . . . 151
10.4.7 Input Status and Mode . . . 151
10.5 File System Interface: Open and Close . . . 152
10.6 Loading Modules . . . 153
10.7 Reading Files into PSL . . . 155
10.8 About I/O Channels . . . 157
10.9 I/O to and from Lists and Strings . . . 158
10.10 Generalized Input/Output Streams . . . 159
10.10.1 Using the "Special" Form of Open . . . 160
10.11 Scan Table Internals . . . 161
10.12 Scan Table Utility Functions . . . 163
10.13 Binary I/O Functions . . . 163
11.1 Introduction . . . 167
11.2 The General Purpose Top Loop Function . . . 167
11.3 Changing the Default Top Level Function . . . 170
12.1 Introduction . . . 173
12.2 The Basic Error Functions . . . 173
12.3 Basic Error Handlers . . . 175

12.4 Break Loop . . . 176
12.5 Details on the Break Loop . . . 179
12.6 Some Convenient Error Calls . . . 179
13.1 The Debug Module . . . 183
13.1.1 Overview of Functionality . . . 183
13.1.2 Using Break and Trace . . . 184
13.1.3 Sample Session . . . 185
13.1.4 Redefining a Broken or Traced Function . . . 187
14.1 Simulating a Stack . . . 189
14.2 Ring Buffers . . . 189
14.3 Word Vector Operations . . . 190
15.1 Introduction . . . 191
15.2 Compiling Files . . . 191
15.2.1 Order of Functions for Compilation . . . 193
15.3 Compiling Functions into Memory . . . 193
15.4 Compiler Errors and Warnings . . . 194
15.5 Differences between Compiled and Interpreted Code . . . 195
15.6 Constant Declaration . . . 198
15.7 Fluid and Global Declarations . . . 200
15.8 Control Over the Time When Something is Done . . . 200
15.9 Switches That Control the Compiler . . . 201
15.10Conditional Compilation . . . 204
15.11Implementation Details . . . 204
16.1 Exiting PSL . . . 207
16.2 Saving an Executable PSL . . . 207
16.3 Init Files . . . 208
16.4 Miscellaneous Functions . . . 208
16.5 Garbage Collection . . . 209
17.1 Introduction . . . 211
17.1.1 RCREF - Cross Reference Generator for PSL Files . . . 211
17.1.2 Restrictions . . . 212
17.1.3 Usage . . . 212
17.1.4 Options . . . 213
17.2 Scanalyzer . . . 213
17.2.1 Introduction . . . 213
17.2.2 Philosophy . . . 214
17.2.3 Functions . . . 215
17.2.4 Environment Arguments . . . 216
17.2.5 Analysis Hooks . . . 216
17.2.6 Properties . . . 216
17.2.7 Information Made Available By Scanalyzer . . . 217
17.2.8 Expansion And Preprocessing Hooks . . . 217

17.2.9 Cross Reference Support . . . 217
18.1 Introduction . . . 221
18.2 Prettyprinting Files and Data . . . 221
18.3 Formats . . . 222
18.4 Dispatch . . . 224
18.5 Specifying Formats . . . 225
19.1 Introduction . . . 229
19.1.1 Terminology . . . 229
19.2 Creating Objects . . . 230
19.2.1 Methods . . . 231
19.2.2 Protection of Objects . . . 231
19.3 Reference Information . . . 231
19.3.1 Loading Objects . . . 232
19.3.2 Flavor Definition . . . 232
19.3.3 Method Definition . . . 234
19.3.4 Object Creation . . . 235
19.3.5 Message Sending . . . 237
19.3.6 Printing Objects . . . 238
19.3.7 Useful Functions on Objects . . . 239
19.4 Using Inheritance . . . 241
19.4.1 Warning on Inheritance Usage . . . 243
19.4.2 Using SELF and MYSELF with Inheritance . . . 244
19.4.3 Inheritance and Initialization . . . 244
19.4.4 Making Changes to Inherited Code . . . 244
19.5 Debugging Information . . . 245
20.1 Vectors . . . 247
20.1.1 About the Basic Operations on Vectors . . . 249
20.1.2 The Operations . . . 249
20.1.3 Built-in Operations on Vectors . . . 250
20.2 Word Vectors . . . 250
20.3 General X-Vector Operations . . . 251
21.1 Calling the Command Shell . . . 253
21.2 The Working Directory . . . 253
21.3 Invoking Pipes . . . 254
21.3.1 Pipes under Unix . . . 254
21.3.2 Pipes under MS/DOS . . . 255
21.3.3 Pipes under MS Windows . . . 255
21.4 Socket Interface (Unix only) . . . 255
21.5 Shared Memory Interface (Unix only) . . . 255
21.6 Miscellaneous Features . . . 257
26.1 Installation . . . 267
26.1.1 Reading the tape . . . 268

26.1.2 Reading the tape for IBM RS/6000 268
26.1.3 Reading Diskettes for LINUX 386 268
26.1.4 Customizing Makefiles and scripts 269
26.1.5 Printing Documentation 271
26.2 New unexec procedure, Image model 271
26.3 Dynamic configuration of Heap Size and Binding Stack 272
26.4 Size of Address Space 272
26.5 Arbitrary Precision Integer Support 273
26.6 Monitoring of Performance 273
26.6.1 SPY (Unix only) 273
26.6.2 Qualified timing 274
26.6.3 Qualified counting 275
26.7 Compiler Modifications 276
26.8 Disassembler 276
26.9 More unsupported software 277
26.9.1 Oload not supported 277
26.9.2 Portable Common Lisp Subset (PCLS) not supported . 277
26.10Shared Memory Interface (Unix only) 278
26.11Socket interface (Unix only) 279
26.12Pipe Interface (Unix only) 280
26.13Mapping of LISP Addresses to C addresses 280

0

Introduction

1.1 Opening Remarks

This document describes PSL (Portable Standard Lisp [1]), a portable, "small" and fast LISP developed originally at the University of Utah and the Hewlett-Packard Company. The present version of PSL is supported by the Konrad–Zuse-Zentrum Berlin. PSL is upward-compatible with Standard Lisp [2]. In most cases, Standard Lisp did not commit itself to specific implementation details (since it was to be compatible with a portion of "most" LISPs). PSL is more specific and provides many more functions than described in that report.
The goals of PSL include:

- Providing implementation tools for LISP that can be used to implement a variety of LISP-like systems, including mini-Lisps embedded in other language systems.

- Effectively supporting the algebra system on a number of machines.

- Studying the utility of a LISP-based systems language for other applications (such as CAGD or VLSI design) in which Lisp code provides efficiency comparable to that of C or BCPL, yet enjoys the interactive program development and debugging environment of LISP.

1.2 Scope of the Manual

This manual is intended to describe the syntax, semantics, and implementation of PSL. While we have attempted to make it comprehensive, it is not intended for use as a primer. Some prior exposure to LISP will prove to be very helpful.

[1] "LSP" backwards!
[2] J. B. Marti, A. C. Hearn, M. L. Griss, C. Griss: The Standard Lisp Report

1.2.1 Typographic Conventions within the Manual

A large portion of this manual is devoted to descriptions of the functions that make up PSL. For each function there is a prototypical header line. Each argument is given a name, followed by the type of argument expected. If more than one type is allowed then the choices will be enclosed within the brackets {, and }. For example, the following header shows a function named vector-fetch which accepts two arguments. The first V, is a vector, the second I, is an integer. The term expr refers to a type of function.

(vector-fetch V:vector I:integer): any *expr*
The value returned is the i'th element of the vector V.

Within compiled code, some function calls are replaced by a sequence of instructions whose execution is equivalent to a call on the function. Functions of this type will have a note saying open-compiled next to the function type. Some functions accept an arbitrary number of arguments. The header for these functions shows a single argument enclosed in square brackets, indicating that zero or more occurrences of that argument are allowed.

(and [U:form]): extra-boolean *fexpr*
And is a function which accepts zero or more arguments each of which may be any form.

In some cases, LISP code is given in the function documentation as the function's definition. This code is given to clarify the semantics of the function, it is not a copy of the actual PSL definition.

1.3 Hints on Using the PSL System

The following sub-sections collect a few miscellaneous notes that are further expanded on elsewhere. They are provided here simply to get you started.

1.3.1 Loading Optional Modules

Certain modules are not present in the "kernel" or "bare-psl" system, but can be loaded as options. Some of these optional modules will be automatically loaded when first referenced. Other modules may be explicitly loaded by the user, or included by the installer when building the PSL core image. Optional modules can be loaded by executing the following.

```
(load modulename)
```

When a module is loaded its name is added to a list referenced by the global variable options*. The modules which make up the BARE-PSL kernel are not included in this list. An application of load will be aborted if the argument is found in the options* list. If it is necessary to load a module a second time use reload, do not attempt to alter the value of options*.

1.3.2 Error and Warning Messages

Many functions detect and signal appropriate errors ; in many cases, an error message is printed. The error conditions are given as part of a function's definition in the manual. An error message is preceded by five stars (*); a warning message is preceded by three. For example, most primitive functions check the type of their arguments and display an error message if an argument is incorrect. The type mismatch error mentions the function in which the error was detected, gives the expected type, and prints the actual value passed.

```
1 lisp> (car "STR")
***** An attempt was made to do CAR on '"STR"', which is not a pair
```

The switch **usermode** is used to distinguish between functions which comprise PSL and functions defined by the user. When the value of this switch is t then the name of each user defined function is flagged **user**. When a function is about to be defined, if *usermode is t, there is another function associated with the name, and the name is not flagged lose then the following prompt will be displayed.

```
Do you really want to redefine the system function 'NAME'?
```

You are expected to respond with `YES`, `Y`, `NO`, `N`, or `B`. A response of `B` will result in a break loop (see Chapter 16). After quitting the break loop you should respond with `YES`, `Y`, `NO`, or `N` (see yesp in Chapter 15). Unless you consider yourself an expert it is dangerous to give an affirmative response. It is best to simply give your function a different name. If the switch **redefmsg** is set to t then whenever a function is redefined the message

```
*** Function 'NAME' has been redefined
```

will be printed.
If an identifier is flagged LOSE then it cannot be assigned a functional definition.

```
*** 'NAME' has not been defined, because it is flagged LOSE
```

1.4 Switches and Globals

Generally, the name of an identifier which represents a switch will start with a "*". For example, *redefmsg and *usermode are switches. The value of a switch is modified with the functions on and off. Note that in the example below, the prefix "*" is omitted when using on and off.

```
1 lisp> (on redefmsg)
nil
2 lisp> *redefmsg
t
```

The name of an identifier which represents a global ends with "*". options* is an example of a global.

1.5 Compilation Versus Interpretation

PSL uses both compiled and interpreted code. If compiled, a function usually executes faster and is smaller. However, there are some semantic differences of which the user should be aware. For example, some recursive functions are made non-recursive, and certain functions are open-compiled. A call to an open-compiled function is replaced, on compilation, by a series of online instructions instead of just being a reference to another function. Functions compiled open may not do as much type checking.

```
1 lisp> (de list-first (p) (car p))
list-first
2 lisp> (list-first "STR")
***** An attempt was made to do CAR on `"STR', which is not a pair
3 lisp> (compile '(list-first))
nil
4 lisp> (list-first "STR")
#<Unknown f7000002>
```

To avoid this the user would have to add code which checks the type of P before car is applied.

2.1 Data Types

2.1.1 Data Types and Structures Supported in PSL

Data Types

In contrast to many programming languages, type declarations are not needed in PSL. Data objects contain information about their type. Some functions,

like equal, are "generic" in that the result they return depends on the types of the arguments.

```
1 lisp> (equal "sextuped" "sextuped")
T
2 lisp> (equal [bencolin beef] [bencolin beef])
T
```

For the purposes of input and output, an appropriate notation is used for each type of data object used in PSL. For example, double quotes are used to delimit the characters of a string. For a full discussion on syntax see Chapter 12.
The basic data types supported in PSL and a brief indication of their representations are described below.

integer The integers are also called "fixed" numbers. The magnitude of integers is essentially unrestricted if the "big number" module, **zbig**, is loaded. The notation for integers is a sequence of digits in an appropriate radix (radix 10 is the default, which can be overridden by a radi prefix, such as 2#, 8#, 16# etc). There are three internal representations of integers, chosen to suit the implementation:

inum A signed number fitting into info. Inums do not require dynamic storage and are represented in the same form as machine integers.

fixnum A full-word signed integer, allocated in the heap.

bignum A signed integer of arbitrary precision, allocated as a vector of system integers. These integers may be not integers for PSL, because they do not fit into info. Beware. Bignums are currently not installed by default, to use them load the ZBIG module.

float A floating point number, allocated in the heap. The precision of floats is determined solely by the implementation. Usually, the floating numbers are equivalent to system 'doubles', (64 bits). The notation for a float is a sequence of digits with the addition of a single floating point (.) and an optional exponent (E <integer>). (No spaces may occur between the point and the digits). Radix 10 is used for representing the mantissa and the exponent of floating point numbers.

id An identifier (or id) is an item whose info field points to a five-item structure containing the print name, property cell, value cell, function cell, and package cell. This structure is contained in the id space. The notation for an id is its print name, an alphanumeric character sequence. One always refers to a particular id by giving its print name. When presented with an appropriate print name, the PSL reader will find a unique id to associate with it. See Chapters 4 and 12 for more information on ids and their syntax. The ids t and nil are considered special in that it is not possible for the user to redefine their value cells.

pair A primitive two-item structure which has a left and right part. A notation called dot-notation is used, with the form: (<left-part> . <right-part>). The <left-part> is known as the car portion and the <right-part> as the cdr portion. The parts may be any item. (Spaces are used to resolve ambiguity with floats; see Chapter 12).

vector A primitive uniform structure of items; an integer index is used to access random values in the structure. The individual elements of a vector may be any item. Access to vectors is by means of functions for indexing, sub-vector extraction and concatenation, defined in Section 7.1. In the notation for vectors, the elements of a vector are surrounded by square brackets: [item-0 item-1 ... item-n].

string A packed vector (or byte vector) of characters; the elements are small integers representing the ASCII codes for the characters (usually inums). The elements may be accessed by indexing, substring and concatenation functions, defined in Chapter 6. String notation consists of a series of characters enclosed in double quotes, as in "THIS IS A STRING". A quote is included by doubling it, as in "HE SAID, ""LISP""". A string may be input across the end of a line but a warning will be given unless the switch **eolinstringok** is non-nil (see Chapter 12).

w-vector A vector of machine-sized words, used to implement such things as fixnums, bignums, etc. The elements are not considered to be items, and are not examined by the garbage collector.

byte-vector A vector of bytes. Internally a byte-vector is the same as

a string, but it is printed differently as a vector of integers instead of characters.

code-pointer This item is used to refer to the entry point of compiled functions (exprs, fexprs, macros, etc.), permitting compiled functions to be renamed, passed around anonymously, etc. New code-pointers are created by the loader (lap, fasl) and associated functions. They can be printed; the printing function prints the number of arguments expected as well as the entry point. The value appears in the convention of the implementation (e.g. #<Code A N>, where A is the number of arguments and N is the entry point).

Other Notational Conventions

Certain functional arguments can be any of a number of types. For convenience, we give these commonly used sets a name. We refer to these sets as "classes" of primitive data types. In addition to the types described above and the names for classes of types given below, we use the following conventions in the manual. {XXX, YYY} indicates that either data type XXX or data type YYY will do. {XXX}-{YYY} indicates that any object of type XXX can be used except those of type YYY; in this case, YYY is a subset of XXX. For example, {integer, float} indicates that either an integer or a float is acceptable; {any}-{vector} means any type except a vector.

any Any of the types given above. S-expression is another term for any. All PSL entities have some value unless an error occurs during evaluation.

atom The class any-pair.

boolean The class of global variables t, nil, or their respective values, t, nil. (See Section 4.6).

character Integers in the range of 0 to 255 without 128 representing ASCII character codes. These are distinct from single-character ids.

constant The class of integer, float, string, vector, code-pointer. A constant evaluates to itself (see the definition of eval in Chapter 11).

extra-boolean Any value in the system. Anything that is not nil has the boolean interpretation t.

ftype The set of ids (expr, fexpr, macro, and nexpr), which represent definable function types. The ftype is only an attribute of identifiers, and is not associated with either executable code code-pointers or lambda expressions.

io-channel A small integer representing an io channel (see Chapter 12 for a complete discussion of io-channels).

number The class of integer, float.

x-vector Any kind of vector; i.e. a string, vector, w-vector, or word.

Undefined An implementation-dependent value returned by some low-level functions; i.e. the user should not depend on this value.

None Returned A notational convenience used to indicate control functions that do not return directly to the calling point, and hence do not return a value (for example, see the function go in Chapter 8).

Structures

Structures are entities created using pairs. Lists are structures very commonly required as parameters to functions. If a list of homogeneous entities is required by a function, this class is denoted by xxx-list, in which xxx is the name of a class of primitives or structures. Thus a list of ids is an id-list, a list of integers is an integer-list, and so on.

list A list is recursively defined as nil or the pair (any . list). A special notation called list-notation is used to represent lists. List-notation eliminates the extra parentheses and dots required by dot-notation, as illustrated below. List-notation and dot-notation may be mixed, as shown in the second example.

```
DOT NOTATION                LIST NOTATION
(A . (B . (C . NIL)))         (A B C)
((A . (B)) . C)               ((A B) . C)
```

Note: () is an alternate input representation of nil.

a-list An a-list, or association list, is a list in which each element is a pair, the car part being a key associated with the value in the cdr part.

form A form is an S-expression (any) which is legally acceptable to eval; that is, it is syntactically and semantically accepted by the interpreter or the compiler.

lambda A lambda expression must be of the following form, the square brackets are used to indicate zero or more occurances of an expression.

```
(LAMBDA <parameters> [<form>])
```

The expression `<parameters>` is a list of ids which represents the formal parameters or the body (the sequence of `<form>`s). The evaluation of the body takes place as if the `<form>`s were enclosed within a progn.

function A lambda expression or a code-pointer, the function type is assumed to be expr. This means that the arguments will be evaluated, and that the number of arguments must agree with the number of parameters.

2.1.2 Predicates Useful with Data Types

Most functions in this Section return t if the condition defined is met and nil if it is not. Exceptions are noted. Defined are type-checking functions and elementary comparisons.

Functions for Testing Equality

Functions for testing equality are listed below. For other functions comparing arithmetic values see Chapter 3.

(eq U:any V:any): boolean *open-compiled expr*
Returns t if U points to the same object as V, i.e. if they are identical items. Eq is not a reliable comparison between numeric arguments. This function should only be used in special circumstances. Normally, equality should be tested with equal, described below.

(eqn U:any V:any): boolean *expr*
Returns t if U and V are eq or if U and V are numbers and have the same value and type.

(equal U:any V:any): boolean *expr*

Returns t if U and V are the same. A usually valid heuristic is that if two objects look the same if printed with the function print, they are equal. Equal is open-compiled as eq if one argument is known to be an atom.

```
(de equal (u v)
  (cond ((and (pairp u) (pairp v))
         (and (equal (car u) (car v))
              (equal (cdr u) (cdr v))))
        ((and (stringp u) (stringp v))
         (string= u v))
        ((and (vectorp u) (vectorp v))
         (vector-equal u v))
        (t (eqn u v))))
```

```
1 lisp> (setq x '(lisa) y x)
(LISA)
2 lisp> (eq x y)
T
3 lisp> (eq x '(lisa))
NIL
4 lisp> (equal x '(lisa))
T
5 lisp> (eq 1.0 1.0)
NIL
6 lisp> (eqn 1.0 1.0)
T
7 lisp> (equal 0 0.0)
NIL
```

(neq U:any V:any): boolean *macro*

(not (equal U V)).

(ne U:any V:any): boolean *open-compiled expr*

(not (eq U V)).

(eqstr U:any V:any): boolean *expr*
Compare two strings, for exact (case sensitive) equality. The function string-equal (which is defined in the STRINGS module), is not sensitive to case. Eqstr returns t if U and V are eq or if U and V are equal strings.

(eqcar U:any V:any): boolean *expr*
Tests whether (eq (car U) V)). If the first argument is not a pair, eqcar returns nil.

Predicates for Testing the Type of an Object

(atom U:any): boolean *open-compiled expr*
Returns t if U is not a pair.

(codep U:any): boolean *open-compiled expr*
Returns t if U is a code-pointer.

(constantp U:any): boolean *expr*
Returns t if U is a constant (that is, neither a pair nor an id). Note that vectors are considered constants.

(fixp U:any): boolean *open-compiled expr*
Returns t if U is an integer. If BIG is loaded, this function also returns t for bignums.

(floatp U:any): boolean *open-compiled expr*
Returns t if U is a float.

(idp U:any): boolean *open-compiled expr*
Returns t if U is an id.

(null U:any): boolean *open-compiled expr*
Returns t if U is nil. This is exactly the same function as not, defined in Section 2.2.3. Both are available solely to increase readability.

(numberp U:any): boolean *open-compiled expr*
Returns t if U is a number (integer or float).

(pairp U:any): boolean *open-compiled expr*
Returns t if U is a pair.

(stringp U:any): boolean *open-compiled expr*
Returns t if U is a string.

(vectorp U:any): boolean *open-compiled expr*
Returns t if U is a vector.

Boolean Functions

Boolean functions return nil for false; anything non-nil is taken to be true, although a conventional way of representing truth is as t. Note that t always evaluates to itself, its value cannot be redefined. Nil may also be represented as (). As a matter of style, () should be used to refer to an empty list. The Boolean functions and, or, and not can be applied to an object of any type. And and or may also be used as control structures (see Section 8.2 for more information).

Since PSL treats any value which is non-nil as a representation for true, there is no clear distinction between an arbitrary function and a boolean function. However, the three functions presented here are by far the most useful in constructing more complex tests from simple predicates.

(not U:any): boolean *open-compiled expr*
Returns t if U is nil. This is exactly the same function as null, defined in Section 2.2.2. Both are available solely to increase readability.

(and [U:form]): extra-boolean *open-compiled fexpr*
And evaluates each U until a value of nil is found or the end of the list is encountered. If a non-nil value is the last value, it is returned; otherwise nil is returned. Note that and called with zero arguments returns t. In the example which follows and is used to select the first element of a list, if the call on pairp returns nil then car will not be applied.

```
1 lisp> ((lambda (p) (and (pairp p) (car p)))  '(robin))
robin
```

(or [U:form]): extra-boolean *open-compiled fexpr*
U is any number of expressions which are evaluated in order of their appearance. If one is found to be non-nil, it is returned as the value of or. If all are nil, nil is returned. Note that if or is called with zero arguments, it returns nil. The following function defines a predicate for numbers.

```
(de number-p (n)
  (or (fixp n) (floatp n)))
```

2.1.3 Converting Data Types

The following functions are used in converting data items from one type to another. They are grouped according to the type returned. Numeric types may be converted using functions such as fix and float, described in Section 3.2.

(intern U:id,string): id *expr*
Returns an identifier from the symbol table (also called the id-hash-table). When the PSL reader reads a sequence of characters which notate an id, it will apply intern to the string of characters. Therefore, it generally does not make sense to apply intern to an id. Intern will search the symbol table for an id whose print name matches U. If the search is successful then the matching id is returned. Otherwise a new id will be entered into the symbol table and a reference to it will be returned. If U has more than the maximum number of characters permitted by the implementation, an error will be signalled.

```
***** Too many characters to INTERN
```

The id which is returned from an application of intern to a string will have the string as its print name. Most identifiers have lowercase print names (even though you may type in lower case letters), but interning "ABC" yields an id with a lower case print name.

```
1 lisp> (eq (intern "abc") 'abc)
NIL
```

The maximum number of characters in any token is system dependent, around 5000 can be expected to be allowed.

(newid S:string): id *expr*

Allocates a new identifier and sets its print name to the string S. The identifier is not added to the symbol table (an identifier which does not appear in the symbol table is said to be uninterned). The string is not copied.

```
1 lisp> (setq new (newid "NEWONE"))
NEWONE
2 lisp> (eq new 'newone)
nil
```

If one refers directly to an identifier (for example 'newone), the reader will apply intern to the string of characters it has read ("NEWONE"). In the example, the identifier created by the call on newid is different from the one created by the reader when it read 'newone.

(int2id I:integer): id *expr*

Converts an integer to an id; this refers to the I'th id in the id space. Since 0 ... 255 correspond to ASCII characters, int2id with an argument in this range converts an ASCII code to the corresponding single character id. The id NIL is always found by `(int2id 128)`.

(id2int D:id): integer *expr*

Returns the id space position of D as a LISP integer.

(id2string D:id): string *expr*

Get name from id space. Id2string returns the print name of its argument as a string. This is not a copy, so destructive operations should not be performed on the result. PSL uses an escape convention for notating identifiers which contain special characters. Any character which follows the character ! is considered to be an alphabetic character. In the example, notice that the character ! does not appear in the result.

```
1 lisp> (id2string 'is-!%)
"is-%"
```

(string2list S:string): inum-list *expr*

Creates a list of length (add1 (size S)), converting the ASCII characters into small integers.

```
1 lisp> (string2list "STRING")
(83 84 82 73 78 71)
```

(list2string L:inum-list): string *expr*

Allocates a string of the same size as L, and converts small integers into characters according to their ASCII code. An integer outside the range of 0 ... 127 will result in an error.

```
***** An attempt was made to do LISP2CHAR on 'N' which is not a character
```

```
1 lisp> (list2string '(83 84 82 73 78 71))
"STRING"
```

(string [I:inum]): string *nexpr*

Creates and returns a string containing each of the small integers

```
1 lisp> (string 83 84 82 73 78 71)
"STRING"
```

(vector [U:any]): vector *nexpr*

Creates and returns a vector containing all the Us given.

```
1 lisp> (vector 83 84 82 73 78 71)
[83 84 82 73 78 71]
```

(vector2string V:vector): string *expr*

Pack the small integers in the vector into a string of the same size, using the integers as ASCII values. An integer outside the range of 0 ... 255 will result in an error.

```
1 lisp> (vector2string [83 84 82 73 78 71])
"STRING"
```

(string2vector S:string): vector *expr*

Unpack the string into a vector of the same size. The elements of the vector are small integers, representing the ASCII values of the characters in S.

```
1 lisp> (string2vector "VECTOR")
[V E C T O R]
```

(vector2list V:vector): list *expr*

Create a list of the same size as V, the elements are copied in a left to right order.

```
1 lisp> (vector2list [L I S T])
(L I S T)
```

(list2vector L:list): vector *expr*

Copy the elements of the list into a vector of the same size.

```
1 lisp> (list2vector '(V E C T O R))
[V E C T O R]
```

3.1 Numbers and Arithmetic Functions

Most of the arithmetic functions in PSL expect numbers as arguments. In all cases an error occurs if the parameter to an arithmetic function is not a number:

```
*****  Non-numeric argument in arithmetic
```

Exceptions to the rule are noted.

The underlying machine arithmetic requires parameters to be either all integers or all floats. If a function receives mixed types of arguments, integers are converted to floats before arithmetic operations are performed. The range of numbers which can be represented by an integer is different than that represented by a float. Because of this difference, a conversion is not always possible; an unsuccessful attempt to convert may cause an error to be signalled.
The **mathlib** package contains some useful mathematical functions.

3.1.1 Big Integers

Loading the ZBIG [1] module redefines the basic arithmetic operations, including the logical operations, to permit arbitrary precision (or "bignum") integer operations.

[1] ZIB version of big integers, the optimal version for the particular architecture

3.1.2 Conversion Between Integers and Floats

The conversions mentioned above can be done explicitly by the following functions. Other functions which alter types can be found in Section 2.3.

(fix U:number): integer *expr*
Returns the integer which corresponds to the truncated value of U. The result of conversion must retain all significant portions of U. If U is an integer it is returned unchanged.

```
1 lisp> (fix 2.1)
2
2 lisp> (fix -2.1)
-2
```

(float U:number): float *expr*
The float corresponding to the value of the argument U is returned. Some of the least significant digits of an integer may be lost due to the implementation of float. If U a float then it will be returned unchanged. If U is too large to represent in float an error occurs.

```
***** Argument to FLOAT is too large
```

3.1.3 Arithmetic Operators

This section describes arithmetic functions in the library module **numeric-ops**. The names of these functions are based upon mathematical notation.

There is a switch called **fast-integers** whose value has an effect on the compilation of forms which contain applications of the functions described here. The documentation assumes that the switch fast-integers is nil. When this switch is non-nil the compiler will generate code which is very efficient. However, it is assumed that arguments and results will be integers in the inum range. If this assumption is violated then at best your code will not generate correct results, you may actually damage the PSL system.

Common LISP operators

(= X:number Y:number): boolean *expr*
Numeric Equal. True if the two arguments are numbers of the same type and same value. Unlike the Common LISP operator, no type coercion is done, no error is signalled if one or both arguments are non-numeric, and only two arguments are permitted. Instead, it is merely incorrect to supply a non-numeric argument.

(/= X:number Y:number): boolean *expr*
Numeric Not Equal. Nil if X and Y are numbers of equal type and value; t if X and Y are numbers of unequal type or value. It is incorrect to supply a non-numeric argument.

(< X:number Y:number): boolean *expr*
Numeric Less Than. True if X is less than Y, regardless of type. An error is signalled if either argument is not numeric.

(> X:number Y:number): boolean *expr*
Numeric Greater Than. True if X is greater than Y, regardless of type. An error is signalled if either argument is not numeric.

(<= X:number Y:number): boolean *expr*
Numeric Less Than or Equal. True if X is less than or equal to Y, regardless of numeric type. An error is signalled if either argument is not numeric.

(>= X:number Y:number): boolean *expr*
Numeric Greater Than or Equal. True if X is greater than or equal to Y, regardless of numeric type. An error is signalled if either argument is not numeric.

(+ [N:number]): number *macro*
Numeric Addition. The value returned is the sum of all the arguments. The arguments may be of any numeric type. An error is signalled if any argument is not numeric. If supplied no arguments, the value is 0.

(– N:number [N:number]): number *macro*
Numeric Minus or Subtraction. If given one argument, returns the negative of that argument. If given more than one argument, returns the result of successively subtracting succeeding arguments from the first argument. Signals an error if no arguments are supplied or if any argument is non-numeric.

(∗ [N:number]): number *macro*
Numeric Multiplication. The value returned is the product of all the arguments. The arguments may be of any numeric type. An error is signalled if any argument is not numeric. If supplied no arguments, the value is 1.

(/ N:number [N:number]): number *macro*
Numeric Reciprocal or Division. If given one argument, returns the reciprocal of that argument. If given more than one argument, returns the result of successively dividing succeeding arguments from the first argument. Signals an error if no arguments are supplied or if any argument is non-numeric.

Additional Operators

(∼= X:number Y:number): number *expr*
Numeric Not Equal. Same as /=.

(// X:integer Y:integer): integer *expr*
Integer Remainder. Same as remainder.

(∼ X:integer): integer *expr*
Integer Bitwise Logical Not. Same as lnot.

(& X:integer Y:integer): integer *expr*
Integer Bitwise Logical And. Same as land.

(| X:integer Y:integer): integer *expr*
Integer Bitwise Logical Or. Same as lor.

(∧ X:integer Y:integer): integer *expr*
Integer Bitwise Logical Xor. Same as lxor.

(<< X:integer Y:integer): integer *expr*
Integer Bitwise Logical Left Shift. Same as lshift.

(>> X:integer Y:integer): integer *expr*
Integer Bitwise Logical Right Shift. Same as (lshift X (minus Y)).

3.1.4 Arithmetic Functions

The functions described below handle arithmetic operations. Please note the remarks at the beginning of this Chapter regarding the mixing of argument types.

(abs U:number): number *expr*
Returns the absolute value of its argument.

(add1 U:number): number *expr*
Returns the value of U plus 1; the returned value is of the same type as U (integer or float).

(decr U:form [Xi:number]): number *macro*
This function is defined in the **useful** module. With only one argument, this is equivalent to

```
(setf u (sub1 u))
```

With multiple arguments, it is equivalent to

```
(setf u (difference u (plus x1 ... xn)))
```

```
1 lisp> (setq y '(1 5 7))
(1 5 7)
2 lisp> (decr (car y))
0
3 lisp> y
(0 5 7)
4 lisp> (decr (cadr y) 3 4)
-2
5 lisp> y
(0 -2 7)
```

(difference U:number V:number): number *expr*
The value of U - V is returned.

(divide U:number V:number): pair *expr*
The pair (quotient . remainder) is returned, as if the quotient part was computed by the quotient function and the remainder by the remainder function. An error occurs if division by zero is attempted:

```
***** Attempt to divide by 0 in Divide
```

(expt U:number V:integer): number *expr*
Returns U raised to the V power. A float U to an integer power V does not have V changed to a float before exponentiation.

(incr U:form [Xi:number]): number *macro*
Part of the **useful** package. With only one argument, this is equivalent to

```
(setf u (add1 u))
```

With multiple arguments it is equivalent to

```
(setf u (plus u x1 ... xn))
```

(minus U:number): number *expr*
Returns -U.

(plus [U:number]): number *macro*
Forms the sum of all its arguments. Plus may be called with only one argument. In this case it returns its argument. If plus is called with no arguments, it returns zero.

(plus2 U:number V:number): number *expr*
Returns the sum of U and V.

(quotient U:number V:number): number *expr*
The quotient of U divided by V is returned. Division of two positive or two negative integers is conventional. If both U and V are integers and exactly one of them is negative, the value returned is truncated toward 0. If either argument is a float, a float is returned which is exact within the implemented precision of floats. An error occurs if division by zero is attempted:

```
***** Attempt to divide by 0 in QUOTIENT
```

(recip U:number): float *expr*
Recip converts U to a float if necessary, and then finds the inverse using the function quotient.

(remainder U:integer V:integer): integer *expr*
(- U (* V (fix (/ U (float V)))))

```
(remainder 13 4)    =  1
(remainder -13 4)   = -1
(remainder 13 -4)   =  1
(remainder -13 -4)  = -1
```

(sub1 U:number): number *expr*
Returns the value of U minus 1. If U is a float, the value returned is U minus 1.0.

(times [U:number]): number *macro*
Returns the product of all its arguments. Times may be called with only one argument. In this case it returns the value of its argument. If times is called with no arguments, it returns 1.

(times2 U:number V:number): number *expr*
Returns the product of U and V.

3.1.5 Functions for Numeric Comparison

The following functions compare the values of their arguments. Functions which test for equality and inequality are documented in Section 2.2.1.

(geq U:any V:any): boolean *expr*
Equivalent to (¿= U V).

(greaterp U:number V:number): boolean *expr*
Equivalent to (¿ U V).

(leq U:number V:number): boolean *expr*
Equivalent to (¡= U V).

(lessp U:number V:number): boolean *expr*
Equivalent to (¡ U V).

(max [U:number]): number *macro*
Returns the largest of the values in U (numeric maximum). If two or more values are the same, the first is returned.

(max2 U:number V:number): number *expr*
Returns the larger of U and V. If U and V are of the same value, U is returned (U and V might be of different types).

(min [U:number]): number *macro*
Returns the smallest (numeric minimum), of the values in U. If two or more values are the same, the first of these is returned.

(min2 U:number V:number): number *expr*
Returns the smaller of its arguments. If U and V are the same value, U is returned (U and V might be of different types).

(minusp U:any): boolean *expr*
Returns t if U is a number and less than 0. The return value is nil if U is not a number, or if U is a positive number.

(onep U:any): boolean *expr*
Returns t if U is a number and has the value 1 or 1.0. Returns nil otherwise.

(zerop U:any): boolean *expr*
Returns t if U is a number and has the value 0 or 0.0. Returns nil otherwise.

3.1.6 Bit Operations

The functions described in this section operate on the binary representation of the integers given as arguments. The returned value is an integer.

(land U:integer V:integer): integer *expr*
Bitwise or logical and. Each bit of the result is independently determined from the corresponding bits of the operands.

(lor U:integer V:integer): integer *expr*
Bitwise or logical or. Each bit of the result is independently determined from corresponding bits of the operands. This is an inclusive or, the value of (lor 1 1) is 1.

(lnot U:integer): integer *expr*
Logical not. Defined as (- -U 1) so that it works for bignums as if they were 2's complement.

(lxor U:integer V:integer): integer *expr*
Bitwise or logical exclusive or, the value of (lxor 1 1) is 0. Each bit of the result is independently determined from the corresponding bits of the operands.

(lshift N:integer K:integer): integer *expr*
Shifts N to the left by K bits. The effect is similar to multiplying by 2 to the K power. Negative values are acceptable for K, and cause a right shift (in the usual manner). Lshift is a logical shift, so right shifts do not resemble division by a power of 2.

3.1.7 Various Mathematical Functions

The optionally loadable **mathlib** module defines several commonly used mathematical functions. Some effort has been made to be compatible with Common Lisp. When reading the examples, note that the precision of the results depend on the machine being used.

(ceiling X:number): integer *expr*
Returns the smallest integer greater than or equal to X. For example:

```
1 lisp> (ceiling 2.1)
3
2 lisp> (ceiling -2.1)
-2
```

(floor X:number): integer *expr*
Returns the largest integer less than or equal to X. (Note that this differs from the fix function.)

```
1 lisp> (floor 2.1)
2
2 lisp> (floor -2.1)
-3
3 lisp> (fix -2.1)
-2
```

(round X:number): integer *expr*
Returns the nearest integer to X. If the fractional part of X is 0.5 then the smaller integer is returned.

```
(de round (x)
  (if (fixp x) x (floor (plus x 0.5))))

1 lisp> (round 2.5)
3
2 lisp> (round -2.5)
-2
```

(transfersign S:number VAL:number): number *expr*

Transfers the sign of S to VAL.

```
(de transfersign (s val)
  (if (>= s 0) (abs val) (minus (abs val))))
```

(mod M:integer N:integer): integer *expr*

(- U (* V (floor (/ U (float V)))))

```
(mod 13 4)    =  1
(mod -13 4)   =  3
(mod 13 -4)   = -3
(mod -13 -4)  = -1
```

(degreestoradians X:number): number *expr*

Returns an angle in radians given an angle in degrees.

```
1 lisp> (degreestoradians 180)
3.1415926
```

(radianstodegrees X:number): number *expr*

Returns an angle in degrees given an angle in radians.

```
1 lisp> (radianstodegrees 3.1415926)
180.0
```

(radianstodms X:number): list *expr*

Given an angle X in radians, returns a list of three integers, which represent the degrees, minutes, and seconds.

```
1 lisp> (radianstodms 1.0)
(57 17 45)
```

(dmstoradians Degs:number Mins:number Secs:number): number
expr
Returns an angle in radians, given three arguments representing an angle in degrees minutes and seconds.

```
1 lisp> (dmstoradians 57 17 45)
1.0000009
2 lisp> (dmstoradians 180 0 0)
3.1415926
```

(degreestodms X:number): list *expr*
Given an angle X in degrees, returns a list of three integers giving the angle in (Degrees Minutes Seconds).

(dmstodegrees Degs:number Mins:number Secs:number): number
expr
Returns an angle in degrees, given three arguments representing an angle in degrees minutes and seconds.

(sin X:number): number *expr*
Returns the sine of X, an angle in radians.

(sind X:number): number *expr*
Returns the sine of X, an angle in degrees.

(cos X:number): number *expr*
Returns the cosine of X, an angle in radians.

(cosd X:number): number *expr*
Returns the cosine of X, an angle in degrees.

(tan X:number): number *expr*
Returns the tangent of X, an angle in radians.

(tand X:number): number *expr*
Returns the tangent of X, an angle in degrees.

(cot X:number): number *expr*
Returns the cotangent of X, an angle in radians.

(cotd X:number): number *expr*
Returns the cotangent of X, an angle in degrees.

(sec X:number): number *expr*
Returns the secant of X, an angle in radians.

(secd X:number): number *expr*
Returns the secant of X, an angle in degrees.

(csc X:number): number *expr*
Returns the cosecant of X, an angle in radians.

(cscd X:number): number *expr*
Returns the cosecant of X, an angle in degrees.

(asin X:number): number *expr*
Returns the arc sine, as an angle in radians, of X.

```
(eqn (sin (asin X)) X)
```

(asind X:number): number *expr*
Returns the arc sine, as an angle in degrees, of X.

(acos X:number): number *expr*
Returns the arc cosine, as an angle in radians, of X.

```
(eqn (cos (acos X)) X)
```

(acosd X:number): number *expr*
Returns the arc cosine, as an angle in degrees, of X.

(atan X:number): number *expr*
Returns the arc tangent, as an angle in radians, of X.

```
(eqn (tan (atan X)) X)
```

(atand X:number): number *expr*
Returns the arc tangent, as an angle in degrees, of X.

(atan2 Y:number X:number): number *expr*
Returns an angle in radians corresponding to the angle between the X axis and the vector [X Y]. Note that Y is the first argument.

```
1 lisp> (atan2 0 -1)
3.1415927
```

(atan2d Y:number X:number): number *expr*
Returns an angle in degrees corresponding to the angle between the X axis and the vector [X Y].

```
1 lisp> (atan2d -1 1)
315.0
```

(acot X:number): number *expr*
Returns the arc cotangent, as an angle in radians, of X.

```
(eqn (cot (acot X)) X)
```

(acotd X:number): number *expr*
Returns the arc cotangent, as an angle in degrees, of X.

(asec X:number): number *expr*
Returns the arc secant, as an angle in radians, of X.

```
(eqn (sec (asec X)) X)
```

(asecd X:number): number *expr*
Returns the arc secant, as an angle in degrees, of X.

(acsc X:number): number *expr*
Returns the arc cosecant, as an angle in radians, of X.

```
(eqn (csc (acsc X)) X)
```

(acscd X:number): number *expr*
Returns the arc cosecant, as an angle in degrees, of X.

(sqrt X:number): number *expr*
Returns the square root of X.

(exp X:number): number *expr*
Returns the exponential of X.

(log X:number): number *expr*
Returns the natural (base e) logarithm of X. note that (log (log (exp X)) is equal to X.

(log2 X:number): number *expr*
Returns the base two logarithm of X.

(log10 X:number): number *expr*
Returns the base ten logarithm of X.

(random N:integer): integer *expr*
Returns a pseudo-random number uniformly selected from the range 0 ... (sub1 N).

The random number generator uses a linear congruential method.

randomseed = Initially: set from time *global*
To get a reproducible sequence of random numbers you should assign one (or some other small number) to the fluid variable randomseed.

(factorial N:integer): integer *expr*
The factorial function is defined as follows

```
1. The factorial of 0 is 1.

2. The factorial of N is N times the factorial of N-1.
```

```
(de factorial (n)
  (if (zerop n) 1 (* n (factorial (sub1 n)))))
```

Identifiers

4.1 Introduction

Ids or identifiers can be used in a number of different ways. Every id has a name, called its print name. Given an id, one can obtain its name in the form of a string. Conversely, given the name of an id as a string one can obtain the id itself.

Ids have a component called the property list. This list consists of pairs and ids. A pair contains two elements, the first is the name of the property, the second is the value associated with that property. An id on the property list represents a flag.

Each id also references a package, see Chapter [packages] for more information on the package system.

In PSL an id can be used simultaneously as a variable and as a name for a function. Aside from the functions described in this chapter, there are additional functions for dealing with the values associated with an id.

An id can be referenced simply by writing its name. If the name consists only of uppercase alphabetic characters, digits, or a subset of the special characters (listed below), and if the name of the id cannot be mistaken for a number, then the id can be notated by the sequence of characters in its name.

```
$+ - $ & * / : ; | < = > ? ^ _ { } ~ @$
```

An id may have uppercase letters, lowercase letters, or both in its print name. The PSL reader normally (i.e. version 4.2 and above) converts uppercase letters to the corresponding lowercase letters when reading ids. Therefore, most of the time case makes no difference when notating ids.
The conversion of letters is controlled by the functions **input-case** and **output-case**.

(input-case modus): modus *expr*

If modus equals lower, characters on input are converted to the corresponding lowercase characters. If modus equals raise, characters are raised during input. A modus NIL leaves characters unchanged. The former modus is returned. The default is `lower`.

(output-case modus): modus *expr*

If modus equals lower, characters on output are converted to the corresponding lowercase characters. If modus equals raise, characters are raised during output. A modus NIL leaves characters unchanged. The former modus is returned. The default is NIL.

Ids are kept in a table which is called the symbol table (or id-hash-table). Two ids are considered different if their corresponding print names are different. For example, the id whose name is "that" is different from the id whose name is "THAT". The ids which name PSL functions have lowercase names. The reason you can type such names with uppercase letters is that the reader is converting uppercase letters to lowercase by default.

```
1 lisp> (Add1 2)
3
2 lisp> (input-case nil)
lower
3 lisp> (Add1 2)
***** `!Add1' is an undefined function
```

If the user tries to use a PSL function name for a function he is defining a warning message appears.

```
Do you really want to redefine the system function `NAME'?
(Y or N)
```

If the user responds "Y", his definition replaces the current definition. (See Chapter 9 for a description of the switch *usermode which controls the printing of this message).

There is an escape convention for notating an id whose name contains special characters. Any character which follows a ! is considered to be an ordinary character. In addition to lowercase letters, the following characters are considered special:

```
! " % ' ( ) . [ ] ` , # |
```

If it is not clear from the printed output, this set of characters includes both quote and accent grave. Note that if any character wthin a name is preceded by a !, then the name will not be interpreted as a number.

```
SUSAN     % three ways to notate the same id
susan
SuSan
+$        % an id without alphabetic characters
1+        % an id whose first character is a digit
+1        % this is a number
x^2+y^2   % an id which looks like an expression
!9        % the id whose name is "9", not the number 9
```

4.2 Identifiers and the Id Hash Table

The method used by PSL to retrieve information about an id makes use of a symbol table. PSL uses a technique called hashing to implement this table (id hash table is another name for the symbol table).

The process of putting an id into the symbol table is called interning. The PSL reader interns ids as they are read. Consider what happens after the name of an id is read. The symbol table is examined to see if it contains an identifier with the same name. If there is a match then a reference to the matching id is returned. Otherwise, a new id is created, it is added to the symbol table, and a reference to it is returned.

4.2.1 Identifier Functions

The following functions deal with identifiers and the symbol table.

(gensym): id *expr*
An id is created which is not interned. Since it is not interned it is not eq to any other id. The id is derived from a string of the form "G0000". The numeric suffix is incremented upon each call to gensym.

(interngensym): id *expr*
Similar to gensym but returns an interned id.

(stringgensym string): *expr*
Similar to gensym but returns a string of the form "L0000" instead of an id.

(remob U:id): id *expr*
If U has been interned in the symbol table then it is removed. The values associated with U will not be affected. U is returned. It is not possible to remove from the symbol table an identifier whose name consists of a single character.

```
1 lisp> (setq what (intern "THIS"))
THIS
2 lisp> (set what "SOMETHING")
"SOMETHING"
3 lisp>  % Remove the id whose name is "THIS".
3 lisp> (remob what)
THIS
4 lisp>  % Although the id whose name is "THIS"
has been removed from
4 lisp>
% the symbol table, it remains in
%existance and its value % cell
4 lisp>  % is still defined as "SOMETHING".
4 lisp> (eval what)
"SOMETHING"
```

(newid U:string): id *expr*
Creates an uninterned identifier with the specified name. The string is used as the print name without being copied. See section 2.3 for the full definition. This function makes it possible to create a number of distinct ids which have the same name. To illustrate the use of this function, the implementation of a package system (see Chapter [packages]), requires a function like newid.

(internp U:id,string): boolean *expr*
Returns t if U is interned.

(mapobl FNAME:function): Undefined *expr*
Mapobl applies function FNAME to each interned id. The following expression will print each id which is flagged global. Note that there should be only one formal parameter to FNAME.

```
(mapobl '(lambda (item) (if (flagp item 'global)
(print item))))
```

Find

These functions take a string or id as an argument, and scan the symbol table to collect a list of ids whose names contain a prefix or suffix which matches the argument. These functions are defined in the library module **find**.

(findprefix KEY:id, string): id-list *expr*
Each interned id whose name contains a prefix which matches the KEY is added to the result. The ids are sorted alphabetically. The expression

```
(findprefix '*)
```

will return a list of all of the interned ids whose name begins with *.

(findsuffix KEY:id, string): id-list *expr*
Each interned id whose name contains a suffix which matches the KEY is added to the result. The ids are sorted alphabetically. The expression

```
(findsuffix "STRING")
```

will return a list of all of the interned ids whose name ends with STRING.

4.3 Property List Functions

A property list is used to associate an id with a set of entities; those entities are called flags if their use associates a boolean value with the id, and properties if the id is to be associated with an arbitrary attribute.

(put U:id IND:id PROP:any): any *expr*
The indicator IND with the property PROP is placed on the property list of U. If the action of put occurs, the value of PROP is returned. If either U or IND are not ids then a type mismatch error occurs.

```
***** An attempt was made to do PUT on 'U', which is not
an identifier
```

The definition of a property will cause the previous definition to be lost.

(get U:id IND:id): any *expr*

Returns the property associated with indicator IND from the property list of U. If U does not have indicator IND, nil is returned. Get returns nil if U is not an id.

(deflist U:list IND:id): list *expr*

U is a list in which each element is a two-element list: (ID:id PROP:any). Each id in U has the indicator IND with property PROP placed on its property list by the function put function. The value of deflist is a list of the first elements of each two-element list.

```
1 lisp> (deflist '((plus2 'two)
                   (plus 'many))
                 'no-operands)
(plus2 plus)
2 lisp> (get 'plus 'no-operands)
many
```

(remprop U:id IND:id): any *expr*

Removes the property with indicator IND from the property list of U. Returns the removed property or nil if there was no such indicator.

(rempropl U:id-list IND:id): nil *expr*

Removes the property IND from all of the ids in U.

The following example is intended to illustrate the idea of data driven programming. We define a function called simplify which will simplify symbolic algebraic expressions. These expressions are represented as lists. To begin, there will be only one operator (plus), and operands may be integers, variables or an application of plus. Prefix notation is used. The addition of variable x and 3 would be represented as (plus x 3). The first version of simplify will certainly do the job.

```
(de simplify (expression)
  (cond ((atom expression) expression)
```

```
        ((eq (first expression 'plus)
             (add-simplify expression))
        (t expression)))
```

However, as we add operands it will become necessary to redefine simplify. A better approach is to allow the operator to specify the information on how to simplify the expression.

```
(de simplify (expression)
  (cond ((atom expression) expression)
        (t (apply (get (first expression) 'simplify)
                  (ncons expression)))))


(put 'plus 'simplify 'add-simplify)
```

This version will not have to be rewritten when a new operator is added. For example, if the operator times is added then we only need to define a function called times-simplify and attach its name to the property list of times under the indicator simplify. We can design add-simplify in a similar fashion. Using this approach we will be able to accomodate numbers other than integers.

```
  (de add-simplify (expression)
    (let ((left (second expression))
          (right (third expression)))
      (cond ((zerop left) right)
            ((zerop right) left)
            ((and (numberp left)(numberp right))
             (let ((new (common-type left right)))
               (apply (get (data-type (first  new))'add-op) new)))
            (t (list (first expression)
                     (simplify left)
                     (simplify right))))))


(put 'integer 'add-op 'plus2)


1 lisp> (simplify '(plus (plus 1 8) (plus x 0)))
(plus 9 X)
```

4.3.1 Functions for Flagging Ids

(flag U:id-list V:id): nil *expr*
Flag flags each id in U with V; that is, the effect of flag is that for each id X in U, (flagp X V) has the value t. Both V and all of the elements of U must be identifiers or a type mismatch error occurs. The id V will appear on the property list of each id in U. However, flags cannot be accessed, placed on, or removed from property lists using the normal property list functions get, put, and remprop. Note that if an error occurs during execution of flag, then some of the ids in U may be flagged with V, and others may not be. The statement below causes the flag Lose to be placed on the property lists of the ids x and y.

```
(flag '(x y) 'lose)
```

(flagp U:id V:id): boolean *expr*
Returns t if U has been flagged with V; otherwise returns nil. Returns NIL if either U or V is not an id.

(remflag U:id-list V:id): nil *expr*
Removes the flag V from the property list of each member of the list U. Both V and all the elements of U must be ids or the type mismatch error occurs.

(flag1 U:id V:any): Undefined *expr*
The identifier U is flagged V. The effect is to add V to the property list of U.

(remflag1 U:id V:any): Undefined *expr*
The identifier U is no longer flagged V. The effect is to removed V from the property list of U.

4.3.2 Direct Access to the Property Cell

Use of the following functions can destroy the integrity of the property list. Since PSL uses properties at a low level, care should be taken in the use of these functions.

(prop U:id): any *expr*
Returns the property list of U.

(setprop U:id L:any): L:any *expr*
Store item L as the property list of U.

4.4 Value Cell Functions

The contents of the value cell of an id is usually accessed by eval (Chapter 11) or valuecell (below) and changed by setq, setf or sometimes set.

(setq [VARIABLE:id VALUE:any]): any *fexpr*
The value of each VARIABLE is set to the corresponding value of VALUE. Each argument VALUE is evaluated, each argument VARIABLE is not evaluated. It is not true that

```
(setq variable value)
```

is equivalent to

```
(set 'variable value)
```

Where setq may be used to set any type of variable (fluid, global or local) the function set is restricted to fluid and global variables.

(set EXP:id VALUE:any): any *expr*
Set is used to define the value cell of fluid and global identifiers. An error occurs if EXP does not evaluate to an identifier.

```
***** An attempt was made to do SET on 'EXP', which  is  not
an identifier
```

If EXP evaluates to t or nil an error occurs.

```
***** Cannot change T or NIL
```

(desetq U:any V:any): V:any *macro*
This function is part of the USEFUL package. Desetq is a destructuring setq. That is, the first argument is a list whose elements are ids. The value of each id is set to the corresponding element in the second argument. For example, evaluation of

```
(desetq (A (B) . C) '((1) (2) (3) 4))
```

defines the value of A to be (1), B to be 2, and C to ((3) 4).

(psetq [VARIABLE:id VALUE:any]): Undefined *macro*

This function is defined in the USEFUL package. Psetq is very similar to setq. The difference is that with psetq each VALUE is evaluated before any assignment is made.

```
1 lisp> (setq a 'same  b a)
SAME
2 lisp> (eq a b)
T
3 lisp> (psetq a 'other  b a)
OTHER
4 lisp> (eq a b)
NIL
```

(setf [LHS:form RHS:any]): RHS:any *macro*

The ability to assign values to ids allows us to think of ids as variables. We can generalize this notion of variable. For example, a list can be thought of as a collection of anonymous variables. Usually there are seperate access and update functions for each kind of generalized variable. For example the function cdr accesses the cdr of a pair, the function rplacd updates it. However, we can think of a call on an access function as a reference to a storage location. Just as we consider the mention of an id to be a reference to its value, (cdr pair) can be thought of as the name for the cdr for some pair. Rather than having to remember two functions for each kind of generalized variable (rplacd corresponds to cdr), we can adopt a uniform syntax for updating storage locations using the setf macro.

The application of setf can take on any one of the following forms:

`(setf id data)` expands into `(setq id data)`
`(setf (eval form) data)` expands into `(set form data)`

The same effect is obtained by substituting value in place of eval.

`(setf (car pair) data)` expands into `(rplaca pair data)`
`(setf (cdr pair) data)` expands into `(rplacd pair data)`
`(setf (getv vector index) data)` expands into
`(putv vector index data)`

`(setf (indx form index) data)` expands into
`(setindx form index data)`
`(setf (sub form start size) data)` expands into
`(setsub form start size data)`
`(setf (nth pair index) data)` expands into an expression similar to
`(rplaca (pnth pair index) data)`

If the first argument to setf is a macro then it will be expanded before setf is. For example, if first is defined as

`(ds first (p) (car p))` then

`(setf (first p) data)` is equivalent to `(setf (car p) data)`

The USEFUL module contains an expanded version of setf. The basic definition of setf is not consistent with that of setq. The value returned from an application of setq is always the value assigned. For example, the expression

`(setf (car '(a b)) 'd)` expands into `(rplaca '(a b) 'd)`

The value returned after evaluating this second expression is (d b). The extended version of setf will always return the value assigned.

An application of the extended version of setf will accept the additional following forms:

`(setf (cons left right) pair)` will expand into an expression similar to
`(progn (setf left (car pair)) (setf right (cdr pair)))`

`(setf (cXYr pair) data)` expands into an expression similar to
`(rplacX (cYr pair) data)`

where X is either a or d and Y is either a, d, aa, ..., or ddd

`(setf (flagp id name) data)` expands into an expression similar to `(flag (list id) name)`, if data is non-nil otherwise
`(remflag (list id) name)`
`(setf (get id name) data)` expands into `(put id name data)`
`(setf (getd name) data)` expands into an expression similar to
`(putd name (car data) (cdr data))`

This expansion assumes that data is similar to an expression which would be returned by a call on getd. If data is a code-pointer or a lambda expression then 'expr is used in place of (car data).

`(setf (lastcar pair) data)` expands into an expression similar to `(rplaca (lastpair pair) data)`.

`(setf (list a b c ...) pair)`
the expansion of this expression is very similar to the expansion of
`(desetq (a b c ...) pair)`

`(setf (pnth pair index) data)` expands into an expression similar to
`(rplacd (pnth pair (sub1 index)) data)`

`(setf (vector al bl cl ...) [ar br cr ...])`
expands into an expression similar to
`(progn (setf al ar)(setf bl br)(setf cl cr)...)`

The setf function is extensible to permit additional operators on the left hand side. If there is an assign-op property on the property list of the operator then the value of that property (either a lambda expression or the name of a function) is used to build the expansion of the macro. The effect is similar to

```
(apply (get op 'assign-op) (append (cdr lhs) (list rhs)))
```

The property setf-expand is searched for when their is no assign-op property. If there is such a property its value is applied to the two arguments passed to setf. The effect is similar to

```
(apply op (list lhs rhs))
```

If the left hand side operator is flagged as setf-safe, then it is assumed that the expansion of the macro will yeild an expression which will return the value of the right hand side. Otherwise the expansion will take one of the forms listed below. Within the second expansion, references to RHS are replaced with references to VAR. The second form is used when RHS is a list, the assumption being that it is effecient to evaluate an application only once.

```
(progn expansion rhs)

(let ((var rhs))
  expansion
  var)
```

(psetf [LHS:form RHS:any]): Undefined *macro*
This function is defined in the USEFUL package. Psetf is very similar to setf. The difference is that with psetf each RHS is evaluated before any assignment is made.

(makeunbound U:id): Undefined *expr*
U is made an unbound identifier, that is to say it will no longer have a value. This function should be applied to fluid identifiers only.

(valuecell U:id): any *expr*
Safe access to the value cell of an id. If U is not an id a type mismatch error occurs. If U is an unbound id, an unbound id error occurs. Otherwise the current value of U is returned. This function should be applied to fluid identifiers only.

(unboundp U:id): boolean *expr*
Returns t is U is unbound. This function should be applied to fluid identifiers only.

4.5 System Global Variables, Switches and Other "Hooks"

4.5.1 Introduction

A number of global variables provide global control of the PSL system, or reference values which are constant throughout execution. Certain options are controlled by switches, variables which have a value of either t or nil. For example, the value of *verboseload controls the display of messages when files are loaded. The values of other global variables are not restricted to be boolean. For example, the value of outputbase* is the radix in which numbers are printed. PSL uses the convention that the name of a global variables which is a switch begins with "*". The names of other global variables end with "*".

4.5.2 Setting Switches

Strictly speaking, NAME is a switch and *NAME is a corresponding global variable that assumes a value of t or nil. Both NAME and *NAME are

loosely referred to as switches elsewhere in the manual.

The functions on and off functions are used to change the values of the identifiers associated with switches. Some switches contain an s-expression on their property lists under the indicator simpfg [1]. The s-expression has the form of a cond list:

```
((t (action-for-on)) (nil (action-for-off)))
```

If the simpfg indicator is present, then the on and off functions also evaluate the appropriate action in the s-expression.

(on [U:id]): Undefined *macro*
For each switch U, the associated identifier *U is set to nil. If the clause (t (action-for-on) is found under the indicator simpfg on U then the expression action-for-on will be evaluated.

(off [U:id]): Undefined *macro*
For each switch U, the associated identifier *U is set to nil. If the clause (nil (action-for-off) is found under the indicator simpfg on U then the expression action-for-off will be evaluated.

The switch fast-integers is used by the compiler when arithmetic expressions are compiled. There are definitions of numeric operators which do not check the types of their arguments in order to reduce execution time. Evaluation of

```
(get 'fast-integers 'simpfg)
```

returns

```
((t (enable-fast-numeric-operators))
 (nil (disable-fast-numeric-operators))))
```

Evaluation of (on fast-integers) will result in *fast-integers being set to t and evalution of the function enable-fast-numeric-operators.

[1] The name simpfg comes from its introduction in the Reduce algebra system, where it was used to specify various simplifications to be performed as various switches were turned on or off.

4.5.3 Special Global Variables

nil = [Initially: nil] *global*
A special global variable whose value cannot be modified by set or setq.

t = [Initially: t] *global*
A special global variable whose value cannot be modified by set or setq.

4.5.4 Special Put Indicators

Some actions search the property list of relevant ids for the following indicators.

breakfunction Associates a function to be run with an id typed in a break loop (see Chapter 16). For example, q is used to exit from a break loop and (get 'q 'breakfunction) returns breakquit.

type PSL uses the property type to indicate whether a function is a fexpr, macro, or nexpr. If this property is absent, expr is assumed. For example, (get 'and 'type) returns fexpr.

vartype PSL uses the property vartype to indicate whether an identifier is of type global or fluid.

```
1 lisp > (fluid '(mary))
nil
2 lisp > (get 'mary 'vartype)
fluid
```

***lambdalink** The interpreter looks under *lambdalink for a lambda expression when a compiled definition cannot be found.

```
1 lisp > (de list-first (p) (car p))
list-first
2 lisp > (get 'list-first '*lambdalink)
(lambda (p) (car p))
```

The compiler and loader use a number of other indicators, see Chapter 19.

4.5.5 Special Flag Indicators

eval, ignore These flags are used primarily to control the evaluation of expressions during compilation (for more information see Chapter 19).

lose The function putd is used to associate function definitions with ids. Its application is aborted if the id has been flagged lose.

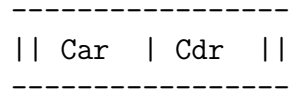
```
*** 'NAME' has not been defined, because it is flagged LOSE
```

user If *usermode is t, when a function is defined its name will be flagged user. This is used to distinguish user defined functions from system functions (see Chapter 9 for more information).

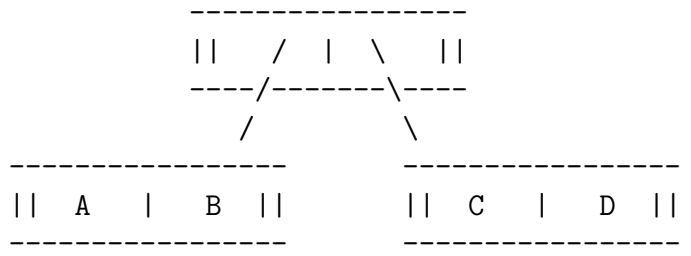
List Structure

5.1 Introduction to Lists and Pairs

The pair is a fundamental PSL data type, and is one of the major attractions of LISP programming. A pair consists of a two-item structure. In PSL the first element is called the car and the second the cdr; in other LISPs, the physical relationship of the parts may be different. An illustration of the tree structure is given below as a box diagram; the car and the cdr are each represented as a portion of the box.



As an example, a tree written as ((A . B) . (C . D)) in dot-notation is drawn below as a box diagram.



The box diagrams are tedious to draw, so dot-notation is normally used. Note that a space is left on each side of the . to ensure that pairs are not confused with floats.
A list is a special case of a dotted pair structure. A list is either

1. NIL
2. A dotted pair whose car is an expression and whose cdr is a list.

List notation in general is a lot easier to read than the equivalent dotted pair notation.

```
(A . NIL)                = (A)
(A . B)                  = (A . B)
(NIL . NIL)              = (NIL)
(A . (B . NIL))          = (A B)
((A . NIL) . NIL)        = ((A))
((A . NIL) . (B . NIL)) = ((A) B)
(A . (B . C))            = (A B . C)
```

The following is an algorithm for writing a dotted pair structure in list notation.

1. Set a pointer Q to the beginning of the dotted pair structure and write a left parenthesis.

2. Write the list notation for the data structure pointed to by the car of Q and reset Q to the cdr of Q.

3. If Q is now the null pointer, then write a right parenthesis; otherwise, write a space, and if Q is an atom, write a period, a space, Q's name, and a right parenthesis; otherwise write a space and go to step 2.

5.2 Basic Functions on Pairs

The following are elementary functions on pairs. All functions in this Chapter which require pairs as parameters signal a type mismatch error if the parameter given is not a pair.

(cons U:any V:any): pair *expr*
Returns a pair which is not eq to anything else and has U as its car part and V as its cdr part.

(car U:pair): any *open-compiled expr*
The left part of the pair U is returned. Note that applications of car are open compiled, a compiled application of car will not verify that its argument is a pair. The function SafeCar may be used in place of car. The definitions of these two functions are identical, the difference between them is that safecar is not open compiled. For interpreted applications, the car of nil is nil and if U is something other that a pair or nil the following error will result.

```
*****  An attempt was made to do CAR on 'U', which is not a pair
```

(cdr U:pair): any *open-compiled expr*
The left part of the pair U is returned. Note that applications of cdr are open compiled, a compiled application of cdr will not verify that its argument is a pair. The function SafeCdr may be used in place of cdr. The definitions of these two functions are identical, the difference between them is that safecdr is not open compiled. For interpreted applications, the cdr of nil is nil and if U is something other that a pair or nil the following error will result.

```
***** An attempt was made to do CDR on 'U', which is not a pair
```

The composites of car and cdr are supported up to four levels.

```
              Car                                 Cdr
      Caar            Cdar            Cadr            Cddr
  Caaar  Cdaar    Cadar  Cddar    Caadr  Cdadr    Caddr  Cdddr
 Caaaar  Cadaar  Caadar  Caddar  Caaadr  Cadadr  Caaddr  Cadddr
 Cdaaar  Cddaar  Cdadar  Cdddar  Cdaadr  Cddadr  Cdaddr  Cddddr
```

These are all exprs of one argument. Applications of these functions are generally open compiled. An example of their use is that (cddar p) is equivalent to (cdr (cdr (car p))). For interpreted applications, a type mismatch error occurs if the argument does not possess the specified component.
As an alternative to employing chains of car and cdr to obscure depths, particularly in extracting elements of a list, consider the use of the functions first, second, third, fourth, rest, nth, and pnth.

(ncons U:any): pair *expr*
Equivalent to (cons u nil).

(xcons LEFT:any RIGHT:any): pair *expr*
Equivalent to (cons RIGHT LEFT), this function is useful for generating efficient list building code for the compiler.

(copy X:any): any *expr*
This function returns a copy of X. While each pair is copied, atomic elements (for example ids, strings, and vectors) are not. See totalcopy in section 7.5. Note that copy is recursive and will not terminate if its argument is a circular list.

```
(de copy (u)
  (if (pairp u)
    (cons (copy (car u)) (copy (cdr u)))
    u))
```

```
1 lisp> (setq p '("AKU" (charlie)))
("AKU" (CHARLIE))
2 lisp> (setq q (copy p))
("AKU" (CHARLIE))
3 lisp> (eq p q)
NIL
4 lisp> (eq (first p) (first q))
T
5 lisp> (eq (third p) (third q))
NIL
```

See Chapter 6 for other relevant functions.

The following functions are known as "destructive" functions, because they change the structure of the pair given as their argument, and consequently change the structure of any object containing the pair. They are frequently used to make code more efficient. For example append will copy its first argument whereas nconc will not. These functions are also used to build structures which share sub-structure. It is possible to create self referential structures with these functions. This can create havoc with normal printing and list traversal functions.

(rplaca U:pair V:any): pair *open-compiled expr*
The car of the pair U is replaced by V, the modified pair U is returned.
A type mismatch error occurs if U is not a pair

```
1 lisp> (setq fruit '(orange apple))
(ORANGE APPLE)
2 lisp> (setq food (cons 'cheese fruit))
(CHEESE ORANGE APPLE)
3 lisp> (rplaca fruit 'peach)
(PEACH APPLE)
4 lisp> food
(CHEESE PEACH APPLE)
```

(rplacd U:pair V:any): pair *open-compiled expr*
The cdr of the pair U is replaced by V, the modified pair U is returned.
A type mismatch error occurs if U is not a pair.

```
1 lisp> (setq pair '(left))
(LEFT)
2 lisp> (progn (rplacd pair 'right) pair)
(LEFT . RIGHT)
```

(rplacw A:pair B:pair): pair *expr*
Replaces the whole pair: the car of A is replaced with the car of B, and the cdr of A with the cdr of B. The modified pair A is returned.

5.3 Functions for Manipulating Lists

The following functions are meant for the special pairs which are lists, as described in Section 5.1. An argument which is not a list could give unexpected results. For example, length is used to determine the number of top level elements in a list.

```
1 lisp> (length '(a b c))
3
2 lisp> (length '(a b . c))
2
```

Selecting List Elements

(first L:pair): any *macro*
A synonym for car.

(second L:pair): any *macro*
A synonym for cadr.

(third L:pair): any *macro*
A synonym for caddr.

(fourth L:pair): any *macro*
A synonym for cadddr.

(rest L:pair): any *macro*
A synonym for cdr.

(lastpair L:pair): any *expr*
Returns the last pair of a L. It is often useful to think of this as a pointer to the last element for use with destructive functions such as rplaca. If L is not a pair then a type mismatch error occurs.

```
(de lastpair (l)
  (if (or (atom l) (atom (cdr l)))
    l
    (lastpair (cdr l))))
```

(lastcar L:pair): any *expr*
Returns the last element of the pair L. A type mismatch error results if L is not a pair.

```
(de lastcar (l)
  (if (atom l) l (car (lastpair l))))
```

(nth L:pair N:integer): any *expr*
Returns the Nth element of the list L. If L is atomic or contains fewer than N elements, an out of range error occurs.

```
(de nth (l n)
  (cond ((null l) (range-error))
        ((onep n) (first l))
        (t (nth (rest l) (sub1 n)))))
```

Note that this definition is not compatible with Common LISP. The Common LISP definition reverses the arguments and defines the car of a list to be the "zeroth" element.

(pnth L:list N:integer): any *expr*
Returns a list starting with the nth element of the list L. Note that the result is a pointer to the nth element of L, a destructive function like rplaca can be used to modify the structure of L. If L is atomic or contains fewer than N elements, an out of range error occurs.

```
(de pnth (l n)
  (cond ((onep n) l)
        ((not (pairp l)) (range-error))
        (t (pnth (rest l) (sub1 n)))))
```

5.3.1 Membership and Length of Lists

(member A:any L:list): extra-boolean *expr*

Returns nil if A is not equal to some top level element of the list L; otherwise it returns the remainder of L whose first element is equal to A.

```
(de member (a l)
  (cond ((not (pairp l)) nil)
        ((equal a (car l)) l)
        (t (member a (cdr l)))))
```

(memq A:any B:list): extra-boolean *expr*

The same as member except that eq is used for comparison instead of equal. Note that the value returned by either member or memq is eq to the portion of the list which begins with A. Thus a function like rplaca may be used to alter A.

```
1 lisp> (setq sequence '(1 3 3))
(1 3 3)
2 lisp> (rplaca (memq 3 sequence) 2)
(2 3)
3 lisp> sequence
(1 2 3)
```

(length X:any): integer *expr*

The top level length of the list X is returned.

```
(de length (l)
  (if (atom l) 0 (add1 (length (cdr l)))))
```

Constructing, Appending, and Concatenating Lists

(list [U:any]): list *fexpr*

Construct a list of the arguments.

```
1 lisp> (list (car '(left . right)) (list 'next))
(LEFT (NEXT))
```

(append U:list V:list): list *expr*

Returns a constructed list in which the last element of U is followed by the first element of V. The list U is copied, but V is not.

```
(de append (u v)
  (cond ((not (pairp u)) v)
        (t (cons (first u) (append (rest u) v)))))
```

(nconc U:list V:list): list *expr*

Destructive version of append, the cdr of the last pair of U is modified to reference V. While append creates a copy of U, nconc uses U itself in constructing the result.

```
(de nconc (u v)
  (if (not (pairp u))
    v
    (rplacd (lastpair u) v)))
```

```
1 lisp> (setq a '(swan))
(SWAN)
3 lisp> (nconc a '(giles))
(SWAN GILES)
4 lisp> a
(SWAN GILES)
```

(aconc U:list V:any): list *expr*

Destructively adds element V to the tail of list U.

```
1 lisp> (setq a '(phillips))
(PHILLIPS)
2 lisp> (progn (aconc a 'posner) a)
(PHILLIPS POSNER)
```

(lconc PTR:list LST:list): list *expr*

Effectively nconc, but avoids scanning from the front to the end of the first list by maintaining a pointer. PTR is a pair whose car is a list L and whose cdr is a reference to the last pair of L. The value returned is the updated pair PTR.

```
(progn (rplacd (cdr ptr) lst)
       (rplacd ptr (lastpair lst)))
```

This function is useful for building lists from left to right, PTR should be initialized to (nil . nil) before the first call on lconc.

(tconc PTR:list ELM:any): list *expr*
Effectively aconc, but avoids scanning from the front to the end of the first list by maintaining a pointer. PTR is a pair whose car is a list L and whose cdr is a reference to the last pair of L. The value returned is the updated pair PTR.

```
(progn (setq elm (ncons elm))
       (rplacd (cdr ptr) elm)
       (rplacd ptr elm))
```

This function is useful for building lists from left to right. PTR should be initialized to (nil . nil) before the first call on tconc.

Lists as Sets

A set is a list in which each element occurs only once. Since the order of elements in a set does not matter, these functions may not preserve order.

(adjoin ELEMENT:any SET:list): list *expr*
Add Element to SET if it is not already a member.

```
(de adjoin (elm set)
  (if (member elm set) set (cons elm set)))
```

Recall that member uses equal to test for equality.

(adjoinq ELEMENT:any SET:list): list *expr*
Similar to adjoin except that eq is used to test for set membership.

(union X:list Y:list): list *expr*
Returns the union of sets X and Y.

```
(de union (x y)
  (if (not (pairp x))
    y
    (union (rest x)
```

```
            (if (member (first x) y)
              y
              (cons (first x) y)))))
```

Notice that the two arguments to union are assumed to be sets, if either contains duplicates then the result may contain duplicates as well.

```
1 lisp> (union '(1 2 2) '(3))
(2 1 3)
2 lisp> (union '(3) '(1 2 2))
(3 1 2 2)
```

(unionq X:list Y:list): list *expr*

Similar to union except that eq is used to test for set membership.

(intersection U:list V:list): list *expr*

Returns the intersection of sets U and V.

```
(de intersection (u v)
  (cond ((not (pairp u)) nil)
        ((member (car u) v)
         (cons (car u)
               (intersection (cdr u) (delete (car u) v))))
        (t (intersection (cdr u) v))))
```

Notice that the two arguments to intersection are assumed to be sets, if either contains duplicates then the result may contain duplicates as well.

```
1 lisp> (intersection '(1 2) '(2))
(2)
2 lisp> (intersection '(2 2) '(1 2 2))
(2 2)
```

(intersectionq U:list V:list): list *expr*

Similar to intersection except that eq is used to test for set membership.

(list2set SET:list): list *expr*

Remove redundant elements from the top level of SET using equal.

(list2setq SET:list): list *expr*
Remove redundant elements from the top level of SET using eq.

5.3.2 Deleting Elements of Lists

Note that the functions suffixed by IP will destructively modify the list from which elements are being deleted. If you use such a function do not rely on the result being eq to the argument. The value returned will have all of the elements removed, but the modifications which have been made to the argument may not reflect this. In particular, the leading elements which are equal to the element being deleted will not be spliced out of the list.

```
1 lisp> (setq this '(a b c))
(a b c)
2 lisp> (deletip 'a this)
(b c)
3 lisp> this
(a b c)
2 lisp> (reversip this)
(c b a)
3 lisp> this
(a)
```

(delete U:any V:list): list *expr*
Returns V with the first top level occurrence of U removed from it. Equal is used for comparing elements. The result consists of a copy of the portion of U which comes before the deleted element, and the portion of U which follows the deleted element.

```
(de delete (u v)
  (cond ((not (pairp v)) v)
        ((equal (car v) u) (cdr v))
        (t (cons (car v) (delete u (cdr v))))))
```

(del F:function U:any V:list): list *expr*
Generalized delete function with F as the comparison function.

```
1 lisp> (del '(lambda (i e) (> i e)) 0 '(-2 3 -1))
(-2 -1)
```

(deletip U:any V:list): list *expr*
The destructive version of delete, V may be modified.

(delq U:any V:list): list *expr*
Returns V with the first top level occurrence of U removed from it. Eq is used for comparing elements. The result consists of a copy of the portion of U which comes before the deleted element, and the portion of U which follows the deleted element.

(delqip U:any V:list): list *expr*
The destructive version of delq, V may be modified.

(delasc U:any V:a-list): a-list *expr*
Returns V with the first top level occurrence of (U . ANY) removed from it. Equal is used for comparisons. The result consists of a copy of the portion of U which comes before the deleted element, and the portion of U which follows the deleted element.

(delascip U:any V:a-list): a-list *expr*
The destructive version of delasc, V may be modified.

(delatq U:any V:a-list): a-list *expr*
Returns V with the first top level occurrence of (U . ANY) removed from it. Eq is used for comparisons. The result consists of a copy of the portion of U which comes before the deleted element, and the portion of U which follows the deleted element.

(delatqip U:any V:a-list): a-list *expr*
The destructive version of delatq, V may be modified.

5.3.3 List Reversal

(reverse U:list): list *expr*
Returns a copy of the top level of U in reverse order.

```
(de reverse (l)
  (do ((result nil (cons (car pointer) result))
       (pointer l (cdr pointer)))
      ((not (pairp pointer)) result)))
```

(reversip U:list): list *expr*
The destructive version of reverse. The argument may be destructively modified to produce the result. Note also that the result may not be eq to the argument.

```
(de reversip (l)
  (prog (next result)
        (while (pairp l)
          (setq next (cdr l))
          (setq result (rplacd l result))
          (setq l next))
        (return result)))
```

5.3.4 Functions for Sorting

The **gsort** module provides functions for sorting lists. Some of the functions take a comparison function as an argument. The comparison function takes two arguments and should return nil if the second argument should come before the first in the sorted result. A lambda expression is acceptable as a comparison function. Note that since sorting requires many comparisons, and thus many calls on the comparison function, the sort will be much faster if the comparison function is compiled.

(gsort TABLE:list LEQ-FN:id,function): list *expr*
Returns a sorted list. LEQ-FN is the comparison function used to determine the sorting order. The original TABLE is unchanged. Gsort uses a stable sorting algorithm. In other words, if X appears before Y in TABLE then X will appear before Y in the result unless X and Y are out of order.

(gmergesort TABLE:list LEQ-FN:id,function): list *expr*
The destructive version of gsort, this function is somewhat faster than gsort. Note that you should use the value returned by the function, don't depend on the modified argument to give the right answer.

(idsort TABLE:list): list *expr*
Returns a list of the ids in TABLE, sorted into alphabetical order. The original list is unchanged. Case is not significant in determining the alphabetical order.

```
1 lisp> (setq x '(3 8 -7 2 1 5))
(3 8 -7 2 1 5)
2 lisp>     % sort from smallest to largest
2 lisp> (gsort x 'leq)
(-7 1 2 3 5 8)
3 lisp>     % sort from largest to smallest
3 lisp> (gmergesort x 'geq)
(8 5 3 2 1 -7)
4 lisp>     % note that the value of x has been modified
4 lisp> x
(3 2 1 -7)
5 lisp> (idsort '(the quick brown fox jumped over the lazy dog))
(brown dog fox jumped lazy over quick the the)
```

5.4 Functions for Building and Searching A-Lists

(assoc U:any V:a-list): pair, nil *expr*
If U occurs as the car portion of an element of the a-list V, the pair in which U occurred is returned, otherwise nil is returned. The function equal is used to test for equality. As illustrated below, it is possible to update the table that was the second argument to assoc by using the function rplacd on the result.

```
(de assoc (u v)
  (cond ((not (pairp v)) nil)
        ((and (pairp (car v)) (equal u (caar v))) (car v))
        (t (assoc u (cdr v)))))
```

```
1 lisp> (setq table '((oranges . 4) (apples . 2)))
((oranges . 4) (apples . 2))

2 lisp> (rplacd (assoc 'apples table) 0)
(apples . 0)

3 lisp> table
((oranges . 4) (apples . 0))
```

(atsoc R1:any R2:any): any *expr*

Similar to assoc except that eq is used to make comparisons.

(ass F:function U:any V:a-list): pair, nil *expr*

Ass is a generalized assoc function. F is the comparison function.

(sassoc U:any V:a-list FN:function): any *expr*

Searches the a-list V for an occurrence of U. If U is not in the a-list, the evaluation of function FN is returned. Note that FN should be a function with no formal parameters.

```
(de sassoc (u v fn)
  (cond ((not (pairp v)) (apply fn nil))
        ((and (pairp (car v)) (equal u (caar v))) (car v))
        (t (sassoc u (cdr v) fn))))
```

(pair U:list V:list): a-list *expr*

U and V are lists which must have an identical number of elements. If not, an error occurs. Returned is a list in which each element is a pair, the car of the pair being from U and the cdr being the corresponding element from V.

```
(de pair (u v)
  (cond ((and (pairp u) (pairp v))
         (cons (cons (car u) (car v)) (pair (cdr u) (cdr v))))
        ((or (pairp u) (pairp v))
         (length-error))
        (t nil)))
```

5.5 Substitutions

(subst U:any V:any W:any): any *expr*
Returns the result of substituting U for all occurrences of V in W. Copies all of W which is not replaced by U. The test used is equal.

```
(de subst (u v w)
  (cond ((null w) nil)
        ((equal v w) u)
        ((not (pairp w)) w)
        (t (cons (subst u v (car w)) (subst u v (cdr w))))))
```

(substip U:any V:any W:any): any *expr*
Destructive subst.

(sublis X:a-list Y:any): any *expr*
This performs a series of substs in parallel. The value returned is the result of substituting the cdr of each element of the a-list X for every occurrence of the car part of that element in Y. Sublis is not quite the correct function for arbitrary code substitutions. As illustrated below, substitutions may enter places you might wish they did not.

```
(de sublis (x y)
  (if (not (pairp x))
    y
    (let ((u (assoc y x)))
      (cond ((pairp u) (cdr u))
            ((not (pairp y)) y)
            (t (cons (sublis x (car y)) (sublis x (cdr y))))))))

1 lisp> (sublis '((x . 100)) '(list 'x 'is x))
(list (quote 100) (quote is) 100)
```

(subla U:a-list V:any): any *expr*
Eq version of sublis; replaces atoms only.

Characters and Strings

6.1 Characters

In PSL a character is its ASCII code representation. Using numeric codes to represent characters leads to programs which are difficult to read. You are encouraged to use char to represent characters.

(char U:id): integer *macro*

This macro is part of the USEFUL module. The char macro returns the ASCII code which corresponds to the single character passed as an argument. Char will accept alias's for characters. An alias is established by defining a charconst property on the property list of the alias. The value of this property should be the ASCII code of the character which is being aliased. The following alias's are defined when the **useful** package is loaded.

NULL	
BELL	
BACKSPACE	
TAB	
LF	line feed
EOL	end of line
FF	form feed
CR	carriage return
EOF	end of file
ESCAPE	may be abbreviated ESC
SPACE	an alias is BLANK
RUBOUT	may be abbreviated RUB
DELETE	may be abbreviated DEL

By default, the PSL reader converts upper case alphabetic characters to lower case. This default is controlled by the value of the switch raise. A

value of t indicates that the conversion should be done. Assuming raise is t, the expression (char a) refers to the character "a". The character "A" is referenced by (char !A). The "!" is used as an escape character, see chapter 12 for more information. The application of lower is said to modify the character. Lower is not a defined function but it does have the attribute char-prefix-function on its property list. The value of this property is a function which will modify the ASCII code of its argument. Modifiers are control and meta (the modifier cntrl is an abbreviation for control). The following example is a simplified definition of char.

```
(defmacro char (u)
  (cond ((idp u) (or (get u 'charconst)
                     (id2int u)))
        ((digit-char u))
        ((and (pairp u)
              (get (first u) 'char-prefix-function))
         (list (get (first u) 'char-prefix-function)
               (list 'char (second u))))))
```

Notice that the digits have to be treated as a special case. The PSL reader converts a digit to a reference to a numeric constant. The definition of the alias space is

```
(put 'space 'charconst 32)
```

If the type of the argument to char is not correct then the warning

```
*** Unknown character constant `FORM'
```

is printed and the result will be zero.

The USEFUL package also defines the read macro #. Read macros are explained in detail in chapter 12. When the reader encounters #, char is applied to the next character. If this next character is a lower case alphabetic character then it will be converted to upper case if the switch raise is set to t. If this next character is a read macro then it will be applied. Such a read macro should return an expression which is acceptable to char.

```
1 lisp> #\a
65
2 lisp> #\`(a b c)
*** Unknown character constant: `(BACKQUOTE (A B C))'
0
```

Common LISP Functions on Characters

The following functions are available by loading the library module **chars**. Common LISP provides a character data type in which every character object has three attributes: code, bits, and font. The bits attribute allows extra flags to be associated with a character. The font attribute permits a specification of the style of the glyphs (such as italics). PSL does not support nonzero bit and font attributes. Because of this, some of the Common LISP character functions described below have no affect or are not very useful as implemented in PSL. They are present for compatibility.
An argument to any of the following functions should make use of char or one of the two read macros #/ and #\. Since a character in PSL is represented as its ASCII code it is possible to give a numeric argument to these functions. However, the use of numbers makes the code difficult to read and much less transportable. The read macro #\ is described in the discussion of char above. The read macro #/ returns the ASCII code of the next character. In contrast to #\, the switch raise is ignored and if the next character is a read macro it is not applied.

```
1 lisp> (eq (char (lower a)) #/a)
T
2 lisp> (eq (char (lower a)) #\a)
NIL
3 lisp> #/‘
96
4 lisp> #\‘
*** Unknown character constant: ‘(BACKQUOTE !^Z)’
0
```

(standard-charp C:character): boolean *expr*

Returns t if the argument is one of the 95 ASCII printing characters.

```
1 lisp> (standard-charp (char a))
T
2 lisp> (standard-charp (char (control a)))
NIL
```

(graphicp C:character): boolean *expr*

Returns t if C is a printable character and nil otherwise. Control and formatting characters are considered to be not printable. The space character is a printable character.

(string-charp C:character): boolean *expr*
Returns t if C is a character that can be an element of a string. Any character that satisfies standard-charp and graphicp also satisfies string-charp.

(alphap C:character): boolean *expr*
Returns t if C is an alphabetic character. Liter is an equivalent function.

(uppercasep C:character): boolean *expr*
Returns t if C is an upper case letter.

(lowercasep C:character): boolean *expr*
Returns t if C is a lower case letter.

(bothcasep C:character): boolean *expr*
In PSL this function is the same as alphap.

(digitp C:character): boolean *expr*
Returns t if C is a digit character (optional radix not supported). An equivalent function is digit

(alphanumericp C:character): boolean *expr*
Returns t if C is a digit or an alphabetic.

(char= C1:character C2:character): boolean *expr*
Returns t if C1 and C2 are the same in all three attributes.

(char-equal C1:character C2:character): boolean *expr*
Returns t if C1 and C2 are similar. Differences in case, bits, or font are ignored by this function.

(char< C1:character C2:character): boolean *expr*
Returns t if C1 is strictly less than C2.

(char> C1:character C2:character): boolean *expr*
Returns t if C1 is strictly greater than C2.

(char-lessp C1:character C2:character): boolean *expr*
Like char< but ignores differences in case, fonts, and bits.

(char-greaterp C1:character C2:character): boolean *expr*
Like char> but ignores differences in case, fonts, and bits.

(char-code C:character): character *expr*
Returns the code attribute of C. In PSL this function is an identity function.

(char-bits C:character): integer *expr*
Returns the bits attribute of C, which is always 0 in PSL.

(char-font C:character): integer *expr*
Returns the font attribute of C, which is always 0 in PSL.

(code-char I:integer): character,nil *expr*
The purpose of this function is to be able to construct a character by specifying the code, bits, and font. Because bits and font attributes are not used in PSL, code-char is an identity function.

(character C:character, string, id): character *expr*
Attempts to coerce C to be a character. If C is a character, C is returned. If C is a string, then the first character of the string is returned. If C is a symbol, the first character of the print name of the symbol is returned. Otherwise an error occurs.

```
***** 'FORM' cannot be coerced to a character
```

(char-upcase C:character): character *expr*
If (lowercasep C) is true, then char-upcase returns the code of the upper case of C. Otherwise it returns the code of C.

(char-downcase C:character): character *expr*
If (uppercasep C) is true, then char-downcase returns the code of the lower case of C. Otherwise it returns the code of C.

(digit-char N:fixnum): integer *expr*
If N corresponds to a single digit then the character which represents that digit is returned. The Common LISP version will accept an optional radix argument, this function assumes a radix of ten. If N does not correspond to a single digit then nil is returned.

(char-int C:character): integer *expr*
Converts character to integer. This is the identity operation in PSL.

(int-char I:integer): character *expr*
Converts integer to character. This is the identity operation in PSL.

6.2 Strings

6.2.1 String Creation and Copying

The following are built-in string creation and copying functions:

(allocate-string SIZE:integer): string *expr*
Constructs and returns a string with SIZE characters. The contents of the string are not initialized.

(make-string SIZE:integer INITVAL:integer): string *expr*
Constructs and returns a string with SIZE characters, each initialized to the ASCII code INITVAL.

(mkstring UPLIM:integer INITVAL:integer): string *expr*
An old form of make-string. Returns a string of characters all initialized to INITVAL, with upper bound UPLIM. The returned string contains a total of (add1 UPLIM) characters.

(string [ARGS:integer]): string *nexpr*
Create string of elements from a list of ARGS.

```
1 lisp> (string 65 66 67)
"ABC"
```

(copystringtofrom NEW:string OLD:string): NEW:string *expr*
Copy all characters from old into new. This function is destructive.

(copystring S:string): string *expr*
Copy to new string, allocating space.

6.2.2 About the Basic String Operations

The representation of strings is very similar to that of vectors. Due to this similarity, there are functions which may be applied to either data type. Such functions provide primitive operations on strings. PSL provides many other functions specific to strings which are defined in various library modules.

6.2.3 The Operations

This section documents functions in the library module **slow-strings (s-strings)**. There is another library module called **fast-strings (f-strings)**. The fast-strings module provides alternate definitions for these functions. When the switch fast-strings is non-nil the compiler will use these alternate definitions to produce efficient code. However, there will not be any verification that arguments are of correct type (in addition, it is assumed that numeric arguments are within a proper range). If invalid arguments are used, then at best your code will not generate correct results, you may actually damage the PSL system. There are two side effects to loading **fast-strings** The **slow-strings** module will be loaded and the switch fast-strings will be set to t. For more information on the switch fast-strings see Chapter 19.

(string-fetch S:string I:integer): any *expr*
Accesses an element of a PSL string. Indexes are zero based. The ASCII character stored in that position of the string is returned.

Characters are represented by inums. You should not rely on this when you write code. Such code cannot be easily transported to other systems where either the encoding is different or where characters are a seperate data type.

(string-store S:string I:integer X:char): None Returned *expr*
Stores into a PSL string. String indexes start with 0.

(string-length S:string): integer *expr*
Returns the number of elements in a PSL string. Since indexes start with index 0, the size is one larger than the greatest legal index. Compare this function with string-upper-bound, documented below.

(string-upper-bound S:string): integer *expr*
Returns the greatest legal index for accessing or storing into a PSL string. Compare this function with string-length, documented above.

(string-empty? S:string): boolean *expr*
True if the string has no elements (its size is 0), otherwise nil.

6.3 Common LISP String Functions

A Common LISP compatible package of string functions has been implemented in PSL, obtained by loading the **strings** module. This section describes the **strings** module, including a few functions in it that are not Common LISP functions.

6.3.1 String comparison:

(string= S1:string S2:string): boolean *expr*
Compares two strings S1 and S2, case sensitive.

(string-equal S1:string S2:string): boolean *expr*
Compare two strings S1 and S2, ignoring case, bits and font.

The following string comparison functions are extra-boolean. If the comparison results in a value of t, the index of the first character position at which the strings fail to match is returned. The result can also be thought of as the longest common prefix of the two strings.

(string< S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case sensitive.

(string> S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case sensitive.

(string<= S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case sensitive.

(string>= S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case sensitive.

(string<> S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case sensitive. In Common LISP the function is named string/=.

(string-lessp S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case differences are ignored.

(string-greaterp S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case differences are ignored.

(string-not-greaterp S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case differences are ignored.

(string-not-lessp S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case differences are ignored.

(string-not-equal S1:string S2:string): extra-boolean *expr*
Lexicographic comparison of strings. Case differences are ignored.

6.3.2 String Concatenation:

(string-concat [S:string]): string *macro*
Concatenates all of its string arguments, returning the newly created string. Not in Common LISP.

(string-repeat S:string I:integer): string *expr*
Appends copy of S to itself total of I-1 times. Not in Common LISP.

6.3.3 Transformation of Strings:

(substring S:string LO:integer HI:integer): string *expr*
Same as subseq, but the first argument must be a string. Returns a substring of S of size (sub1 (- HI LO)), beginning with the element with index LO. Not in Common LISP.

(string-trim BAG:{list, string} S:string): string *expr*
Remove leading and trailing characters in BAG from a string S.

```
1 lisp> (string-trim "ABC" "AABAXYZCB")
"XYZ"
2 lisp> (string-trim (list (char a)
2 lisp>                    (char b)
2 lisp>                    (char c))
2 lisp>              "AABAXYZCB")
"XYZ"
3 lisp> (string-trim '(65 66 67) "ABCBAVXZCC")
"VXZ"
```

(string-left-trim BAG:{list, string} S:string): string *expr*
Remove leading characters from string.

(string-right-trim BAG:{list, string} S:string): string *expr*
Remove trailing characters from string.

(string-upcase S:string): string *expr*
Copy and raise all alphabetic characters in string.

(nstring-upcase S:string): string *expr*
Destructively raise all alphabetic characters in string.

(string-downcase S:string): string *expr*
Copy and lower all alphabetic characters in string.

(nstring-downcase S:string): string *expr*
Destructively lower all alphabetic characters in string.

(string-capitalize S:string): string *expr*
Copy and raise first letter of all words in string; other letters in lower case.

(nstring-capitalize S:string): string *expr*
Destructively raise first letter of all words; other letters in lower case.

6.3.4 Type Conversion:

(string-to-list S:string): list *expr*
Unpack string characters into a list. Not in Common LISP.

(string-to-vector S:string): vector *expr*
Unpack string characters into a vector. Not in Common LISP.

6.3.5 Other:

(string-length S:string): integer *expr*
Last index of a string, plus one. Not in Common LISP. Use string-size.

(rplachar S:string I:integer C:character): character *expr*
Store a character C in a string S at position I.

6.3.6 Substring Comparison

The library module STRING-SEARCH provides efficient functions for comparing a string against a substring of another string.

(substring= S1:string S2:string START:integer): boolean *expr*
Returns true if there is a substring of S2 starting at START and string= to S1, otherwise returns nil.

Similar to

```
(string= S1 (substring S2 START (+ START (string-length S1))))
```

but note that this returns nil (no error is signalled) if there are fewer than

```
(string-length S1)
```

characters from position START through the end of S2.

(substring-equal S1:string S2:string START:integer): boolean *expr*
Returns true if there is a substring of S2 starting at START and string-equal to S1, otherwise returns nil.

Similar to

```
(string-equal S1 (substring S2 START (+ START (string-length S1))))
```

but note that this returns nil (no error is signalled) if there are fewer than

```
(string-length S1)
```

characters from position START through the end of S2.

6.3.7 Searching for Strings

The library module **str-search** provides functions for searching for a string within another string. These functions are efficiently implemented.

The two strings involved in these searching operations are referred to as the "domain" and the "target". These functions search for an occurrence of the "target" string within the "domain" string.

The operations for string searching return the index of the leftmost character in the first matching part of the domain string that is found, starting from the left. If no match is found, nil is returned.

(string-search TARG:string DOM:string):{integer, nil} *expr*
Searches for the leftmost occurrence of the target in the domain. This function is case-sensitive. If passed two strings, Common LISP "search" will give the same results.

(string-search-from TARG:string DOM:string START:integer):
{integer, nil} *expr*
Like string-search, but the search effectively starts at index START in the domain.

(string-search-equal TARG:string DOM:string):{integer, nil} *expr*
Like string-search except that the comparisons are case-insensitive.

(string-search-from-equal TA:string D:string ST:integer):
{integer, nil} *expr*
Like string-search-from except that the comparisons are case-insensitive.

6.3.8 Reading and Writing Strings

The library module **str-input** provides some facilities to support taking input from strings. Among other things, this permits a user to obtain a number from its printed representation using the PSL number parser.

(with-input-from-string HEADER:list [BODY:form]):any *macro*
The argument HEADER should be a two element list.
The first element should be an identifier, the second a string. (< *channel* >< *string* >). The argument < *string* > is treated as if it were the text of a file. The argument < *channel* > is bound to an input channel which is opened to give access to < *string* >. Once the channel has been opened, each form BODY is evaluated (the forms are evaluated in a left to right order). It is expected that these forms will be used to read and process input from < *string* >. The value of the last form is returned. Once the application of with-input-from-string is complete the input channel will be closed.

```
(de string-to-words (string)
  (with-input-from-string
    (channel string)
    (do ((result nil (aconc result item))
         (item (channelread channel) (channelread channel)))
        ((eq item $eof$) result))))

1 lisp> (string-to-words "DOCUMENTATION IS FUN")
(DOCUMENTATION IS FUN)
```

(string-read STR:string): any *expr*
Reads one s-expression from the string STR. The function channelread is used to do this. Note that it is not necessary to terminate the string with a delimiter character. An end of file character is also considered to be a delimiter character.

```
1 lisp> (string-read "TOKEN")
TOKEN
2 lisp> (string-read "TWO TOKENS")
TWO
```

Notice that characters beyond the first s-expression are ignored. This function is defined in terms of with-input-from-string.

```
(de string-read (string)
  (with-input-from-string
    (ch string)
    (channelread ch)))
```

The library module **string-output** provides a facility for writing to strings. The function bldmsg provides the capability to construct a string using formatting directives. However, complicated strings can be constructed more easily using the macro with-output-to-string. For example, longer strings can be constructed by including the channellinelength function; items can be printed to the string incrementally (in a stream-like fashion) from a loop

(with-output-to-string HEADER:list [BODY:form]): string *macro*
The argument HEADER should be a two element list. The first element should be an identifier. The second element can be either a string or nil (in which case a default initial string is allocated) – in either case, the initial string is extened as necessary.
The written string is return as a result (this is a substring copy of the string used for writing). For example,

```
(setf row
  (with-output-to-string (wchan nil)
    (channellinelength wchan 350)
    (for (in tab '(0 100 200 300))
     (in str '("A" "B" "C" "D"))
     (do (channelprintf wchan "%t%w" tab str))
     )))
```

Flow of Control

7.1 Introduction

In a PSL program the flow of control is described primarily by function application. A function may call any number of other functions, including itself. This allows complex operations to be described by a number of small functions, each of which implements a simple operation. In addition, PSL provides a number of other control constructs.

7.1.1 Conditionals

Conds and Ifs

(cond [U:form-list]): any *open-compiled fexpr*

A typical application of cond is shown below, the square brackets are used to indicate zero of more occurances of an expression.

```
(COND ([(<predicate> [<action>])]))
```

The first form in each clause is treated as a predicate; the remaining (zero or more) forms are treated as if they were enclosed within a progn. The evaluation proceeds by sequentially evaluating the predicates in the order of their appearance until one evaluates to a non-nil value. Then the remaining forms in this clause are evaluated and the value of the last form is returned as the result. If only the predicate appears, then its value (if non-nil) becomes the value returned. If no predicate is non-nil then the result is nil. The following definition demonstrates the use of cond.

```
(de size (data)
```

```
(cond ((pairp data) (length data))
      ((stringp data) (string-length data))
      ((vectorp data) (vector-size data))
      (t 'unknown)))
```

This function will compute the length of lists strings and vectors.
The following macros are defined in the USEFUL module.

(if E:form S0:form [S:form]): any *macro*

The form S0 is evaluated if the test E is non-nil, otherwise the remaining forms S are evaluated, the value of the last is returned. If is a macro to simplify the writing of a common form of cond in which there are only two clauses and the antecedent of the second is t.

The expression

```
(if (minusp number) 'negative 'positive)
```

is preferred over

```
(cond ((minusp number) 'negative)
      (t 'positive))
```

Notice that a single form is evaluated when the test expression E is non-nil but there may be any number of expressions evaluated when the value of E is nil.
Related macros for common cond forms are when and unless.

(when E:form [S:form]): any *macro*

When the value of the test expression E is non-nil the forms S are evaluated. The value of the last form is returned as the result. The result is nil if the test expresson E is nil.

(unless E:form [U:form]): any *macro*

If the value of E is nil then the forms S are evaluated. The result is nil if the test expression E is non-nil.
The boolean functions and and or (see Chapter 2), may be used as conditional forms. For example an expression like

```
(and (pairp x) (eq (car x) 'a))
```

which relies upon the order of the evaluation of the arguments is often used. If and had evaluated the second argument first an error may have been generated. However, the arguments are always evaluated from left to right. Or may be used to retrieve a value as in the function definition below.

```
(de dispatch (data arguments)
(apply (or (get data 'function) default-function)
arguments))
```

In this example the function which is applied is either found on the property list of the data or it is a default. In reading such an expression one considers an argument to be preferred over anything which follows it. Note that the use of these functions as conditionals may yield code which is confusing. For example the code

```
(setq x (and y 3))
```

will set x to 3 if y is bound to a value other than nil, otherwise x will be set to nil. It is recommended that the following be used instead.

```
(setq x (if y 3 nil))
```

The following version of if is defined in the module **if**. It is upward compatible with the if macro defined above. This version will accept the keywords then, else, and elseif. If a keyword appears immediately after the conditional expression of the if then the expression is taken to be in keyword form.

(if <expr> [then <expr> ...][<elseif-part> ...][else <expr> ...]): ***expr***
any

This is not the same notation used generally in the PSL manual. Square brackets enclose parts optionally present. The ellipses indicate arbitrary additional repetitions of the thing appearing just before them. The elseif-part may be one of two forms.
<elseif-part> = else if <expr> [then <expr> ...]
<elseif-part> = else if <expr> [then <expr> ...]

7.2 Case and Selectq Statements

Case is a form of conditional in which a "key" value is compared against a set of values in order to choose a corresponding set of forms to evaluate. The "key" value must be an integer. This is a restricted cond, and therefore can

be compiled into more efficient code. The compiler expends some effort to examine special cases (for example compact vs. non-compact sets of cases and short vs. long sets of cases).

(case I:form [U:case-clause]): any *open-compiled fexpr*

Case selects a case-clause (one of the U's), to evaluate based on the value of I. The expression I should evaluate to an integer. Each clause has the form (case-expression form), where the case-expression has one of the following forms.

nil	the default case
(i1 i2 ... in)	where each ik is an integer
((range low high))	where low and high are integers and low is less than or equal to high

The following example illustrates the use of case

```
(case i ((1) (print "first"))
        ((2 3) (print "second"))
        (((range 4 10)) (print "third"))
        (nil (print "fourth")))
```

(selectq I:form [U:selectq-clause]): any *macro*

This function selects an action based on the value of the "key" I. Each selectq-clause is of the form (key-part [action]). The key-part is a list of keys, t, or otherwise. If there is only one key in a key-part it may be written in place of a list containing it, provided that the key is not a list, nil, t, or otherwise.
After I is evaluated, it is compared against the members of each key-part in turn. If the key is eq to any member of the key-part then each of the forms in that selectq-clause are evaluated. The value of the last form of the list is the value of the selectq. If a selectq-clause with key-part t or otherwise is reached, its forms are evaluated without further testing. A t or otherwise clause should be the last clause. If no clause is satisfied then nil is returned.

```
(selectq (car w)
  ((nil) nil)
  (end (print 'done) 'end)
  ((0 1 2 3 4 5 6 7 8 9) 'digit)
  (otherwise 'other))
```

7.3 Sequencing Evaluation

These functions provide for explicit control sequencing, and the definition of blocks altering the scope of local variables.

(let A:list [B:form]): any *macro*
The general form follows, the square brackets are used to indicate zero or more occurances of an expression.

```
(LET ([(<var> <value>)]) [<body>])
```

The <value>s are evaluated (in an unspecified order), and then the <var>s are bound to these values. The body, consisting of the <body> forms, is evaluated in a left to right order. The value returned is the result of the last body form or nil if the body is empty. Note that the <value>s are evaluated in the outer environment, before the <var>s are bound.

This function is equivalent to

```
((lambda ([<var>]) [<body>]) [<value>])
```

The let-style is attractive since it places the <var>s close to their binding forms (<value>s), thereby increasing readability.

There are two shorthand formats for (<var> <value>). One is (< *var* >) and the other is just < *var* >. Both of these mean bind < *var* > to nil. As a rule of style, you should use (< *var* > nil) if you mean to use the value of < *var* > without assigning it it first, and just (< *var* >) or < *var* > if you do not care what < *var* > gets bound to.

The following expression returns the middle element of a vector.

```
(let ((n (vector-size vector)))
  (unless (zerop n)
    (vector-fetch vector (add1 (/ n 2)))))
```

(let* A:list [B:form]): any *macro*
Let* is just like let except that it makes the assignments sequentially. That is, the first binding is made before the value for the second one is computed. The example below illustrates the difference between let and let*.

```
1 lisp> (setq front 'red back 'orange)
orange
2 lisp> (let ((front 'blue) (back front))  back)
red
3 lisp> (let* ((front 'blue) (back front))  back)
blue
```

(progn [U:form]): any *open-compiled fexpr*

U is a set of expressions which are executed sequentially. The value returned is the value of the last expression.

(prog2 A:form B:form): any *open-compiled expr*

Returns the value of the second argument B. Note that prog2 expects only two arguments.

(prog1 [U:form]): any *macro*

Prog1 is a function defined in the USEFUL package. Prog1 evaluates its arguments in order, like progn, but returns the value of the first.

(prog VARS:id-list [PROGRAM:id,form]): any open-compiled fexpr

VARS is a list of ids which are considered fluid if the prog is interpreted and local if compiled (see the "Variables and Bindings" Section, 9.2). The prog's variables are allocated space when the prog form is applied, and are deallocated when the prog is exited. Prog variables are initialized to nil. The program is a set of expressions to be evaluated in order of their appearance in the prog function. An id which appears at the top level of the program are labels which can be referred by go. The value returned by the prog function is determined by a return function or nil if the prog "falls through".

```
(de sum-up (seq)
   (prog (sum)
     (setq sum 0)
    loop
     (when (null seq) (return sum))
     (when (numberp (first seq))
       (setq sum (plus sum (first seq)))
       (setq seq (rest seq))
```

```
        (go loop))))

1 lisp> (sum-up '(1 3 5))
9
2 lisp> (sum-up '(a))
nil
```

There are restrictions as to where the control functions go and return may be placed. The functions go and return are intended to be used within a prog. This is so that they may have only locally determinable effects. Unfortunately these restrictions are not consistent across compiled and interpreted code. It is recommended that if a non-local exit is required then use catch and throw (see section 8.5).
In interpreted code, upon encountering a return a search is made for the latest instance of an entrance to a prog. If the search is successful then the evaluation of that prog is considered complete, the value returned is the argument to return. If the search for a prog fails then the message

```
***** RETURN attempted outside the scope of a PROG
```

is displayed. The treatment of return is much different when the code is compiled. A return may appear outside the scope of a prog. For example, the compiler will compile the sequence

```
(if (not (numberp x)) (return 'unknown))
(compute x))
```

as if it were were enclosed within a prog.

```
(prog ()
  (if (not (numberp x)) (return 'unknown))
  (return (compute x))))
```

Upon reaching a go within interpreted code a search is made for the latest entrance to a prog. If this search is successful then a second search is made for a label which matches the argument to go. The failure of either search is an error. If the first search fails then

```
***** GO attempted outside the scope of a PROG
```

is printed. The message

```
***** `LABEL' is not a label within the current scope
```

is printed if the second search fails. When a prog form is compiled the compiler expects to be able to resolve all label references. Thus every go must appear within a prog otherwise the following error message is printed.

```
***** FORM invalid go
```

In addition, the argument to a go must refer to a label defined inside the prog which contains that go. The message

```
***** Compiler bug: missing label LABEL
```

is printed to indicate when this second restriction is not met.

(go LABEL:id): None Returned *open-compiled fexpr*
Go alters the normal flow of control within a prog function. The next statement of a prog function to be evaluated is immediately preceded by label.

(return U:form): None Returned *open-compiled expr*
Within a prog, return terminates the evaluation of a prog and returns U as the value of the prog.

7.3.1 Iteration

(while E:form [S:form]): nil *macro*
This is a commonly used construct for indefinite iteration in PSL. E is evaluated; if non-nil the S's are evaluated from left to right and then the process is repeated. If E evaluates to nil the while returns nil. Exit may be used to terminate the while and to return a value. Next may be used to terminate the current iteration.

(repeat [S:form] E:form): nil *macro*
The S's are evaluated left to right, and then E is evaluated. This is repeated until the value of E is non-nil, at which point repeat returns nil. Next and exit may be used in the S's to branch to the next iteration of a repeat or to terminate one and possibly return a value.

(next): None Returned *open-compiled, restricted macro*
This terminates the current iteration of the most closely surrounding while or repeat, and causes the next to commence. Both while and repeat are macros which expand into prog's and next is essentially a go. The restrictions on the placement of next are similar to those of go, see section 8.3 for details.

(exit [U:form]): None Returned *open-compiled,restricted macro*
The U's are evaluated left to right, the most closely surrounding while or repeat is terminated, and the value of the last U is returned. With no argument nil is returned. Both while and repeat are macros which expand into prog's and exit is essentially a return. The restrictions on the placement of exit are similar to those of return, see section 8.3 for details.
The following function defintion is intended to illustrate the use of repeat and while. The function will return a list of prime numbers which are less than or equal to the argument N, which is assumed to be greater than one.

```
(de primes (n)
  (let ((result (list 2))
        (number 3)
        (pointer)
       (prime))
    (while (<= number n)
      (setq pointer result)
      (setq prime t)
      (repeat
        (when (zerop (remainder number (first pointer)))
          (setq prime nil))
        (setq pointer (rest pointer))
        (or (null pointer) (not prime)))
      (when prime (setq result (aconc result number)))
      (incr number))
    (cons 1 result)))
```

For

A simple for construct is available in the basic PSL system; an extended version is defined in the USEFUL package. The basic PSL for provides only the iterator FROM and the action clause DO. PSL users should use the extended version, described here:

(for [S:form]): any *macro*

Each argument to for is a one of the clauses described below. If an argument is not a clause then one of the following continuable errors will occur.

```
***** For clauses may not be atomic: 'SYMBOL'

***** Unknown for clause operator: 'LIST'
```

A clause is a list, its first element is an identifier, the remaining elements are arguments. A clause may introduce a local variable, specify a return value, or specify when the iteration should cease.
The first few clauses are used to introduce local variables. Some of these clauses also provide the means to stop loop iteration.

```
(in <variable> <list> <function>)
```

The variable <variable> is set to successive elements of <list>. The argument <function> is optional. If present, it may be either the name of a function or a lambda expression. The function is applied to the extracted element before it is assigned to <variable>. Once the argument <list> is exhausted the iteration will stop. The only argument which will be evaluated is <list>.

```
1 lisp> (for (in v '(0 1) add1)
             (do (print v)))
1
2
nil
```

```
(on <variable> <list>)
```

The variable <variable> is set to successive cdrs of <list>. The first value assigned to <variable> is <list>. Once the <list> is exhausted the iteration will stop. The only argument which will be evaluated is <list>.

```
1 lisp> (for (on v '(0 1))
             (do (print v)))
(0 1)
(1)
nil
```

```
(from <variable> <initial> <final> <step>)
```

The variable <variable> is set to <initial> for the first iteration of the loop. The value of <variable> will be incremented by <step> before each following iteration. Once the value of <variable> is larger than <final> the iteration will stop. Both <initial> and <step> are optional, the default values for each is 1. The argument <final> is optional, in which case the iteration must be stopped by another clause. Each argument except for <variable> will be evaluated once, before the first iteration. To specify <step> without <initial> and <final>, or <final> with <initial> omitted place nil in the slot to be omitted.

```
1 lisp> (for (from v 1 5 2)
             (do (print v)))
1
3
5
```

`(FOR <variable> <initial> <next>)`

At the outset of the first iteration, the variable <variable> will be set to the evaluation of <initial>. Prior to subsequent iterations, the expression <next> will be evaluated and assigned to <variable>.

```
1 lisp> (for (for v 1 (add1 v))
             (until (> v 3))
             (do (print v)))


1
2
3
nil
```

`(with [<variable-form>])`

Each argument <variable-form> is either <variable> or (<variable> <initial>). The square brackets are used to indicate zero or more occurances of <variable-form>. If the first form is used then the variable will be set to nil prior to the first iteration. With the second form the variable will be set to the value of <initial>.

`(do [<form>])`

The square brackets are used to indicate zero or more occurances of <form>. Each expression <form> is evaluated during each iteration. They are evaluated in the order of their appearance. You may use return within a <form>, it will cause an immediate exit from the for.
There are two clauses which allow for the evaluation of expressions before the first iteration, and after the last iteration.

```
(initially [<form>])
```

The square brackets are used to indicate zero or more occurances of <form>. Once the iteration variables have been bound to their values each expression <form> will be evaluated. They are evaluated in the order of their appearance.

```
(finally [<form>])
```

The square brackets are used to indicate zero or more occurances of <form>. After the final iteration each expression <form> will be evaluated. They are evaluated in the order of their appearance. The use of the clauses always and never (described below), may prevent evaluation of the arguments to this clause. There are clauses which specify a return value, if none of them are used then the value of the last <form> will be the value of the for.

```
1 lisp> (for (from v 1 3)
             (finally v))
4
nil
2 lisp> (for (for v 1 (add1 v))
             (always (< v 3))
             (finally v))
nil
```

The next few clauses are used to build a value to be returned from for Except for the returns and returning clauses, a second argument is used in these clauses to specify that instead of returning the result it will be stored as the value of this second argument. This means that the second argument should be an identifier, it will not be evaluated.

```
1 lisp> (for (in v '(0 1))
             (collect (add1 v)))
(1 2)
2 lisp> (for (in v '(0 1))
             (collect (add1 v) result))
nil
3 lisp> result
(1 2)
```

If more than one return value is implied then an error will result

```
1 lisp> (for (in v '(0 1))
             (collect v)
             (adjoin v))
***** For loops may only return one value
```

```
(returns [<form>])
```

Prior to returning from the for each <form> is evaluated. The order of evalution is left to right. The value of the last <form> is returned as the value of the for.A synonym for returns is returning.

```
(collect <form> <variable>)
```

This clause is used to build a list. During each iteration the value of <form> is added at the end of the list. The use of the optional argument <variable> is described above.

```
1 lisp> (for (on v '(one two))
             (collect v))
((one two) (two))
```

```
(ADJOIN <form> <variable>)
(ADJOINQ <form> <variable>)
```

These clauses are similar to collect. The difference is that the value of <form> is only added to the list if it is not already an element. To detemine membership in the list adjoin uses member, adjoinq uses memq. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in i '("one" "one"))
             (adjoin i one)
             (adjoinq i two))
nil
2 lisp> one
("one")
3 lisp> two
("one" "one")
```

```
(JOIN <form> <variable>)
(CONC <form> <variable>)
```

These clauses are similar to collect. The difference is that the value of <form> is appended (nonc is used with the conc clause), to the end of the list. The use of the optional argument <variable> is described above.

```
1 lisp> (for (on v '(one two))
             (join v))
(one two two)
```

You should be careful with conc. In the example, if conc were used in place of join the computation would never terminate.

```
(UNION <form> <variable>)
(UNIONQ <form> <variable>)
```

These clauses are used to build a set. During each iteration the union of <form> and the set being constructed is computed. Set membership is determined with equal for union, and eq for unionq. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '((0 2) (0 1 2 3) (0)))
             (unionq v))
(3 2 1 0)
```

```
(INTERSECTION <form> <variable>)
(INTERSECTIONQ <form> <variable>)
```

These clauses are similar to union and unionq. The difference is that the intersection of sets in constructed instead of the union. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '((0 2) (0 1 2 3) (0)))
             (intersectionq v))
(0)
```

```
(COUNT <form> <variable>)
```

Returns the number of times <form> evaluates to a nonnil value. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '(0 1 2))
             (count (zerop (remainder v 2))))
2
```

```
(SUM <form> <variable>)
```

Returns the sum of each evaluation of <form>. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '(2 3 4))
             (sum v))
9
```

```
(PRODUCT <form> <variable>)
```

Returns the product of each evaluation of <form>. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '(2 3 4))
             (product v))
24
```

(MAXIMIZE <form> <variable>)

Returns the maximum value of <form>. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '(1 2 3))
             (maximize v))
3
```

(MINIMIZE <form> <variable>)

Returns the minimum value of <form>. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '(1 2 3))
             (minimize v))
1
```

(MAXIMAL <value> <test> <variable>)
(MINIMAL <value> <test> <variable>)

These clauses are generalizations of the clauses maximize and minimize. Maximal determines the greatest value of <test> over all of the loop iterations. The corresponding value of <value> is returned. As a particular case it is possible to return the value of an iteration variable for which some function attains a maximum value. The functions used for comparisons are greaterp and lessp. The use of the optional argument <variable> is described above.

```
1 lisp> (for (in v '(2 -2))
             (minimal v (- (expt v 2) (* 7 v))))
2
```

The remaining clauses are used to control loop iteration.

(ALWAYS [<form>])

The square brackets are used to indicate zero or more occurances of <form>. If there is more than one form then the clause is equivalent to (ALWAYS (and [<form>])). This clause is used to specify a return value, t is returned if each <form> is non-nil during each iteration. If one of the <form>s evaluates to nil then the for is terminated and nil is returned, this means that arguments of any finally clause will not be evaluated.

```
1 lisp> (for (in v '(1 0 2))
             (always (> v 0))
             (do (print v)))
1
nil
```

`(never [<form>])`

The square brackets are used to indicate zero or more occurances of <form>. If there is more than one form then the clause is equivalent to (NEVER (or [<form>])). Equivalent to (ALWAYS (not [<form>])).

`(THEREIS <form>)`

If the argument <form> evaluates to a non-nil value then the for is terminated, the value of <form> is the return value.

```
1 lisp> (for (in v '(-1 0 2))
             (thereis (and (zerop v) v))
             (do (print v)))
-1
0
```

`(WHILE [<form>])`

The square brackets are used to indicate zero or more occurances of <form>. If there is more than one form then the clause is equivalent to (WHILE (and [<form>])). Loop iteration stops if any <form> evaluates to nil.

```
1 lisp> (for (from v 2 nil -1)
             (while (> v 0))
             (collect (sqrt v))
             (finally (print v)))
0
(1.41421 1.0)
```

`(UNTIL [<form>])`

The square brackets are used to indicate zero or more occurances of <form>. If there is more than one form then the clause is equivalent to (UNTIL (or [<form>])). Loop iteration stops if any <form> evaluates to a non nil value.

```
(WHEN <form>)
  Jump to the next iteration if the value of <form> is nil.

  1 lisp> (for (in v '(-2 2))
               (when (> v 0))
               (do (print (sqrt v))))
  1.41421
  nil

(UNLESS <form>)
```

Jump to the next iteration if the value of <form> is non-nil.
The evaluation of a for expression follows a specific order, irregardless of the order in which you place clauses.

1. Bind loop variables to their initial values. Each of the expressions which represent initial values is evaluated before the variables are bound.

2. If an initially clause is present, evaluate its arguments.

3. Check for termination. A terminating condition may be specified by an in, on, from, while, until, thereis, always, or never clause. Satisfaction of a condition will force control to step 4 (unless the condition was specified by always or never).

4. If present, check when and unless clauses. If any condition is not satisfied then evaluate the body. The body is constructed from the clauses do, collect, adjoin, adjoinq, join, conc, union, unionq, intersection, intersectionq, count, sum, product, maximize, minimize, maximal, and minimal. Continue at step 3.

5. If an finally clause is present, evaluate its arguments. Return the value of the last argument unless a returns or returning clause is present. Otherwise evaluate their arguments, return the value of the last.

(for* [S:form]): any *macro*

Identical to for except that variable bindings and updates are done sequentially instead of in parallel

7.3.2 Mapping Functions

The mapping functions long familiar to LISP programmers are present in PSL. However, we believe that the for construct described above or the simpler foreach described below is generally more useful, since it obviates the usual necessity of constructing a lambda expression, and is often more transparent. Mapping functions with more than two arguments are not currently supported. Note however that several lists may be iterated along with for, and with considerably more generality. For example:

```
(let ((i 0))
  (mapcar l (function (lambda (x)
                        (setq i (add1 i))
                        (cons i x)))))
```

may be expressed more transparently as

```
(for (in x l) (from i 1) (collect (cons i x)))
```

To augment the simpler for loop present in basic PSL the following list iterator has been provided:

(foreach U:any): any *macro*

This macro is essentially equivalent to the the map functions as follows:
Possible forms are: Setting x to successive elements of u:

```
(foreach x in u do (foo x)) --> (mapc u 'foo)
(foreach x in u collect (foo x))--> (mapcar u 'foo)
(foreach x in u conc (foo x))       --> (mapcan u 'foo)
(foreach x in u join (foo x))       --> (mapcan u 'foo)
```

Setting x to successive cdrs of u:

```
(foreach x on u do (foo x)) --> (map u 'foo)
(foreach x on u collect (foo x))--> (maplist u 'foo)
(foreach x on u conc (foo x))       --> (mapcon u 'foo)
(foreach x on u join (foo x))       --> (mapcon u 'foo)
```

Within the context of for the JOIN is used to append successive values. However, inside foreach successive values are concatenated together.

```
1 lisp> (setq x '(a b c) y (1 2 3))
(1 2 3)
2 lisp> (for (in u '(x y)) (join (eval u)))
```

```
(A B C 1 2 3)
3 lisp> x
(A B C)
4 lisp> (foreach u in '(x y) join (eval u))
(A B C 1 2 3)
5 lisp> x
(A B C 1 2 3)
```

(map X:list FN:function): nil *expr*

Applies FN to successive cdrs of X. The first value passed to FN is X, unless X is nil.

```
1 lisp> (map '(one two) #'print)
(one two)
(two)
nil
```

(mapc X:list FN:function): nil *expr*

FN is applied to successive elements of list X, nil is returned.

```
1 lisp> (mapc '(one two) #'print)
one
two
nil
```

(mapcar X:list FN:function): list *expr*

Similar to mapc except that the return value is a list of the results of each applicaton of FN.

```
1 lisp> (map '((one) (two)) #'(lambda (i) i))
(one two)
```

(mapcan X:list FN:function): list *expr*

Similar to mapcar except that the values returned by FN are nconced together instead of being listed together. The argument X may be modified to construct the result.

```
1 lisp> (setq this '((one) (two)))
((one) (two))
```

```
2 lisp> (mapcan this #'(lambda (i) i))
(one two)
3 lisp> this
((one two) (two))
```

(maplist X:list FN:function): list *expr*

Similar to map except that the return value is a list of the results of each application of FN.

```
1 lisp> (maplist '(one two) #'(lambda (i) i))
((one two) (two))
```

(mapcon X:list FN:function): list *expr*

Similar to maplist except that the values returned by FN are nconced together instead of being listed together.

```
1 lisp> (setq this '(one two))
(one two)
2 lisp> (mapcon this #'(lambda (i) (copy i)))
(one two two)
```

Note that the nconc happens as the mapping process proceeds, not afterward. Therefore the result is not the same as nconcing the results of a maplist. Consider what would occur if

```
(mapcon this #'(lambda (i) i))
```

had been used in the example above. Mapcon would apply its second argument to its first argument giving a partial result of (one two). Notice that the first argument to mapcon is eq to this partial result. Now mapcon applies its second argument to (two), the cdr of its first argument. The partial result becomes (one two two). However, the first argument to mapcon has been modified because nconc was used to build the partial result. The value of the first argument is now (one two two). The computation will never terminate, the length of the first argument to mapcon will continue to grow.

Do

(do A:list B:list [S:form]): any *macro*

The general form follows, the square brackets are used to indicate zero or more occurances of an expression.

```
(DO ([<var-form>]) (<exit-test> [<result>]) [<body>])
```

A <var-form> is either an id or a list of the form

```
(<var> <initial> <next>)
```

There are four basic steps in the evaluation of a do form.

1. .....
2. If <exit-test> evaluates to a non-nil value the <result> forms are evaluated and the do is exited, unbinding the local variables. The value of a do is the value of the last <result> unless there are no <result> forms, in which case nil is returned.
3. The body of the do, consisting of the <body> forms, is evaluated in left to right order.
4. The <next> forms are evaluated in parallel, the values are assigned to the corresponding <var>s rather than being bound. Once this is complete the process continues at step 2.

If <next> is omitted, the value of the corresponding <var> is left unchanged during step 4. If both <initial> and <next> are omitted or if the <var-form> is an id then the variable is bound to nil in step 1 and left unchanged during step 4.
The function definition below illustrates the use of do. This function will reverse the order of elements in a list.

```
(de reverse (sequence)
  (do ((local sequence (rest local))
       (result nil (cons (first local) result)))
      ((null local) result)))
```

(do* A:list B:list [C:form]): any *macro*

Do* is like do, except the variable bindings and updatings are done sequentially instead of in parallel. The following is a definition of assoc using do*.

```
(de assoc (item a-list)
  (do* ((local a-list (rest local))
        (pair (first local) (first local)))
       ((or (null local)
            (equal item (first pair)))
        local)))
```

If do had been used, evaluation of the <initial> form (first local) would have resulted in an error. Either local would be unbound or the value of local from the surrounding environment would have been used.

(do-loop A:list B:list C:list [S:form]): any *macro*
Do-loop is like do, except that it takes an additional argument, a prologue. The general form follows, the square brackets are used to indicate zero or more occurances of an expression.

```
(DO-LOOP ([<var-form>]) ([<first>])
         (<exit-test>[<result>]) [<body>])
```

This is executed just like the corresponding do, except that after the initial bindings are established, but before the exit test is first evaluated, the prologue forms, consisting of the <first> forms, are evaluated in a left to right order. Note that all of the <first> forms are evaluated exactly once, assuming that none of the <first> forms causes an error.

(do-loop* A:list B:list C:list [S:form]): any *macro*
Do-loop* does the variable bindings and updates sequentially instead of in parallel.

7.4 Non-Local Exits

The functions catch and throw are very useful for discontinuing a computation. As return provides for local exit, this pair of functions provide for non-local exit. They should not, however, be used indiscriminately. The lexical restrictions on their more local counterparts ensure that the flow of control can be ascertained by looking at a single piece of code. With catch and throw, control may be passed to and from totally unrelated pieces of code.

(catch TAG:id [FORM:form]): any *Open-Compiled fexpr*
Catch evaluates TAG to establish a name for this catcher, called the catch-tag, and then evaluates the FORM's in a protected environment. If during this evaluation a throw occurs with a tag that is the same as the catch-tag (as defined by the function eq), catch immediately returns the result of the form given as the second argument to the throw. If no throw occurs, the results returned by the last FORM are returned as the result of the catch. A catch-tag of nil is considered special, it serves to match any catch-tag specified by throw.

(throw TAG:id VALUE:any): None Returned *expr*

Throw evaluates TAG to produce a catch-tag and evaluates VALUE to produce a result value. At this point, an error is signalled if there is no active catch with the same catch-tag (as determined by the function eq). Otherwise, control is passed to the most recent such catch, and the results of evaluating VALUE become the results of the catch.

In the process of transferring control to the catch, all intervening constructs are exited. Exiting a construct that binds variables has the effect of unbinding those variables.

throwsignal* [Initially: nil] global This fluid variable is set to t if the most recent invocation of Catch was thrown to. throwsignal* is set to nil upon a normal exit from a catch and to t upon a normal exit from a throw.

throwtag* [Initially: nil] global This fluid variable is set to the catch-tag of the most recent throw

The catch, throw pair supply a construct which allows for some control over the evaluation of an expression. Exceptions can be detected during the evaluation of the expression and appropriate action can be taken. The functions which follow define a simple parser. The parse is done inside a catch. If there are no errors then the result of the parse is returned. When an error arises the computation is aborted with a call on throw. An error message is printed prior to aborting the parse.

```
(de parse (*buffer*)
  (catch 'parse-error (list 's (parse-np) (parse-vp))))

(de parse-np ()
  (if (memq (car *buffer*) '(a an the))
    `(np (det ,(pop *buffer*)) (n ,(pop *buffer*)))
    (parse-error "Bad word in noun phrase: %w%n")))

(de parse-vp ()
  (if (memq (car *buffer*) '(sings talks))
    `(vp (v ,(pop *buffer*)))
    (parse-error "Not a verb: %w%n")))


(de parse-error (format-string)
  (throw 'parse-error (printf format-string (car *buffer*))))

1 lisp> (parse '(the bird sings))
(S (NP (DET THE) (N BIRD)) (VP (V SINGS)))
2 lisp> (parse '(the bird eats))
```

```
Not a verb: eats
nil
3 lisp> (parse '(it is small))
Bad word in noun phrase: it
nil
```

The following macros are provided to aid in the use of catch and throw with a nil catch-tag, by examining throwsignal* and throwtag*:

(catch-all HANDLER:function [FORM:form]): any *macro*
Has the same semantics as catch except that all throws, independent of catch-tag, will be caught. The HANDLER must be a function of two arguments. If a throw occurs, the HANDLER will be called on the catch-tag and the value passed by the throw. The HANDLER may itself issue a throw, in which case the catch-all acts as a filter.

(unwind-all HANDLER:function [FORM:form]): any *macro*
This function is very similar to catch-all. However, if no throw occurs the HANDLER will be called on nil and the value returned.

(unwind-protect FORM:form [CLEANUPFORM:form]): any*macro*
The FORM is evaluated and, regardless of how it is exited (a normal return, throw, or error), the CLEANUPFORMs will be evaluated. One common use of unwind-protect is to ensure that a file will be closed after processing.

```
(setq channel (open file ....))
(unwind-protect (process-file)
            (close channel))
```

This primitive can be used to implement arbitrary kinds of state-binding without fear that an unusual return (an error or throw), will violate the binding.

```
(defmacro bind ((name value) . body)
  (let ((old-value (gensym)))
    `(let ((,old-value ,name))
       (unwind-protect
    (progn (setq ,name ,value)
           ,@body)
    (setq ,name ,old-value)))))
```

```
1 lisp> (setq number 5)
5
2 lisp> (bind (number 2) (print number) (/ number 0))
2
***** Attempt to divide by zero in Quotient
3 lisp> number
5
```

Note: Certain special tags are used in the PSL system, and should not be interfered with casually:

$error$	Used by error and errorset which are implemented in terms of catch and throw, (see Chapter 16).
$unwind-protect$	A special catch-tag placed to ensure that all throws pause at the unwind-protect "mark".
$prog$	Used to communicate between interpreted progs, gos.

Function Definition and Binding

8.1 Function Definition in PSL

Functions in PSL are global entities. To avoid function-variable naming clashes, the Standard LISP Report required that no variable have the same name as a function. There is no conflict in PSL, as separate function cells and value cells are used.

The first major section in this chapter describes how to define new functions; the second describes the binding of variables in PSL. The final section presents binding functions useful in building new interpreter functions.

8.1.1 Function Types

Eval-type functions are those called with evaluated arguments. NoEval functions are called with unevaluated arguments. Spread-type functions have their arguments passed in a one-to-one correspondence with their formal parameters. NoSpread functions receive their arguments as a single list.

There are four function types implemented in PSL:

expr An Eval, Spread function, with a maximum number of arguments. The maximum depends upon the implementation. In referring to the formal parameters we mean their values. Each function of this type should always be called with the expected number of parameters, as indicated in the function definition.

fexpr A NoEval, NoSpread function. There is no limit on the number of arguments. In referring to the formal parameters we mean the unevaluated arguments, collected as a single list, and passed as a single formal parameter to the function body.

nexpr An Eval, NoSpread function. Each call on this kind of function may present a different number of arguments, which are evaluated, collected into a list, and passed in to the function body as a single formal parameter.

macro The macro is a function which creates a new S-expression for subsequent evaluation or compilation. There is no limit to the number of arguments a macro may have. The descriptions of the eval and expand functions in Chapter 11 provide precise details.

8.1.2 Notes on Code Pointers

A code-pointer may be displayed by the print functions or expanded by explode. The value appears in the convention of the implementation e.g. (#<Code:A N>, where A is the number of arguments of the function, and N is the function's entry point). A code-pointer may not be created by compress. (See Chapter 12 for descriptions of explode and compress.) The code-pointer associated with a compiled function may be retrieved by getd and is valid as long as PSL is in execution. Normally a code-pointer is stored using putd. It may be checked for equivalence by eq. The value may be checked for being a code-pointer by the codep function.

8.1.3 Functions Useful in Function Definition

In PSL, ids have a function cell that usually contains an executable instruction which either JUMPs directly to the entry point of a compiled function or executes a CALL to an auxiliary routine that handles interpreted functions, undefined functions, or other special services (such as auto-loading functions, etc). The user can pass anonymous function objects around either as a code-pointer, which is a tagged object referring to a compiled code block, or a lambda expression, representing an interpreted function.

(putd Fname:id TYPE:ftype BODY: lambda,code-pointer):id *expr*

Creates a function with name Fname and type TYPE, with body as the function definition. If successful, putd returns the name of the defined function.

If the body is a lambda there are two possible outcomes. When the switch comp is set to t the body will be compiled and a special instruction to jump to the start of the code is placed in the function cell. If the switch *comp is set to nil then the body is saved on the property list under the indicator *lambdalink and a call to an interpreter function

```
(lambdalink) is placed in the function cell.
```

If the body is a code-pointer then a special instruction to jump to the start of the code is placed in the function cell.

The Type is recorded on the property list of Fname if it is not an expr.

After using putd on Fname, getd will return a pair which specifies the type and the body of the definition.

The following switches are useful when defining functions.

***redefmsg** = [Initially: t] *switch*

If *redefmsg is not nil, the message
`*** Function 'FOO' has been redefined`
is printed whenever a function is redefined.

***usermode** = (Initially: t) *switch*

Controls action on redefinition of a function. All functions defined when *usermode is t are flagged USER. Functions which are flagged USER can be redefined freely. If an attempt is made to redefine a function which is not flagged USER, the query
`Do you really want to redefine the system function FOO?`
is made, requiring a Y, N, YES, NO, or B response. B starts the break loop, so that one can change the setting of *usermode. After exiting the break loop, one must answer Y, Yes, N, or No (See yesp in Chapter 12). If *usermode is nil, all functions can be redefined freely, and all functions defined have the USER flag removed. This provides some protection from redefining system functions.

***comp** = [Initially: nil] *switch*

The value of *comp controls whether or not putd compiles the function before defining it. If *comp is nil the function is defined as a lambda expression. If *comp is non-nil, the function is first compiled. Compilation produces certain changes in the semantics of functions, see Chapter 19 for information.

(getd U:any): nil, pair *expr*

If U is not the name of a defined function, nil is returned. otherwise (expr, fexpr, macro, nexpr . code-pointer, lambda) is returned.

(copyd NEW:id OLD:id): NEW:id *expr*

Normally the function body and type of NEW become the same as those of OLD. However, if the switch *comp is set to t and the body of OLD is not compiled then NEW will be set to the compiled version of the body of OLD. If no definition exists for OLD an error:

```
***** OLD has no definition in COPYD
```

is given. NEW is returned.

(remd U:id): nil, pair *expr*

Removes the function named U from the set of defined functions. Returns the (ftype . function) pair or nil, as does getd. If the function type is not expr then it was recorded on the property list when the function was defined. In such cases this function removes the type information from the property list.

8.1.4 Function Definition in LISP Syntax

The functions de, df, dn, dm, and ds are used in PSL to define functions and macros. The functions are compiled if the compiler is loaded and the switch comp is set to t.

(de Fname:id PARAMS:id-list [FN:form]): id *macro*
Defines the function named Fname, of type expr. The forms FN are made into a lambda expression with the formal parameter list PARAMS, and this is used as the body of the function.
Previous definitions of the function are lost. The name of the defined function, Fname, is returned.
The COMMON module defines the macro DeFun which is equivalent to de.

(df Fname:id PARAM:id-list [FN:form]): id *macro*
Defines the function named Fname, of type fexpr. The forms FN are made into a lambda expression with the formal parameter list PARAM, and this is used as the body of the function. The parameter list should only contain one parameter.
Previous definitions of the function are lost. The name of the defined function, Fname, is returned.

(dn Fname:id PARAM:id-list [FN:form]): id *macro*
Defines the function named Fname, of type nexpr. The forms FN are made into a lambda expression with the formal parameter list PARAM, and this is used as the body of the function. The parameter list should only contain one parameter.
Previous definitions of the function are lost. The name of the defined function, Fname, is returned.

(dm Mname:id PARAM:id-list [FN:form]): id *macro*
Defines the function named Fname, of type macro. The forms FN are made into a lambda expression with the formal parameter list PARAM, and this is used as the body of the function. The parameter list should only contain one parameter.
Previous definitions of the function are lost. The name of the defined function, Fname, is returned.

The function list can be defined as follows

```
(dm list (a)
  (if (< (length a) 2) ()
    (cons 'cons
          (cons (second a)
                (ncons (cons 'list (rest (rest a))))))))
```

Now consider what occurs during the evaluation of (list 1 2). The list (list 1 2) is passed to the list macro which returns (cons 1 (list 2)) for further evaluation. The evaluator will call the list macro again for (list 2). This second call on the macro will return (cons 2 (list)). Finally, (list) is transformed into nil by a third call on the list macro and the entire process will terminate after evaluating (cons 1 (cons 2 nil)). Notice the lack of distinction between program and data The data structure representation of the function call is passed to the function as its parameter.

(ds Sname:id PARAMS:id-list [FN:form]): id *macro*

Defines the function named Sname of type macro. The syntax of ds is similar to that of de except that a macro is defined instead of an expr.

Perhaps the behavior of this function is best described with an example. The evaluation of

```
(ds first (x) (car x))
```

will generate an expression similar to

```
(dm first (a)
  (prog (x)
        (setq a (cdr a))
        (setq x (car a))
        (return (list 'car x))))
```

which is then evaluated. A sequence of assignment statements are created to initialize each parameter to its corresponding argument. Each id within the body FN which is not in the parameter list is quoted. It is an error to call a macro defined this way with more arguments than are specified by the parameter list. An error of this type will cause the message

```
***** Argument mismatch in SMacro expansion
```

to be printed. When a call is made without enough arguments the additional parameters are set to nil.
The following macro utilities are in the **useful** module.

(defmacro A:id B:form [C:form]): id *macro*

Defmacro is a useful tool for defining macros. The form of an application of defmacro is given below, square brackets are used to indicate zero or more occurances of an expression.

```
(DEFMACRO <name> <pattern> [<form>])
```

The pattern is an list, each element is either an id or a pair. All of the ids in the pattern are local variables which may be used freely in the body (the ¡form¿s). When the macro is called the pattern is matched against the cdr of the macro call, binding the ids of the pattern to their corresponding parts of the call. Once the binding is complete the body is evaluated, the result is the value of the last form. Defmacro is often used with backquote. The following examples illustrate the use of defmacro. The first is intended to provide a contrast with ds.

```
(defmacro car (s)
  `(car ,s))

(defmacro nth (s i)
  (if (onep i)
    `(car ,s)
    `(nth (cdr ,s) ,(sub1 i))))
```

(deflambda name:id PARAMS:id-list [FN:form]): id *macro*

Defines the function named name, of type macro. Deflambda is similar to ds except that the body of the definition is enclosed within a lambda expression. The number of parameters of the lambda expression is the same as the number specified by PARAMS. When the macro is applied each of the arguments are evaluated and then bound to the parameters of the lambda expression. Finally, the body of the lambda is evaluated. Note that each argument is evaluated once. This may not be true had the macro been defined using ds. The following example illustrates when deflambda should be used in place of ds. The expansion of the first version of Check contains two occurances of (long-calculation n), where the expansion of the second version only contains one.

```
(ds check (any)
  (when (> any 0) (compute any)))

(deflambda check (any)
  (when (> any 0) (compute any)))

(check (long-calculation n))

(when (> (long-calculation n) 0)    % Expansion of the first
  (compute (long-calculation n)))   % version of check

((lambda (any)                      % Expansion of the second
```

```
    (when (> any 0)                    % version of check
      (compute any)))
  (long-calculation n))
```

8.1.5 BackQuote

(backquote A:form): form *macro*

Backquote is the function name for ‘ (accent grave). With some printers it may be difficult to distinguish between the quote and accent grave characters. For this reason, in the examples which follow 'expression is written as (quote expression). You must load USEFUL to define backquote.
In the previous section the function list was defined as a macro.

```
(dm list (a)
  (if (< (length a) 2)
    ()
    (cons (quote cons)
          (cons (second a)
                (ncons (cons (quote list)(rest (rest a))))
))))
```

The body of this definition is somewhat difficult to read. An abbreviated syntax was developed to aid both the writer and the reader. The notation, called "backquote" (‘), works as an "anti-quote". Unquoted forms within its scope are assumed to be constants. To indicate that a form should be evaluated, one should prefix the form with a comma (,). Consider the redefinition of the macro list.

```
(dm list (a)
  (if (< (length a) 2)
    ()
    ‘(cons ,(second a)
           ,(cons (quote list) (rest (rest a))))))
```

While this is an improvement, the explicit construction ,(cons (quote list) (rest (rest a))) clutters up the appearance. The application of cons is there to indicate that we want to combine the result of (rest (rest a)) with the identifier list. The form ,(list (rest (rest a)) will not give this effect, instead it would create a list of two elements. The at-sign (@) is used in conjunction with the comma to mean "splice-in" rather than "cons-in". This results in the following defintion.

```
(dm list (a)
  (if (< (length a) 2)
    ()
    `(cons ,(second a) (list ,@(rest (rest a))))))
```

(unquote A:any): Undefined *fexpr*

Function name for comma ,. It is an error to eval this function; it should occur only inside a backquote.

(unquotel A:any): Undefined *fexpr*

Function name for comma-atsign ,@. It is an error to eval this function; it should only occur inside a backquote.

The two examples which follow are similar definitions of an arithmetic if form. There are four arguments: a test form, a negative form, a zero form, and a positive form. One of the last three forms is chosen to be evaluated according to whether the test form is positive, negative, or zero. The first uses the traditional dm while the second uses defmacro and backquote. Clearly the second is much simpler. To clarify printer output, there are no occurances of the accent grave character in the first definition and the second definition does not contain an occurance of the quote character.

```
(dm number-if (calling-form)
  (list 'let
        '((var (gensym)))
        (list 'let
              (list (list 'var (nth calling-form 2)))
              (list 'cond
                    (list (list 'minusp 'var)
                          (nth calling-form 3))
                    (list (list 'zerop 'var)
                          (nth calling-form 4))
                    (list t (nth calling-form 5))))))

(defmacro number-if (test minus-form zero-form plus-form)
  (let ((var (gensym)))
    `(let ((,var ,test))
       (cond ((minusp ,var) ,minus-form)
             ((zerop ,var) ,zero-form)
             (t ,plus-form)))))
```

8.1.6 MacroExpand

(macroexpand A:form [B:id]): form *macro*
Macroexpand is a useful tool for debugging macro definitions. If given one argument, macroexpand expands all the macros in that form. Often one wishes for more control over this process. For example, if a macro expands into a let, we may not wish to see the let itself expanded to a lambda expression. Therefore additional arguments may be given to macroexpand. If these are supplied, they should be the names of macros, and only those specified are expanded.

Low Level Function Definition Primitives

The following functions are used especially by putd and Getd, defined above in Section 9.1.3, and by eval and apply, defined in Chapter 11.

(funboundp U:id): boolean *expr*
Tests whether there is a definition in the function cell of U; returns nil if so, t if not.
Note: The functional value of an identifier which does not define a function is actually a call on undefinedfunction. The evaluation of undefinedfunction will result in a continuable error.

(fboundp U:id): boolean *macro*
Equivalent to (not (funboundp U)), the function funboundp is described above.

(flambdalinkp U:id): boolean *expr*
Tests whether U is an interpreted function; return t if so, nil if not. This is done by checking for the special code-address of the lambdalink function, which calls the interpreter.

(fcodep U:id): boolean *expr*
Tests whether U is a compiled function; returns t if so, nil if not.

(makefunbound U:id): nil *expr*
Makes U an undefined function by planting a special call to the function, undefinedfunction, in the function cell of U. See funboundp above for more information about undefinedfunction.

(makeflambdalink U:id): nil *expr*
Makes U an interpreted function by planting a special call to an interpreter support function (lambdalink) function in the function cell of U.

(makefcode U:id C:code-pointer): nil *expr*
Makes U a compiled function by planting a special JUMP to the code-address associated with C.

(getfcodepointer U:id): code-pointer *expr*
Gets the code-pointer for U.

(code-number-of-arguments C:code-pointer): {nil,integer} *expr*
Some compiled functions have the argument number they expect stored in association with the code-pointer C. This integer, or nil is returned.

8.1.7 Function Type Predicates

(exprp U:any): boolean *expr*
Test if U is a code-pointer, lambda form, or an id with expr definition.

(fexprp U:any): boolean *expr*
Test if U is an id with fexpr definition.

(nexprp U:any): boolean *expr*
Test if U is an id with nexpr definition.

(macrop U:any): boolean *expr*
Test if U is an id with macro definition.

8.2 Wrappers

Once a function has been defined, one may want it to do something a little different, or just a little bit more. Breaking and tracing functions for debugging purposes can be thought of in this way. Having a function keep track of the time spent in its execution can also be thought of in this way. The Wrappers module provides a convenient facility to support this.
When a function is wrapped, its name becomes a reference for two different function definitions. In PSL it is possible to create distinct identifiers which have the same name. The original name of the function is associated with a new function called the wrapper. In general a wrapper does additional work before and/or after applying the original definition. The original definition is associated with an identifier whose name is identical with the original name but which is not interned. Application of getd to the original name will return the original definition. Since a wrapper is identified by an indicator on the property list of its name getd knows when to look elsewhere for the original function definition, and putd knows to alter the original definition, not the wrapper.
Typically a wrapper may be added to a function and later removed. A function potentially may be wrapped up inside more than one wrapper at the same time. It is possible to redefine a wrapped function but if the order or number of formal parameters is changed then it will be necessary to unwrap all wrappers first.

8.2.1 Notes on Writing Wrappers

This section describes guidelines for writing wrappers.
If a wrapper is put on a function that is used either directly or indirectly by the wrapper body an infinite recursion may result. Functions used by the PSL interpreter are particularly susceptible to this problem. We require a means to avoid infinite recursion. In general it is not easy to restrict the set of functions which can be called indirectly by a piece of code. Many PSL system functions use other system functions. Furthermore, a modification to the system may change these relationships. Some system functions call functions through functional variables or functional values stored on property lists. This means that some of the relationships between system functions change over time.
Instead of avoiding recursion introduced by wrappers, we can detect and

recover from it. Wrapper bypasses do this. A wrapper bypass is implemented through a fluid variable, called the controlling variable for that wrapper. On entry to a wrapper, the value of the controlling variable is checked. A non-nil value implies that a previous invocation of this wrapper has not been completed. To avoid a recursive call on the wrapper, a call on the wrapped function replaces evaluation of the wrapper.
If the value of the controlling variable is nil the wrapper is evaluated. During evaluation of the wrapper the controlling variable is set to t except during the application of the wrapped function, at which point it should be set to nil.
Syntactically, wrapper bodies of the following form are supported:

```
(lambda <args>
  (cond (<var> <wrapped-fn-call>)
        (t (prog (<var> <id>*)
             (setq <var> t)
             <call-expr>*))))

<call-expr> ::= <expr>
                | (<call-expr>*)
                | (setq <var> nil) <setq-and-wrapped-fn-call> (setq <var> t)

<setq-and-wrapped-fn-call> ::=
               | <wrapped-fn-call>
               | (setq <id> <wrapped-fn-call>)

<wrapped-fn-call> ::= (wrapped-function <expr>*)
                  | (funcall 'wrapped-function <expr>*)

<var> ::= <id>
<id> is any PSL identifier <expr> is any PSL s-expression.
```

The controlling variable of the wrapper is $< var >$. It must be a fluid variable whose top level value is nil.
The notation is BNF with the addition of the regular expression "*" operator. This operator is used to indicate zero or more repetitions of an item.
Note that setf is not supported in place of setq. You should not use apply in place of funcall or let in place of prog. You can eliminate redundant instances of (setq $< var >$ t) and (setq $< var >$ nil). For example, if a `<wrapped-fn-call>` appears at the beginning of a prog then the form `(setq <var> nil)` can be omitted.

8.2.2 Exported Functions

(wrap name:id wtype:id BODY:list COMPILE?:boolean): id *expr*
Wrap creates a wrapper. name is the name of the function to be wrapped. wtype is an id which identifies the wrapper type, it must be a member of the list wrapper-standard-order (documented below). BODY is a list which should be a lambda expression.
Wrap redefines the functional value of name. The original function is renamed. The print representation of this new name is identical to name but the identifiers are distinct.

When BODY is a lambda expression (you are encouraged to use a lambda expression for this argument), then the new name of the function which is being wrapped is substituted for each occurance of wrapper-function. The result of this substitution becomes the definition of the wrapper.

When BODY is not a lambda expression a different method is used to create the wrapper. BODY is embedded within a lambda expression which has the same number of arguments as the function which is being wrapped. Each occurance of the form (wrapped-invocation), is replaced by an application of the wrapped function.

If COMPILE? is t then the wrapper will be compiled.

The wrappers package permits a function to have any number of wrappers of any permitted type. For a given type of wrapper, a new wrapper always encloses preexisting wrappers of that type. However, you are advised against depending upon this ordering.

(remove-wrapper name:id TYPE:id): boolean *expr*
Removes a wrapper of type TYPE from the identifier name. If name does not have a wrapper of this type then the return value will be nil, otherwise t is returned.

(wrapped? name:id): boolean *expr*
Returns t if there is at least one wrapper on name.

(wrapper-of-type? name:id TYPE:id): name:id nil *expr*
Searches for a wrapper of the given type on name. name is returned if a wrapper is found, otherwise nil is returned.

(wrapper-types name:id): list *expr*
Returns a list of all the types of all the wrappers around the function named name.

(function-lambda-list FN:id, code-pointer, lambda): list *expr*
Returns a list suitable for use as the formal parameter list of a wrapper for that function. If the function is interpreted, the argument list of the definition is returned. If the actual argument list is not available, names for the arguments are invented. If FN is not a lambda expression, code-pointer, or identifier then it is an error.

```
***** FN is not a function.
```

It is also an error if FN is an identifier which does not have a functional value.

```
***** FN is not defined as a function.
```

(function-basic-definition name:id): name:id *expr*
Returns the basic definition of name, whether or not a set or wrappers is associated with it. Note that getd and putd operate on this basic definition. If you wish to redefine a wrapped function and the number or order of formal parameters will change it is necessary to first unwrap all wrappers first. Applications of the wrapped function within the wrapper will not be updated when the wrapped function is redefined.

wrapper-standard-order = [Initially: (trace break meter)] *global*
Only wrapper types which are members of this list are legal arguments to primitive functions that operate on wrappers. The wrappers of each function are nested so that each wrapper type appears earlier in wrapper-standard-order than the type of any wrapper it encloses. Wrap wraps according to this ordering, but changing this list does not cause the ordering of existing wrappers to be changed.

8.2.3 Examples

Assume that foo is a commonly called function whose argument is a large data structure. You have found that foo is sometimes called with an argument of nil and would like to cause a break when this situation arises so that you can discover where foo is being called from.
Since foo is a commonly used function a break on each entry to foo is unsatisfactory. A trace of each entry and exit is not sufficient – it doesn't tell

you who called foo in the critical case. The solution is to wrap foo is an advice wrapper. Such a wrapper will print a backtrace and enter a continuable break loop when an argument of nil is discovered.
First, load the **break-trace** module. This will result in the **wrappers** module being loaded and the breaktrace function being defined (see Chapter 17 for more information on breaktrace). The wrapper-type advice is added as the innermost wrapper on the ordered list referenced by wrapper-standard-order (assuming it is not already on the list).

```
(load break-trace)                % loads wrappers &
                                    breakpoint function
(if (not member 'advice wrapper-standard-order) then
  (setf wrapper-standard-order
        (append wrapper-standard-order '(advice))))

(de debug-foo ()
  (wrap
   'foo                           % function to wrap
   'advice                        % wrapper type
   `(lambda (x)                   % wrapper body generator
      (cond ((null x)
             (backtrace)
             (breakpoint "Foo (x=nil)")))
      (wrapped-function x))
   nil))                           % don't compile
                                   % the wrapper body
```

The above will work fine as long as foo is an expr which is not used by the interpreter or by the **wrappers** module itself. In the more general case, the function debug-foo must have three changes:

1. If foo is not an expr, then all occurrences of (wrapped-function $< args >$) must be replaced by (funcall 'wrapped-function $< args >$).

2. If foo is used by the PSL interpreter, by the `wrappers` module, or by expression in the wrapper body then it will be necessary to know when foo is called during the evaluation of foo's wrapper. This will allow you to bypass evaluation of the wrapper, you can simply apply the wrapped function. Otherwise an infinite recursion could result. The definition below uses the fluid variable (called a controlling variable) in-foo-wrapper? to achieve this effect.

3. If foo is possibly used within the interpreter, then the wrapper must be compiled. This means that if (not (interpretive-wrapper-ok? 'foo)),

then the fourth argument to the function wrap must be t, or wrap will signal an error.

The revised definition of debug-foo follows.

```
(fluid  '(in-foo-wrapper?))     % necessary initializations of
  (setq in-foo-wrapper? nil)      % the controlling  variable
(change 2)

  (de debug-foo ()
    (wrap
     'foo
     'advice
     `(lambda (x)
        (cond (in-foo-wrapper?                    % change 2
               (funcall 'wrapped-function x))   % change 1
              (t
               (prog (in-foo-wrapper?)
                 (setq in-foo-wrapper? t)
                 (cond ((null x)
                        (backtrace)
                        (breakpoint "Foo (x=nil)")))
                 (setq in-foo-wrapper? nil)
                 (return
                  (funcall 'wrapped-function x) % change 1
                 ))))
     (not (interpretive-wrapper-ok? 'foo))))    % change 3
```

The function break-on-condition will cause entry into a continuable break loop just before the function fn is applied if the result of evaluating the form bound to bool-expr is non-nil. A sample call might be (break-on-condition 'foo '(eq x 2)).

```
(setf in-wrapper? nil)
(fluid '(in-wrapper?))

(de break-on-condition (fn bool-expr)
    % First, get the parameter-list of the function.
  (let ((parameter-list (function-lambda-list fn)))
    (wrap
     fn
     'advice
     `(lambda ,parameter-list
```

```
        (cond (in-wrapper?
                (funcall 'wrapped-function ,@parameter-list))
              (t
               (prog (in-wrapper?)
                 (setq in-wrapper? t)
                 (cond (,bool-expr
                        (breakpoint "In %p, condition %p=%p"
                          ',fn  ',bool-expr ,bool-expr)))
                        (setq in-wrapper? nil)
                        (return
                         (funcall 'wrapped-function
                                  ,@parameter-list))
                 ))))
    (not (interpretive-wrapper-ok? fn))
    )
  ))
```

8.3 Variables and Bindings

Variables in PSL are ids, and associated values are usually stored in and retrieved from the value cell of this id. If variables appear as parameters in lambda expressions or in prog's, the contents of the value cell are saved on a binding stack. A new value or nil is stored in the value cell and the computation proceeds. On exit from the lambda or prog the old value is restored. This is called the "shallow binding" model of LISP. It is chosen to permit compiled code to do binding efficiently. For even more efficiency, compiled code may eliminate the variable names and simply keep values in registers or a stack. The scope of a variable is the range over which the variable has a defined value. There are three different binding mechanisms in PSL.

Local Binding	Only compiled functions bind variables locally. Local variables occur as formal parameters in lambda expressions and as prog form variables. The binding occurs as a lambda expression is evaluated or as a prog form is executed. The scope of a local variable is the body of the function in which it is defined.
Fluid Binding	Fluid variables are global in scope but may occur as formal parameters or prog form variables. In interpreted functions, all formal parameters and local variables are considered to have fluid binding until changed to local binding by compilation. A variable can be treated as a fluid only by declaration. If fluid variables are used as parameters or locals they are rebound in such a way that the previous binding may be restored. All references to fluid variables are to the currently active binding. Access to the values is by name, going to the value cell.
Global Binding	In theory global variables may never be used as parameters in lambda expressions or as prog variables. This restriction is not enforced in PSL. You are encouraged not to declare an identifier global if it is used as a parameter in a lambda expression or prog. For more information see the Section Fluid and Global Declarations, Chapter 19.

8.3.1 Binding Type Declaration

(fluid IDLIST:id-list): nil *expr*

The ids in IDLIST are declared as fluid type variables. An id which has not been previously declared is initialized to nil. Variables in IDLIST already declared fluid are ignored. It is an error to attempt to change the type of a variable from global to fluid.

```
***** ID cannot be changed to FLUID
```

(global IDLIST:id-list): nil *expr*

The ids of IDLIST are declared global type variables. If an id has not been previously declared, it is initialized to nil. Variables which have already been declared global are ignored. Attempting to change the type of an id from fluid to global will result in an error.

```
***** ID cannot be changed to GLOBAL
```

(unfluid IDLIST:id-list): nil *expr*
The variables in IDLIST which have been declared as fluid are no longer considered fluid. Other variables are ignored.

8.3.2 Binding Type Predicates

(fluidp U:id): boolean *expr*
If U has been declared fluid then t is returned, otherwise nil is returned.

(globalp U:id): boolean *expr*
If U has been declared global then t is returned, otherwise nil is returned.

(unboundp U:id): boolean *expr*
If U does not have a value then t is returned, otherwise nil is returned.

8.4 User Binding Functions

The following functions are available to build one's own interpreter functions that use the built-in Fluid binding mechanism, and interact well with the automatic unbinding that takes place during Throw and Error calls.

(unbindn N:integer): Undefined *expr*
Used in user-defined interpreter functions (like prog) to restore previous bindings to the last N values bound.

(lbind1 IDname:id VALUETOBIND:any): Undefined *expr*
Support for lambda-like binding. The current value of IDname is saved on the binding stack; the value of VALUETOBIND is then bound to IDname.

(pbind1 IDname:id): Undefined *expr*
Support for prog. Binds nil to IDname after saving value on the binding stack. Essentially (lbind1 IDname nil).

The Interpreter

9.1 Evaluator Functions Eval and Apply

The evaluator is responsible for the execution of PSL programs. The evaluator is available to the user through the function eval. This function gives the operational semantics, or meaning, to the programming constructs found in PSL. With eval we have the ability to evaluate constructed expressions. The ability to evaluate lists which have the appearance of expressions highlights the program data duality of LISP.
Apply represents the piece of the evaluator which is responsible for invoking functions. Any list which is not quoted is taken to be a function application by the evaluator. Apply provides the user with a tool to explicity invoke functions.
In eval, apply and the support functions which follow, errors are continuable. When an error occurs the user is permitted to correct the offending expression or perhaps define a missing function (see Chapter 16 for more information). If the number of arguments to a function is not equal to the number of parameters specified by the definition then the error

```
***** Argument number mismatch
```

occurs. If there is an id in the function position of a form, and there is no function definition associated with it, an error occurs and the message

```
***** 'NAME' is an undefined function
```

is printed. If the function position is not occupied by an id, lambda expression or code-pointer then it is an error and one of the following is printed.

```
***** Ill-formed expression in EVAL 'FORM'
```

```
***** Ill-formed expression 'FORM'
```

The first message is displayed when the error is detected within eval.

(eval U:form): any *expr*

The value of the form U is computed. Since eval is a function of type expr it will receive its argument already evaluated. Since it will then evaluate the argument itself, it may seem as though the evaluation is done twice. The argument U cannot access local variables.

As a basis for discussion, an approximation to the actual definition of eval is given.

```
(de eval (e)
  (cond ((is-constant e) (denote e))
        ((idp e) (valuecell e))
        (t (let ((fun (first e))
                 (args (rest e)))
             (cond ((and (idp fun)
                         (not (funboundp fun)))
                    (selectq (first (getd fun))
                      (expr (apply fun (evlis args)))
                      (fexpr (apply fun (list args)))
                      (nexpr
                        (apply fun
                               (list (evlis  args))))
                      (macro
                        (eval (apply fun (list e))))))
                   ((or (codep fun)
                        (is-lambda fun))
                    (apply fun (evlis args)))
                   (t (error)))))))
```

Eval first determines if its argument is a constant. Examples of constants are numbers, strings, vectors and quoted expressions. For the most part, the function denote will simply return its argument. However, if the constant is a quoted expression then the expression is returned as value.

```
1 lisp> (eval "STRING")
"STRING"
2 lisp> (eval (mkquote '(one)))
(ONE)
```

An identifier is considered a variable. In such a case, eval returns the contents of the value cell of the identifier. Of course things are not quite all that simple. This raises a conceptual issue about when to find the values. The issue is one of scoping rules. Scoping rules come into play when functions are defined. In particular, this involves free variables (those variables which

are not parameters of the definition). One approach is static scoping. This strategy locates the values of free variables at the time the function is defined. PSL uses a second approach called dynamic scoping. Using this approach, the values of free variables are not determined until the function is applied.

```
1 lisp> (setq number 0)
0
2 lisp> (de bar () (foo) (add1 number))
BAR
3 lisp> (de foo () (setq number 1))
FOO
% When bar was defined the value of number was 0,
% during the evaluation
% of bar the application of foo changed the value
% of number to 1. If
% static scoping were being used then the result
% would be 1 instead of 2.
4 lisp> (bar)
2
```

The PSL environment (the values associated with variables) will typically change during evaluation. When a function is applied, the variables specified by its definition are associated with the arguments of the application. As a result, the value of each variable has changed. As noted above, the value of a variable is found in its value cell. However, the definition and access of values depends upon the binding strategy being used by the implementation. Two common techniques are shallow binding (used in the current implementation of PSL), and deep binding. In a deep bound system the environment is defined by a table of names and values. Each time a variable is set to a value, an entry is added to the table. To locate the value of a variable this table must be searched. When there are multiple definitions for a given variable within this table, the convention is adopted that it is always the latest value which is retrieved. With this strategy we can easily restore an old value, by simply removing the current value from the table. With shallow binding, the value of a variable is positioned with the id which represents the variable. In this case there is no need to search for a value. It is known to always be in a fixed location, the value cell of the id. This second technique makes finding values much more efficient, but it is much more difficult now to keep track of previous values.

If the expression being evaluated is neither a constant or a variable it is assumed to be a function application form. When we apply the function definition to the arguments we associate the parameters of the definition with

the arguments. This process of association is called binding and simulates substitution. Within the new environment the body of the function is evaluated. Notice we do not explicitly substitute the values for the variables in an expression.

```
1 lisp> (eval '((lambda (this) (add1 this)) 0))
1
% If an expression is neither a constant or
% a variable then it is assumed
% to be a function application form.
2 lisp> (eval '(one))
***** 'ONE' is an undefined function
```

Eval first looks at the function position, making sure a function definition is available. The manner in which the function is applied depends upon the type of the function. For both exprs and nexprs the arguments are evaluated in a left to right order by the function evlis. Since nexprs expect a single argument, the arguments are gathered into a list. The arguments to an fexpr are not evaluated but they are gathered into a list. The remaining function type to consider is macro. The entire function application form is passed to macros. The result of applying the macro is then evaluated.

(apply FUN:id, function ARGS:any-list): any *expr*

Applies the function FUN to the list of arguments ARGS. The following is an approximation of the real code.

```
(de apply (fun args)
  (cond ((and (idp fun) (not (funboundp fun)))
         (if (fcodep fun)
           (codeapply (getfcodepointer fun) args)
           (lambdaapply (get fun '*lambdalink) args)))
        ((codep fun) (codeapply fun args))
        ((is-lambda fun) (lambdaapply fun args))
        (t (error))))
```

Apply uses the additional functions codeapply and lambdaapply to actually evaluate the body of the function. The arguments in ARGS are not evaluated (for example, by a call on evlis). This may seem odd since the type of the function may be expr or nexpr. Apply assumes that the arguments are in the form required for binding to the formal parameters of FUN. Getcodepointer returns the code-pointer associated with an id. The body of a function which is not compiled is found on the property list of its name, under the *lambdalink indicator.

```
1 lisp> (setq fn 'add1)
ADD1
2 lisp> (de fn (a) (sub1 a))
FN
3 lisp> (apply fn '(1))
2
4 lisp> (apply 'fn '(1))
0
5 lisp> (apply 'cons '((add1 2) 3))
((add1 2) . 3)     % NOT (3 . 3)
```

(funcall FUN:id, function [ARGUMENT:any]): any *macro*
Equivalent to (apply FUN (list ARGUMENT1 ... ARGUMENTN)). This function is defined in the USEFUL module.

9.2 Support Functions for Eval and Apply

(evlis U:any-list): any-list *expr*
Evlis evaluates each element of U The list of results is returned.

```
(de evlis (p)
  (if (not (pairp p))
    ()
    (cons (eval (first u))
          (evlis (rest u)))))
```

(idapply FN:id U:any-list): any *expr*
Applies the function referenced by FN to the argument list U. An equivalent form would be (apply FN U). The use of idapply is more efficient. If FN is not an id it is an error and the message

```
***** Ill-formed function expression
```

is printed.

(codeapply FN:code-pointer U:any-list): any *expr*
The body of compiled function referenced by FN is executed with arguments in U.

(codeevalapply FN:code-pointer U:any-list): any *expr*
(codeevalapply FN U) is essentially (codeapply FN (evlis U)). The difference between the two is that the first is more efficient than the second.

(evprogn U:form-list): any *expr*
The forms in U are evaluated in a left to right order. The value of the last is returned. This function is used in situations where an application of progn is implied. For example, the definition of many functions consists of a sequence of expressions. Each expression is evaluated and the value of the last is returned, without having to wrap the definition inside a progn.

```
(de evprogn (u)
  (if (pairp u)
    (progn (while (pairp (cdr u))
             (eval (car u))
             (setq u (cdr u)))
           (eval (car u)))
    nil))
```

9.3 Special Evaluator Functions, Quote and Function

(quote U:any): any *fexpr*
Quote is used to distinguish a constant s-expression from one which is to be evaluated. The return value is U. Since quote is an fexpr U is never evaluated.

(mkquote U:any): list *expr*
Returns the result of (list 'quote U).

(function FN:function): function *fexpr*
Function is similar to quote, except that its argument is a reference to a function. FN is either the name of a function, a lambda expression, or a code-pointer. It is not correct to use quote to suppress the evaluation of an expression which represents a function. The misuse of function can effect the way in which code is compiled, in particular, application of the mapping functions (see Chapter ??? for more information).
Use of the read macro #' will result in an application of function. For example, if the reader encounters #'fun the expression (FUNCTION FUN) will be returned. This read macro is part of the `useful` package.

9.4 Support Functions for Macro Evaluation

(expand L:list FN:function): list *expr*
Expand is a convenient way to expand a certain type of macro. PSL supports functions like plus which accept any number of arguments. The macro plus is actually defined in terms of the binary operator plus2. For example,

```
(plus 1 2 3)
```

expands into

```
(plus2 1 (plus2 2 3))
```

FN should be a function which accepts two arguments, and L should be a list of values which are acceptable to FN as arguments.

```
(de expand (l fn)
  (cond ((not (pairp l)) l)
        ((not (pairp (cdr l))) (car l))
        (t (list fn (car l) (expand (cdr l) fn)))))

1 lisp> (dm plus (a) (expand (rest a) 'plus2))
plus
2 lisp> (macroexpand '(plus 1 2 3))
(plus2 1 (plus2 2 3))
```

(robustexpand L: list FN: function EMPTYCASE: form): list *expr*
If the list L is empty then EMPTYCASE is evaluated, otherwise the result of (expand L FN) is returned.

Input and Output

10.1 Introduction

One category of input and output in LISP is "symbolic" I/O. This allows a user to print or read possibly complex LISP objects with one or a few calls on standard functions. PSL also has powerful general-purpose I/O.
Input from multiple sources and output to multiple destinations can be done in PSL all at the same time. PSL provides I/O functions with explicit specification of sources and destinations for I/O. On the other hand for convenience it is often desirable to let the source or destination be implicit. PSL provides the full set of I/O operations through functions with an implicit source or destination.
The functions with and without an explicit channel designator argument are described together in this chapter. In each case calling the function with the implicit source or destination is the same as calling the version with explicit channel argument and supplying the value of the variable in* or out* as the channel.
The current input or output channel can be changed by setting or rebinding the variables **in*** or **out***. Historically, the functions rds and wrs have been used for this and they are also available along with their special features.

10.1.1 Organization of this Chapter

We first discuss the syntax used for symbolic input and output. The syntax described applies to PSL programs, interactive typein, format of data in data files, and to output by PSL programs except when special formatting is used. Functions for printing and reading follow. All (textual) input and output functions are discussed. Next is open, for setting up input and output with files, plus related functions. A great deal of user input/output programming can be done using just a subset of the functions described in these first sections.
PSL includes functions that load program modules and execute command files. They are essential to building of software systems even if the system itself does no I/O. Functions of this type are described next.

The section on I/O channels discusses some features available for switching the current output from channel to channel, and documents some fluid variables used in directing some of the system's input and output.
Functions in the next section actually operate on objects such as lists and strings! Since I/O functions scan input and format output, and since it is possible to read from or print to a string, I/O functions can be useful for building strings and for scanning them. Some built-in functions are described. The last two sections describe mechanisms that make possible some sophisticated uses of the PSL I/O system. One describes the mechanism in PSL that permits writing to a string or taking input from the text buffer of a text editor. The other discusses the tables used by the PSL scanner, which is modifiable.

10.2 Printed Representation of LISP Objects

Most of this section is devoted to the representation of tokens. In addition to tokens there are composite objects with printed representations: lists and vectors. We briefly discuss their printed formats first.

```
"(" expression expression  . . . ")"
"(" expression expression . . . "." expression)
"[" expression expression . . . "]"
```

Of these the first two are for lists. Where possible, the first notation is preferred and the printing routines use it except when the second form is needed. The second form is required when the cdr of a pair is neither nil nor another pair. The third notation is for vectors. For example:

```
(A . (B . C)) % An s-expression
(A B . C)     % Same value, list notation
(A B C)       % An ordinary list
[A B C]       % A vector
```

The following standards for representing tokens are used:

- An id normally consists of uppercase letters and digits. The first character can be a digit when the first non-digit character is a plus sign, minus sign, or a letter other than "E" or "B". We exclude lowercase letters because the PSL reader will normally convert lowercase letters to uppercase. Although the user may use lowercase letters, the interpreter only sees uppercase. This conversion is controlled by the value of the switch *raise, a value of nil will suppress this conversion. In addition to letters and numbers, the following characters are considered alphabetic for the purpose of notating symbols.

```
+ - $ & * / : ; | < = > ? ^ _ ~ @
```

There is an escape convention for notating an id whose name contains special characters. In addition to lowercase letters the following characters are considered special.

```
! " % ' ( ) . [ ] ` , # |

JULIE     % three ways to notate the same symbol
julie
JuLie
+$
1+
+1        % this is a number, not an id
x^2+y^2   % a single symbol
```

- Strings begin with a double quote (") and include all characters up to a closing double quote. A double quote can be included in a string by doubling it. An empty string, consisting of only the enclosing quote marks, is allowed. The characters of a string are not affected by the value of the *raise. Examples:

```
"This is a string"
"This is a ""string"""
""
```

- Integers begin with a digit, optionally preceded by a + or - sign, and consist only of digits. The global input radix is 10; there is no way to change this. However, numbers of different radices may be read by the following convention. A decimal number from 2 to 36 followed by a sharp sign (#), causes the digits (and possibly letters) that follow to be read in the radix of the number preceding the #. [Footnote: Octal numbers can also be written as a string of digits followed by the letter "B".] Thus 63 may be entered as 8#77, or 255 as 16#ff or 16#FF. The output radix can be changed, by setting outputbase*. If outputbase* is not 10, the printed integer appears with appropriate radix. Leading zeros are suppressed and a minus sign precedes the digits if the integer is negative. Examples:

```
100
+5234
-8#44 (equal to -36)
```

- Floats have a period and/or a letter "e" or "E" in them. Any of the following are read as floats. The value appears in the format [-]n.nn...nnE[-]mm if the magnitude of the number is too large or small to display in [-]nnnn.nnnn format. The crossover point is determined by the implementation. In BNF, floats are recognized by the grammar:

```
<base>        ::= <unsigned-integer>.|
                  .<unsigned-integer>|
                  <unsigned-integer>.
                  <unsigned-integer>
<ebase>       ::= <base>|<unsigned-integer>
<unsigned-float> ::= <base>|
                     <ebase>e<unsigned-integer>|
                     <ebase>e-<unsigned-integer>|
                     <ebase>e+<unsigned-integer>|
                     <ebase>E<unsigned-integer>|
                     <ebase>E-<unsigned-integer>|
                     <ebase>E+<unsigned-integer>
<float>           ::= <unsigned-float>|
                     +<unsigned-float>|
                     -<unsigned-float>
```

  That is:

```
[+|-][nnn][.]nnn{e|E}[+|-]nnn

nnn.
.nnn
nnn.nnn
```

  Examples:

```
1e6
.2
2.
2.0
-1.25E-9
```

- Code-pointers cannot be read directly, but can be printed and constructed. Currently printed as

```
#<Code argument-count hexadecimal-address>.
```

- Anything else is printed as #<Unknown:nnnn>, where nnnn is the hexadecimal value found in the argument register. Such items are not legal LISP entities and may cause garbage collector errors if they are found in the heap. They cannot be read in.

10.3 Functions for Printing

10.3.1 Basic Printing

(prin1 ITM:any): ITM:any *expr*

(channelprin1 CHAN:io-channel ITM:any): ITM:any *expr*
Channelprin1 is the basic printing function. For well-formed, non-circular structures, the result can be parsed by the function read.

(prin2 ITM:any): ITM:any *expr*

(channelprin2 CHAN:io-channel ITM:any): ITM:any *expr*
Channelprin2 is similar to channelprin1, except that strings are printed without the surrounding double quotes, and delimiters within ids are not preceded by the escape character.

The following example illustrates the difference between prin1 and prin2.

```
ARGUMENT     PRIN1        PRIN2

A!%WORD      A!%WORD      A%WORD
"STRING"     "STRING"     STRING
```

(print U:any): U:any *expr*

(channelprint CHAN:io-channel U:any): U:any *expr*
Display U using channelprin1 and then terminate the line using channelterpri.

10.3.2 Whitespace Printing Functions

(terpri): nil *expr*

(channelterpri CHAN:io-channel): nil *expr*
Write an end of line character. The number of characters output on the current line (that which is referenced by channelposn), is defined to be zero and the number of lines output on the current page (that which is referenced by channelposn), is incremented.

(spaces N:integer): nil *expr*

(channelspaces CHAN:io-channel N:integer): nil *expr*
N spaces are written.

(tab N:integer): nil *expr*

(channeltab CHAN:io-channel N:integer): nil *expr*
Move to column N

```
(channelspaces ch (- N (channelposn ch)))
```

If the position on the current output line is past column N then channelterpri is called before moving to column N.

10.3.3 Formatted Printing

(printf FORMAT:string [ARGS:any]): nil *expr*

(channelprintf CHAN:io-channel FORMAT:string [ARGS:any]):nil
expr

Channelprintf is a simple routine for formatted printing. The FORMAT is a string. The characters of this string are printed unless a a interpret and print the other arguments to channelprintf. The following format characters are currently supported.

- %b The next argument is assumed to be an integer, it is passed to spaces.
- %c The next argument should be a single character, it is printed by a call on writechar.
- %d Print the next argument as a decimal integer.
- %e The next argument is evaluated by a call on eval.
- %f Print an end-of-line character if not at the beginning of the output line (does not use a matching argument).
- %l Print the next argument using print2l, this is the same as %w except that lists are printed without the top level pair of parenthesis. The empty list is printed as a blank.
- %n Print end-of-line character (does not use a matching argument).
- %o Print the next argument as an octal integer.
- %p Print the next argument using prin1.
- %p Print the next argument using prin1.
- %r Print the next argument using errprin, the result is the same as %p except that a surrounding pair of quotes are also printed.
- %s The next argument is assumed to be a string, the surrounding double quotes are not printed.
- %t The next argument is assumed to be an integer, it is passed to tab.
- %w Print the next argument using prin2.
- %x Print the next argument as a hexadecimal integer.

If the character following % is not either one of the above or another %, it causes an error. Thus, to include a % in the format to be printed, use %.

There is no checking for correspondence between the number of arguments that FORMAT expects and the number given. If the number given is less than the number in the FORMAT string, garbage will be inserted for the missing arguments. If the number given is greater than the number in the FORMAT string, then the extra ones are ignored.

(prettyprint U:form): U:form *expr*

Prettyprints U.

10.3.4 The Fundamental Printing Function

(writechar CH:character): character *expr*

(channelwritechar CHANNEL:io-channel CH:character): character *expr*
Write one character to the device specified by CHANNEL. All output is defined in terms of this function. The number of characters output on the current line (that which is referenced by channelposn), and the number of lines output on the current page (that which is referenced by channellposn), are updated. Each channel specifies an output function, it is this function that is applied to CHANNEL and CH to actually write the character.

10.3.5 Additional Printing Functions

(prin2l L:any): L:any, nil *expr*
Prin2, except that a list is printed without the top level parentheses. If L is a pair then the return value is nil, otherwise the return value will be L.

(prin2t X:any): any *expr*

(channelprin2t CHAN:io-channel X:any): any *expr*
Output X using channelprin2 and terminate line with channelterpri.

(princ ITM:any): ITM:any *expr*

(channelprinc CHAN:io-channel ITM:any): ITM:any *expr*
Same function as channelprin2.

(errprin U:any): None Returned *expr*
Prin1 with special quotes to highlight U.

(errorprintf FORMAT:string [ARGS:any]): nil *expr*
Errorprintf is similar to printf, except that errout* is used in place of the currently selected output channel. Channelterpri is called before and after printing if the line position is greater than zero.

(eject): nil *expr*

(channeleject CHAN:io-channel): nil *expr*
Skip to top of next output page.

10.3.6 Printing Status and Mode

For information on directing various kinds of output see the section on channels.

outputbase* = [Initially: 10] *global*
This fluid can be set to control the radix in which integers are printed out. If the radix is not 10, the radix is given before a sharp sign.

```
1 lisp> 16
16
2 lisp> (setq outputbase* 8)
8#10
3 lisp> 16
8#20
```

(posn): integer *expr*

(channelposn CHAN:io-channel): integer *expr*
Returns number of characters output on the current line of this channel.

(lposn): integer *expr*

(channellposn CHAN:io-channel): integer *expr*
Returns number of lines output on this page of this channel.

(linelength LEN:{integer, nil}): integer *expr*

(channellinelength CHAN:io-channel LEN: {integer, nil}): integer *expr*
For each channel there is a restriction on the length of output lines. If LEN is nil then the value returned will be the current maximum length. If LEN is an integer greater than zero then it will become the new maximum. However, if the argument is zero there will be no restrictions on the size of the output lines. The following example illustrates how this length is used. PSL uses a similar function to apply output functions like prin1.

```
(de check-line-length (length ch fn token)
  (when (and (> (+ (channelposn ch) length)
                (channellinelength ch nil))
             (> (channellinelength ch nil) 0))
    (channelwritechar ch (char eol)))
  (idapply fn (list ch token)))
```

The fluid variables PrinLevel and PrinLength allow the user to control how deep the printer will print and how many elements at a given level the printer will print. This is useful for objects which are very large or deep. These variables affect the functions prin1, prin2, princ, print, and printf (and the corresponding Channel functions).

prinlevel = [Initially: nil] *global*
Controls how many levels deep a nested data object will print. If PrinLevel is nil, then no control is exercised. Otherwise the value should be an integer. An object to be printed is at level 0.

prinlength = [Initially: nil] *global*
Controls how many elements at a given level are printed. A value of nil indicates that there be no limit to the number of components printed. Otherwise the value of PrinLength should be an integer.

10.4 Functions for Reading

10.4.1 Reading S-Expressions

(read): any *expr*

(channelread CHAN:io-channel): any *expr*
Reads and returns the next S-expression from input channel CHAN. Valid input forms are: vector-notation, dot-notation, list-notation, numbers, strings, and identifiers. Identifiers are interned (see the intern function in Chapter 4), unless the fluid variable *compressing is non-nil. Channelread returns the value of the global variable eof when the end of the currently selected input channel is reached.

```
(de channelread (ch)
  (let ((currentscantable* lispscantable*)
        (currentreadmacroindicator* 'lispreadmacro))
    (channelreadtokenwithhooks ch)))
```

Channelread uses the function channelreadtokenwithhooks. Tokens are scanned within the context of the scan table bound to lispscantable*. The scan table is used to recognize special character types, for example, delimiters like (,), and space. The read macro mechanism is used to parse s-expressions (see section 12.4.5 for more information on read macros). PSL uses a number of read macros which are described below.

(	S-expressions are gathered into a list until the matching right parenthesis is read. Both list and dot notation are recognized. A pair or list is returned.
)	If a right parenthesis is read and a matching left parenthesis had not been previously read then it will be ignored.
[	S-expressions are gathered into a vector until a matching right bracket is read. A vector is returned.
'	The next s-expression read is quoted, the result is of the form (QUOTE S-EXPRESSION).
(char eof)	If an end of file character is read while a list or vector is being constructed then the following error will occur. ***** Unexpected EOF while reading on channel.
'N'	Otherwise the value of *eof* is returned.

The following read macros are defined in the `useful` module.

#´	this is like the quote mark ' but it is used for function instead of quote. For example #'name reads as (FUNCTION NAME).
#/	this returns the numeric form of the following character read without raising it. For example #/a is 97 while #/A is 65.
#\	This is a read macro for char (see Chapter 6 for more information). Note that the argument will be raised if *raise is non-nil. For example, #\a = #\A = 65.
#.	This causes the following expression to be evaluated at read time. For example, '(1 2 #.(plus 1 2) 4) reads as (1 2 3 4).
#+	The next two expressions are read. If the first is a system name which is the current system then the second expression is returned. Otherwise another expression is read and it becomes the value returned.
#-	The next two expressions are read. If the first is a system name which is not the current system then the second expression is returned. Otherwise another expression is read and it becomes the value returned.

10.4.2 Reading Single Characters

(readchar): character *expr*

(channelreadchar CHANNEL:io-channel): character *expr*

Reads one character (an integer) from CHANNEL. All input is defined in terms of this function. If CHANNEL is not open or is open for writing only, an error is generated. If there is a non-zero value in the backup buffer associated with CHANNEL, the buffer is emptied (set to zero) and the value returned. Otherwise, the reading function associated with CHANNEL is called with CHANNEL as argument, and the value it returns is returned by channelreadchar.

```
***** Channel not open

***** Channel open for write only
```

(readch): id *expr*

(channelreadch CHAN:io-channel): id *expr*
Like channelreadchar, but returns the id for the character rather than its ASCII code.

(unreadchar CH:character): Undefined *expr*

(channelunreadchar CHAN:io-channel CH:character): Undefined *expr*
The input backup function. Ch is deposited in the backup buffer associated with Chan. This function should be only called after channelreadchar is called, and before any intervening input operations, since it is used by the token scanner. The unread buffer only holds one character, so it is generally useless to unread more than one character.

10.4.3 Reading Tokens

The functions described here pertain to the token scanner and reader. Globals and switches used by these functions are defined at the end of this section.

(channelreadtoken CHANNEL:io-channel): {id, number, string} *expr*
This is the basic PSL token scanner. The value returned is an item corresponding to the next token from the input stream. An id will be interned unless the switch ***compressing** is non-nil. If *compressing is t then the print name of the id is passed to newid, otherwise it is passed to intern.

The global variable toktype* is set as follows.

0 if the token is an ordinary id,
1 if the token is a string,
2 if the token is a number, or
3 if the token is an unescaped delimiter such as "(", but not "!(" In this last case, the value returned is the id whose print name is the same as the delimiter.

The precise behavior of this function depends on two fluid variables:
Currentscantable* is bound to a vector known as a scan table. Described below.

Currentreadmacroindicator* is bound to an id known as a read macro indicator. Described below.

(ratom): {id, number, string} *expr*
Reads a token from the current input channel.

(channelreadtokenwithhooks CHANNEL:io-channel): any *expr*
This function reads a token and performs any action specified if the token is marked as a read macro under the current indicator. Read uses this function internally. Uses the variable currentreadmacroindicator* urrentreadmacroindicator* to determine the current indicator.

10.4.4 Reading Entire Lines

(readline): string *expr*
Same as (channelreadline in*)
(channelreadline CHANNEL:io-channel): string *expr*
A string is returned which contains each character from the current position of the scanner to the next end-of-line or end-of-file character.

10.4.5 Read Macros

At the top level of PSL, an expression is read, it is evaluated, and then the result is printed. Normally a macro is expanded during evaluation. The read macro is a different type of macro, it is expanded during the reading phase. When the reader encounters a read macro character, the the macro is executed and the result is inserted in place of the read macro character.
A read macro must be a function of two arguments, the first should represent a IO channel, the second a character. A character which represents a read macro must be set to one of two types in the scan table. It may be either a delimiter or a diphthong. Diphthong corresponds to double character read macros, delimiter to single character read macros. In addition, the id which corresponds to the character must have a reference to the name of the function on its property list. For a diphthong the indicator must be lispdiphthong, for a delimiter it must be lispreadmacro.

The quote macro may be defined as follows. Note that we cannot use 'form in place of (quote form) until we have defined the read macro.

```
(de doquote (channel ch)
  (list (quote quote) (channelread channel)))

(put (quote !') (quote lispreadmacro)(function doquote))

(putv lispscantable* (char !') delimiter)
```

This says that when a single quote is read, PSL should replace it with a list consisting of quote and the next expression in the input (obtained by an explicit call to channelread). Since defining a character as a read macro makes it difficult to use the character in a normal way, read macros should not be letters or digits.

10.4.6 Terminal Interaction

(yesp MESSAGE:string): boolean *expr*
If the user responds y or yes, yesp returns a non-nil value. A response of n or no results in a value of nil. It is possible to enter a break loop by responding with b. After quitting the break loop, one must still respond y, yes, n, or no.

10.4.7 Input Status and Mode

promptstring* = [Initially: "x lisp>"] *global*
Displayed as a prompt when any input is taken from TTY. Prompts should therefore not be directly printed. Instead the value should be bound to promptstring*.

***eolinstringok** = [Initially: nil] *switch*
If *eolinstringok is non-nil, the warning message
`*** String continued over end-of-line`
is suppressed.

***raise** = [Initially: t] *switch*
If *raise is non-nil, all characters input for ids through PSL input functions are raised to upper case. If *raise is nil, characters are input as is. A string is unaffected by *raise.

***compressing** = [Initially: nil] *switch*
If *compressing is non-nil, channelreadtoken and other functions that call it do not intern ids.

currentscantable* = [Initially: NIL] *global*
This variable is set to lispscantable* by the function read

currentreadmacroindicator* = [Initially: NIL] *global*
The function read binds this variable to the value LISPREADMACRO. Its value determines the property list indicator used in looking up read macros. The user may define a set of read macros using some new indicator and rebind this variable. Ordinary read macros may be added by putting properties on ids under the LISPREADMACRO indicator.

10.5 File System Interface: Open and Close

(open FILENAME:string ACCESSTYPE:id): CHANNEL:io-channel *expr*
If AccessType is eq to input or output, an attempt is made to access the system-dependent FILENAME for reading or writing. If the attempt is unsuccessful, an error is generated; otherwise a free channel is returned and initialized to the default conditions for ordinary file input or output.

If none of these conditions hold, a file is not available, or there are no free channels, an error is generated.

```
***** Unknown access type
***** Improperly set-up special IO open call
***** File not found
***** No free channels
```

If AccessType is eq to SPECIAL, no file is opened. Instead the channel is initialized as a generalized input and/or output stream. See below.

(filep NAME:string): boolean *expr*
This function will return t if file NAME can be opened, and nil if not, e.g. if it does not exist.

(close CHANNEL:io-channel): io-channel *expr*
The closing function associated with CHANNEL is called, with CHANNEL as its argument. If it is illegal to close CHANNEL, if CHANNEL is not open, or if CHANNEL is associated with a file and the file cannot be closed by the operating system, this function generates an error. Otherwise, CHANNEL is marked as free and is returned.

Here is a simple example of input from a particular file with output sent to the current output channel. This function reads forms from the file MYFILE.DAT and prints out all those whose car is eq to its parameter. Using unwind-protect, we are assured that the channel (and the file), will be closed in all cases, including errors.

```
(de filter-my-file (x)
  (let ((chan (open "MYFILE.DAT" 'input))
        form)
    (unwind-protect
      (while (neq (setq form (channelread chan))
                  $eof$)
             (if (and (pairp form) (eq (car form) x))
               (print form)))
      (close chan))))
```

10.6 Loading Modules

Two convenient procedures are available for loading modules. Various facilities described in this manual are actually in loadable modules and their documentation notes that they must be loaded. Loadable modules typically exist as FASL files .B files see the section on the compiler for information on producing FASL files.

(load [FILE:{string, id}]): nil *macro*
For each argument FILE, an attempt is made to locate a corresponding file. If a file is found then it will be loaded by a call on an appropriate function. A full file name is constructed by using the directory specifications in loaddirectories* and the extensions in loadextensions*. The strings from each list are used in a left to right order, for a given string from loaddirectories* each extension from loadextensions* is used. More information about loaddirectories* and loadextensions* can be found below.
While the file is being loaded *usermode will be set to nil. If a file cannot be found the call on load will be aborted.

```
***** 'FILE' load module not found
```

If either *verboseload or *printloadnames is non-nil then

```
*** loading FULL-NAME
```

is printed just prior to loading the file. Once a file has been loaded the message

```
*** FILE loaded
```

will be printed if the *verboseload is non-nil. In addition, the name FILE is added to the list referenced by options*. If an attempt is made to load a file which has been partially loaded then

```
*** Warning: Load of FILE previously requested, but incomplete.
```

will be printed. If FILE is found to be in options* then the attempt to load FILE will be ignored.

```
*** FILE already loaded
```

Note that memq is used to determine if FILE is in options*. Therefore when you use string arguments for loading files, although identical names for ids refer to the same object, identical names for strings refer to different objects.

(reload [FILE:{string,id}]): nil *macro*
Reload is very similar to load. The difference between the two is that for each name FILE, the first occurance of FILE in options* will be removed before attempting to load the file.

(imports FILES:list): nil *expr*
This function is also used to load modules. If imports is invoked as another module is being loaded, then the modules specified by FILES will not be loaded until the loading of the current module is complete. Otherwise imports is identical to load.

loaddirectories* = [Initially: list] *global*
References a list of strings to append to the front of file names passed as arguments to load, reload, and imports.

loadextensions* = [Initially: association-list] *global*
References the a-list `((".b" . faslin) (".lap" . lapin))`
The car of each pair is a string which represents an extension to append to the end of the file names passed as arguments to load, reload, and imports. The cdr is a function appropriate for loading a file of a particular extension. For example, a file whose extension is B is loaded with the function faslin.

options* = [Initially: nil] *global*
Once a file corresponding to an argument FILE to either load, reload, or imports has been loaded, FILE will be added to the list referenced by options*. An attempt to load a file by applying load or imports will be aborted if FILE is found in options*. Reload removes the first occurance of FILE from options* before passing its argument to load.

***verboseload** = [Initially: nil] *switch*
If non-nil, a message is displayed when a request is made to load a file which has already been loaded, when a file is about to be loaded, and when the loading of a file is complete. Since *redefmsg is set to the value of *verboseload, a non-nil value will also cause a message to be printed whenever a function is redefined during a load.

***printloadnames** = [Initially: nil] *switch*
If non-nil, a message is printed when a file is about to be loaded.

10.7 Reading Files into PSL

The following procedures are used to read complete files into PSL, by first calling open, and then processing each top level form in the file. The effect is similar to what would happen if the file were typed into PSL. File names are represented by strings. File names may be given using full system dependent file name conventions.

***echo** = [Initially: nil] *switch*
The switch echo is used to control the echoing of input. When (on echo) is placed in an input file, the contents of the file are echoed on the standard output device. Dskin does not change the value of *echo, so one may say (on echo) before calling dskin, and the input will be echoed.

(dskin NAME:string): nil, abort *expr*

The contents of the file are processed as if they were typed in. If the processing of the file is aborted the return value will be the identifier abort and the following error message will be printed.

```
***** DSKIN of 'NAME' aborted after N form(s)
```

A count of each top level form is kept as they are processed, which is the value N in the error message. Once in* has been bound to the channel which represents the open file, each form is processed by a function similar to the one below.

```
(de dskin-step ()
  (let ((form (funcall toploopread*)))
    (cond ((eq form $eof$) 'eof)
          ((not *defn)
           (funcall toploopprint* (funcall toploopeval* form))
          (dfprint* (funcall dfprint* form))
          (t (prettyprint form)))))
```

Note that the functions used for reading, evaluating and printing are the same as those used in toploop (see Chapter 15 for more information). If dfprint* has a value (and the switch *defn is non-nil), then it will be applied to the expression instead of applying the functions bound to toploopeval* and toploopprint*. If dfprint* does not have a value (and the switch *defn is nil), then the expression will be simply be printed by a call on prettyprint.

Dskin is flagged ignore. This means that when a file is compiled, a top level application of dskin will be evaluated but not compiled. For more information about the flag ignore see Chapter 19.

(lapin NAME:string): {nil, abort} *expr*

Almost identical to dskin. The difference is that the function bound to toploopprint* is not applied. In general, this means that the results of evaluation are not printed.

Note that lapin is not flagged ignore as is dskin. This means that a top level application of lapin is compiled but not evaluated. For more information on how the flags ignore and eval effect compilation see Chapter 19.

(faslin FILENAME:string): nil *expr*
This is an efficient binary read loop, which fetches blocks of code, constants and compactly stored ids. It uses a bit-table to relocate code and to identify special LISP-oriented constructs. FILENAME must be a complete file name.

10.8 About I/O Channels

(rds CHANNEL:io-channel, nil): io-channel *expr*
Rds sets in* to the value of its argument, and returns the previous value of in*. In addition, if specialrdsaction* is non-nil, it should be a function of 2 arguments, which is called with the old channel as its first argument and the new channel as its second argument. (rds nil) does the same as (rds stdin*).

(wrs CHANNEL:io-channel, nil): io-channel *expr*
Wrs sets out* to the value of its argument and returns the previous value of out*. In addition, if specialwrsaction* is non-nil, it should be a function of 2 arguments, which is called with the old channel as its first argument and the new channel as its second argument. (wrs nil) does the same as (wrs stdout*).
Global variables containing information about channels are listed below.

in* = [Initially: 0] *global*
Contains the currently selected input channel. May be set or rebound by the user. This is changed by the function rds.

out* = [Initially: 1] *global*
Contains the currently selected output channel. May be set or rebound by the user. This is changed by the function wrs.

stdin* = [Initially: 0] *global*
The standard input channel (but not in the Unix sense of standard input). Channel 0 is ordinarily the terminal and this variable is not intended to be set or rebound.

stdout* = [Initially: 1] *global*
The standard output channel. Like channel 0, channel 1 is ordinarily always the terminal, and this variable is not intended to be set or rebound.

breakin* = [Initially: nil] *global*
The channel from which the break loop gets its input. It has been set to default to stdin*, but may have to be changed on some systems with buffered-IO.

breakout* = [Initially: nil] *global*
The channel to which the break loop sends its output. It has been set to default to stdout*, but may have to be changed on some systems with buffered-IO.

errout* = [Initially: 1] *global*
The channel used by the errorprintf.

specialrdsaction* = [Initially: nil] *global*

specialwrsaction* = [Initially: nil] *global*

10.9 I/O to and from Lists and Strings

(bldmsg FORMAT:string, [ARGS:any]): string *expr*
Printf to string. This can be used as a very convenient way of obtaining the printed representation of an object for further analysis. In many cases it is also a very convenient way of constructing a needed string. An error will occur on overflow. Overflow occurs when the length of the result exceeds the value of maxtokensize.

```
***** Buffer overflow while constructing error message: FORMAT
```

(flatsize U:any): integer *expr*
Character length of prin1 S-expression.

(flatsize2 U:any): integer *expr*
Prin2 version of flatsize.

(explode U:any): id-list *expr*
Explode takes the constituent characters of an S-expression and forms a list of single character ids. It is implemented via the function channel-prin1, with a list rather than a file or terminal as destination. Returned is a list of interned characters representing the characters required to print the value of U.

```
1 lisp> (explode 'foo)
(f O O)
2 lisp> (explode '(a . b))
(!( a !  !. !  b !))
```

(explode2 U:atom-vector): id-list *expr*
Prin2 version of explode.

(compress U:id-list): atom-vector *expr*
U is a list of single character identifiers which is built into a PSL entity and returned. Recognized are numbers, strings, and identifiers with the escape character prefixing special characters. The formats of these items appear in the "Primitive Data Types" Section, Section 2.1.2. Identifiers are not interned on the id-hash-table. Function pointers may not be compressed. If an entity cannot be parsed out of U or characters are left over after parsing an error occurs:

```
***** Poorly formed atom in COMPRESS
```

(implode U:id-list): atom *expr*
Compress with ids interned.

10.10 Generalized Input/Output Streams

All input and output functions are implemented in terms of operations on "channels". A channel is just a small integer [1] which has 3 functions and

[1] The range of channel numbers is from 0 to MaxChannels, where MaxChannels is a system-dependent constant, currently 31, defined in the module **io-decls**

some other information associated with it. The three functions are:

1. A reading function, which is called with the channel as its argument and returns the integer ASCII value of the next character of the input stream. If the channel is for writing only, this function is writeonlychannel. If the channel has not been opened, this function is channelnotopen. The reading function is responsible for echoing characters if the flag *echo is non-nil. It should use the function writechar to echo the character. It may not be appropriate for a read function to echo characters. For example, the "disk" reading function does echoing, while the reader used to implement the compress function does not.

   The read function must also be concerned with the handling of ends of "files" (actually, ends of channels) and ends of lines. It should return the ASCII code for an end of file character (system dependent) when reaching the end of a channel. It should return the ASCII code for a line feed character to indicate an end of line (or "newline"). This may require that the ASCII code for carriage return be ignored when read, not returned.

2. A writing function, which is called with the channel as its first argument and the integer ASCII value of the character to write as its second argument. If the channel is for reading only, this function is readonlychannel. If the channel has not been opened, this function is channelnotopen.

3. A closing function, which is called with the channel as its argument and performs any action necessary for the graceful termination of input and/or output operations to that channel. If the channel is not open, this function is channelnotopen.

The other information associated with a channel includes the current position in the output line (used by posn), the maximum line length allowed (used by linelength and the printing functions), the single character input backup buffer (used by the token scanner), and other system-dependent information. Ordinarily, the user need not be aware of the existence of this mechanism. However, because of its generality, it is possible to implement operations other than just reading from and writing to files using it. In particular, the LISP functions explode and compress are performed by writing to a list and reading from a list, respectively (on channels 3 and 4 respectively).

10.10.1 Using the "Special" Form of Open

If Open is called with AccessType eq to SPECIAL and the global variables specialreadfunction*, specialwritefunction*, and specialclosefunction*

are bound to ids, then a free channel is returned and its associated functions are set to the values of these variables. Other non system-dependent status is set to default conditions, which can later be overridden. The functions readonlychannel and writeonlychannel are available as error handlers. The parameter Filename is used only if an error occurs.

The following globals are used by the functions in this section.

specialclosefunction* = [Initially: nil] *global*

specialreadfunction* = [Initially: nil] *global*

specialwritefunction* = [Initially: nil] *global*

10.11 Scan Table Internals

The scan table controls the behaviour of the reader. It can be modified to extend the syntax of PSL or to aid the writing of other parsers. This table contains information about the syntax of each character. It is possible to have several tables describing different syntaxes and to switch from one to another by binding the variable lispscantable*.

The table is a vector of 129 entries, indexed by 0 through 128. The first 128 entries correspond to ASCII character code. Each entry contains an integer between 0 and 21. The meaning of these numbers is given below.

0 . . . 9	DIGIT: indicates the character is a digit and gives the corresponding numeric value.
10	LETTER: indicates the character is alphabetic.
11	DELIMITER: indicates the character is a delimiter, the first character of a diphthong should not be classified as a delimiter.
12	COMMENT: indicates the character begins a comment, terminated by an end of line.
13	DIPHTHONG: indicates the character is a delimiter which may be the starting character of a diphthong. A diphthong is a two character sequence read as one token.
14	IDESCAPE: indicates that the character is an escape character, this character is used within id names to reference a character which is not a digit or alphabetic.
15	STRINGQUOTE: indicates that the character is used to delimit strings.
16	PACKAGE: indicates that the character is used to introduce explicit package names.
17	IGNORE: indicates that the character is to be ignored.
18	MINUS: indicates that the character represents a minus sign.
19	PLUS: indicates that the character represents a plus sign.
20	DECIMAL: indicates that the character represents a decimal point.
21	IDSURROUND: indicates that the character is to act for identifiers as a string quote for strings.

It may be tedious to insert an idescape character (!), before every delimiter character in the name of an id. By setting the type of a character (for example —), to idsurround this can be avoided. Every character between the vertical bars is taken as part of the ids name, as if ! were written before each. Note that !! corresponds to ! and !— to —.

```
(PUTV LISPSCANTABLE* (CHAR '|) IDSURROUND)

|"|     % the same as !"
|dave|  % the name of the id is dave not DAVE
|!|!!|  % the name is |!, the same effect is gotten
        % by writing !|!!
```

lispscantable* = [Initially: as described below] *global*

Lispscantable* associates a type with each character. It is a vector of 129 entries. The type for a character C is stored at index (CHAR C). The table below lists each character under its type. The 128'th entry is the diphthong indicator, LISPDIPHTHONG.

```
DIGIT        0 through 9
```

```
LETTER       a through z, A through Z, #, $, &, *, /,
             @, ;, :, <, >, =, ^, _, {, }, |, ~, ?,
             ^A through ^H, ^K, ^N through ^Y, ^\,^],
             ^^, ^_, rubout
DELIMITER    (, ), ', `, ^Z, [, ]
COMMENT      %
DIPHTHONG    ,
IDESCAPE     !
STRINGQUOTE  "
PACKAGE      \
IGNORE       null, tab, line-feed, ^L,
             carriage-return and space
MINUS        -
PLUS         +
DECIMAL      .
```

10.12 Scan Table Utility Functions

The following functions are provided to manage scan tables, in the **read-utils** module.

(printscantable TABLE:vector): nil *expr*
Prints the entire scantable, gives the 0 ... 127 entries with the name of the character class. Also prints the indicator used for diphthongs.

(copyscantable OLDTABLE:{vector, nil}): vector *expr*
Copies the existing scantable (or currentscantable* if given nil). The 128'th entry (the diphthong indicator), of the copy is set to the result of a call on gensym.

(putdiphthong TABLE:vector D1:id ID2:id DIP:id): nil *expr*
Installs DIP as the name of the diphthong Id1 followed by ID2 in the given scan table.

10.13 Binary I/O Functions

Binary input and output are necessary to provide an efficient access to large files e.g. for loading or generating binary files (e.g. faslin or faslout) or for

communication with other processors e.g. graphics or database processors. Another important application is the connection between two or more PSL processors on incompatible platforms. In the following descriptions HANDLE means something system specific which in many cases should better not be interpreted by the LISP evaluator. One can very well think of HANDLE as a FILE* object belonging to the world of C's I/O library. The results of a binary read function is usually not meant for LISP evaluation, either.

(binaryopenread FILENAME:string): HANDLE *expr*
opens the file FILENAME for reading. If the file cannot be opened, an error message `Couldn't open binary file for input` will be shown, an error condition is raised.

(binaryopenwrite FILENAME:string): HANDLE *expr*
opens the file FILENAME for writing. If the file cannot be opened, an error message `Couldn't open binary file for output` will be shown, an error condition is raised.

(binaryopenappend FILENAME:string): HANDLE *expr*
opens the file FILENAME in append mode. If the file cannot be opened, an error message `Couldn't open binary file for append` will be shown, an error condition is raised.

(binaryopenupdate FILENAME:string): HANDLE *expr*
opens the file FILENAME for update. If the file cannot be opened, an error message `Couldn't open binary file for update` will be shown, an error condition is raised.

(binaryclose Filepointer:handle):NIL *expr*
Closes the file referenced by filepointer

(binaryread CHANNEL:handle): any *expr*
reads one LISP item from file channel and returns it as value. There is no way to distinguish error conditions from normal return values.

(binaryreadblock CHANNEL:handle ADDR:int SIZE:int): int *expr*
reads SIZE LISP items from the file into an array pointed to by ADDR. The number of actually read items is returned, indicating error or end-of-file if the returned value is less than the number of items requested.

(binarywrite CHANNEL:handle WORD:any): NIL *expr*
writes one LISP item in WORD to file channel.

(binarywriteblock Filepointer:handle ADDR:int SIZE:int): NIL *expr*
writes SIZE LISP items from the an array pointed to by ADDR to the file referenced by Filepointer. The number of actually written items is returned, indicating error if the returned value is less than the number of items requested.

(binarypositionfile CHANNEL:handle, SPECIAL:int): any *expr*
this corresponds to C's fseek operation. The special value (the address) as well as the return value is system dependent, please investigate.

Top Level Loop

11.1 Introduction

Normally one interacts with PSL through a toplevel read-eval-print loop, it is the highest level of control and consists of an endless loop that reads an expression, evaluates it, and prints the result. One has an effect on the state of PSL by invoking actions that have side effects. This toplevel loop will catch all throws (this includes errors).

11.2 The General Purpose Top Loop Function

The user interacts with PSL through toploop. The purpose of this loop is to read expressions typed at the terminal, evaluate them, and type the result back on the terminal. Such expressions may be entered to see what value they return or to produce a side-effect on the global environment.
An expression which has a side effect on the global environment

```
(setq base 10)
```

Another example of a side effect, the definition of a function

```
(de square (n) (* n n))
```

Evaluation, just to see what value the expression returns

```
(square 2)
```

A close approximation to the behavior of this loop is given by the following functional description:

```
(de toploop (toploopread*
             toploopeval*
             toploopprint*)
  (while t
```

```
(funcall toploopprint*
         (funcall toploppeval*
                  (funcall toploopread*)))))
```

The actual toploop of PSL is somewhat more complex than the simple one defined here because it provides a few extra features (these are described below). The important point is that the primitives required for writing such a loop are all available to the user.

Syntactically correct expressions have a well defined mapping into PSL data structures, read accomplishes this mapping. For example, linear list notation is converted into the internal tree structure representation, strings are stored in a more efficient non-list form, and numbers are internalized to a form compatible with the arithmetic unit of the machine. The most primitive piece of read is the scanner. This routine recognizes characters special to PSL. For example, space, (, and). The scanner is also responsible for building the internal representation of identifiers. The scanner must make every reference to a particular id point to the same internal structure. This is accomplished at the time an id is created. The character sequence is compared against sequences which have already been converted into ids. This comparison employs an efficient search technique so that not every id is compared (currently a hash algorithm is used).

The eval function is responsible for the evaluation of data structures interpreted as programs. A through discussion of this can be found in Chapter 11.

Print displays data structures in a way which could later be typed back in to read. Some of the more interesting print routines do prettyprinting. That is, they format the output using conventions based on the structural nesting of the expressions.

Giving the user the power to invoke all of these operation from his code is a very powerful feature of LISP which sets it apart from most other languages.

(toploop TOPLOOPREAD*:function TOPLOOPPRINT*:function TOPLOOPEVAL*:function TOPLOOPNAME*:string WELCOMEBANNER:string): nil *expr*

This function is called to establish a new Top Loop. It prints the WELCOMEBANNER and then invokes a Read-Eval-Print loop, using the functions defined by TOPLOOPREAD*, TOPLOOPEVAL*, and TOPLOOPPRINT*. Since each of the parameters to toploop is fluid they may be examined or changed. Timing and history mechanisms are provided. A prompt is constructed by prefixing TOPLOOPNAME* with the history count. As a convention, the name is followed by a number of right angle brackets (>), to indicate the depth of toploop invocations.

toploopread* = [Initially: nil] *global*
When toploop is called this id is bound to the function used for reading.

toploopeval* = [Initially: nil] *global*
When toploop is called this id is bound to the function which evaluates input.

toploopname* = [Initially: string] *global*
Bound to a string (currently "lisp"), which will be part of the prompt for input.

toplooplevel* = [Initially: 0] *global*
Depth of top loop invocations.

initforms* = [Initially: nil] *global*
A list of forms to be evaluated at startup, (prior to calling main). The forms are evaluated in a left to right order, once the last form is evaluated initforms* is set to nil.

***output** = [Initially: T] *switch*
If non-nil, the result of evaluating top level forms is printed.

***time** = [Initially: nil] *switch*
If non-nil, a step evaluation time is printed after each top level form is processed within toploop.

(hist [N:integer]): nil *nexpr*
Hist is called with 0, 1 or 2 integers, which control how much history is to be printed out:

(hist)	Display full history.
(hist n m)	Display history from n to m.
(hist n)	Display history from n to present.
(hist -n)	Display last n entries.

The following functions permit the user to access and resubmit previous expressions, and to re-examine previous results.

(inp N:integer): any *expr*
Return N'th input at this level.

(redo N:integer): any *expr*
Reevaluate N'th input.

(ans N:integer): any *expr*
Return N'th result.

historycount* = [Initially: 0] *global*
The number of entries which have been read so far.

historylist* = [Initially: nil] *global*
A list of pairs, the first element of each pair represents an input form, the second is the result of evaluating the frst. Note that the top level evaluation of historylist* will result in a circular list.

11.3 Changing the Default Top Level Function

As PSL starts up, it first sets the stack pointer and various other variables, and then calls the function main inside a catch form.

```
(catch 'reset (main))
```

By default, main calls standardlisp.

```
(de main ()
  (funcall 'standardlisp))

(de standardlisp ()
  (let ((currentreadmacroindicator* 'lispreadmacro)
        (currentscantable* lispscantable*))
    (toploop 'read 'print 'eval "lisp" "Portable Standard LISP")))
```

In order to have a saved PSL come up in a different top loop, the function Main should be appropriately redefined by the user.

(main): Undefined *expr*

An initialization function which is called after the stack is set. Its redefinition allows the user to change the default top loop.

Error Handling

12.1 Introduction

In PSL, as in most LISP systems, various kinds of errors are detected by functions in the process of checking the validity of their argument types and other conditions. Errors are then "signalled" by a call on an error function. In PSL, the error handler typically calls an interactive break loop, which permits the user to examine the context of the error and optionally make some corrections and continue the computation, or to abort the computation. While in the break loop, the user remains in the binding context of the function that detected the error; the user sees the value of fluid variables as they are in the function itself. If the user aborts the computation, fluid and local variables are unbound. In compiled functions, due to allocation of local variables in the (system) stack, local variables are invisible to the error handler and the debugging facilities. Therefore, compilation can be recommended for fully debugged code, only.

12.2 The Basic Error Functions

(error NUMBER: integer MESSAGE:any): None Returned *expr*
Under the initial (and usual) values of a couple of switches, the error message is printed and an interactive break loop (see below) is entered. If the user "quits" out of the interactive break loop, control returns to the innermost error handler.

The user may supply an error handler. The interactive break loop and the top level loop also supply error handlers, so if the user makes no special preparation, control will return to an existing break loop or to the top level of PSL.

Whenever a call on error results in return to an error handler, the error number of the error becomes the value returned by the error handler. Fluid variables and local bindings are unbound to return to the environment of

the error handler. Global variables are not affected by the process. The error message is printed with 5 leading asterisks on both the standard output device and the currently selected output device unless the standard output device is not open. If the message is a list it is displayed without top level parentheses. The message from the error call is available for later examination in the global variable `emsg*`.
Note: the exact format of error messages generated by PSL functions described in this document may not be exactly as given and should not be relied upon to be in any particular form. Likewise, error numbers generated by PSL functions are not fixed. Currently, a number of different calls on error result in the same error message and number.

(continuableerror NUMBER:integer MESSAGE:any FORM: form): any *expr*

Similar to error. If an interactive break is entered due to a call on continuableerror, the user has options of "continuing" or "retrying" (see information on the break loop, below). In either of these cases the call on continuableerror returns. The value returned is as described in the documentation of the interactive break loop.

The FORM argument is used for "retrying" after a continuable error. The FORM is generally made to look like a call on the function that signalled the error (actual argument values filled in), and the function signalling the error generally returns with the value returned by the call on continuableerror. For example the call on conterror, in the example below is equivalent to the following call on continuableerror:

```
(continuableerror 99 "Attempt to divide by 0 in DIVIDE"
          (list 'divide (mkquote u) (mkquote v)))
```

The FORM argument may be nil. In this case it is expected that the break will be left via "continue" rather than "retry".
As in the example above, setting up the errorform* can get a bit tricky, often involving mkquoteing of already evaluated arguments. The following macro may be useful.

(conterror [ARGS:any]): any *macro*

The format of ARGS is
`(errornumber formatstring {arguments to printf}
reevalform)`.
The FORMATSTRING is used with the following arguments in a call on bldmsg to build an error message. If the only argument to printf is a string, the FORMATSTRING may be omitted, and no call to bldmsg is made.

REEVALFORM is something like `(foo x y)` which becomes `(list 'foo (mkquote x) (mkquote y))` to be passed to the function continuableerror.

```
(de divide (u v)
  (cond ((zerop v)
         (conterror 99
               "Attempt to divide by 0 in DIVIDE"
               (divide u v)))
        (t (cons (quotient u v) (remainder u v)))
 ))
```

(fatalerror S:any): None Returned *expr*

This function allows neither continuation nor even a return to any error handler.

```
(de fatalerror (s)
  (errorprintf "***** Fatal error: %S" s)
  (while t (quit))
  nil)
```

12.3 Basic Error Handlers

(errset U:form *EMSGP:boolean): any *macro*

Errset and errorset are the basic PSL error handler functions.
If an error occurs during the evaluation of U, the value of number from the associated error call is returned as the value of the errset. There are actually a couple of exceptions. If a continuable error is continued by the user in the interactive break loop, no special return to errset is done. Also if the user requests the computation to be aborted completely back to the top level no return to errset is done.
The boolean argument is evaluated without protection of the error handler. The fluid variable *EMSGP is bound to the boolean value for the evaluation of the form. If the value of *EMSGP is nil when an error occurs no error message is printed and no interactive break loop occurs. In this case control must return to the innermost error handler except for the case of a fatal error.
If errset is returned to in the normal way, its value is a list of one element, the value of the form. If errset is returned to via the error mechanism, its value is the error number of the error call that caused the return.

(errorset U:any *EMSGP:boolean *BACKTRACE:boolean): any *expr*

This is an older function than errset. Errset is generally preferred.
In most respects errorset behaves the same as errset. See the documentation of errset above. Note that errorset is an expr, so U gets evaluated once as the parameter is passed and the result is then evaluated inside errorset. Since errorset itself calls eval on its first argument there are likely to be problems with compiled code that uses errorset.
In addition to binding *EMSGP as errset does, errorset overrides the behavior usually specified by the *BACKTRACE switch. The backtrace behavior of PSL errors during the execution of a form inside an errorset error handler is determined by the second parameter to the errorset.
The following two switches and one global variable are used by the functions in this section. Useage of any of these can be considered advanced.

***emsgp** = [Initially: t] *fluid*
: Fluid variable rebound by errset and errorset. Controls error message printing during call on error. If nil, no error message will be printed and no interactive break loop will be entered. If an unwind backtrace has been requested through the backtrace flag or a call on errorset, one will be.

emsg* = [Initially: nil] *global*
: Contains the message generated by the last error call. Particularly useful in case printing of the message was suppressed.

***backtrace** = [Initially: nil] *switch*
: Used by the top level read-eval-print loop to control whether an unwind backtrace will be printed when errors occur outside the scope of any user-specified error handler.

12.4 Break Loop

On detecting an error, PSL normally enters a read/eval/print loop called a break loop. Here the user can examine the state of his computation, change the values of fluid and global variables, or define missing functions. He can then dismiss the error call to the normal error handling mechanism (errorset or errset above). If the error was of the continuable type, he may continue the computation. By setting the switch *break to nil, all break loops can be suppressed, and just an error message is displayed. Suppressing error

messages also suppresses break loops.

***break** = [Initially: t] *switch*
Controls whether the break package is called before unwinding the stack on error.

breaklevel* = [Initially: 0] *global*
The current number of nesting level of breaks.

maxbreaklevel* = [Initially: 5] *global*
The maximum number of nesting levels of breaks permitted. If an error occurs with at least this number of nested breaks already existing, no entry to an interactive break loop is made. Control aborts back to the innermost error handler instead.

The prompt `"lisp break>"` indicates that PSL has entered a break loop. A message of the form "Retry form is ..." may also be printed, in which case the user is able to continue his computation by repairing the offending expression. By default, a break loop uses the functions read, eval, and print. This may be changed by setting `breakreader*, breakevaluator*, or breakprinter*` to the appropriate function name.

errorform* = [Initially: nil] *global*
Contains an expression to reevaluate inside a break loop for continuable errors. Used as a tag for various Error functions.

Several ids, if typed at top-level, are special in a break loop. These are used as commands, and are currently M, R, T, Q, A, I, and C. They call functions stored on their property lists under the indicator breakfunction. These ids are special only at top-level, and do not cause any difficulty if used as variables inside expressions. However, they may not be simply typed at top-level to see their values. This is not expected to cause any difficulty.

The meanings of these commands are:

M Display errorform*, this command calls the function breakerrmsg.

R Retry. This tries to evaluate the retry form, and continue the computation. It evaluates the value of errorform*. This is often useful after defining a missing function or assigning a value to a variable. This command calls the function breakretry.

C Continue. This causes the expression last printed by the break loop to be returned as the value of the call on continuableerror. This is often useful as an automatic stub. If an expression containing an undefined function is evaluated, a break loop is entered, and this may be used to return the value of the function call. This command calls the function breakcontinue.

Q Quit. This exits the break loop by throwing to the closest surrounding error handler. It calls the function breakquit.

A Abort. This aborts to the top level, i.e., restarts PSL. It calls the function reset.

T Trace. This prints a backtrace of function calls on the stack except for those on the lists ignoredinbacktrace* and interpreterfunctions*. It calls the function backtrace.

I Interpreter Trace. This prints a backtrace of only interpreted functions call on the stack except for those on the list interpreterfunctions*. It calls the function interpbacktrace.

An attempt to continue a non-continuable error with R or C prints a message and behaves as Q.

ignoredinbacktrace* = [Initially: string] *global*

A list of function names that will not be printed by the commands I and T given within a break loop (eval apply fastapply codeapply codeevalapply catch errorset evprogn toploop breakeval bindeval break main).

interpreterfunctions* = [Initially: string] *global*

A list of function names that will not be printed by the command I given within a break loop (cond prog and or progn setq).

The above two globals can be reset in an init file if the programmer desires to do so.

```
(de new-nth (seq index)
    (cond ((onep index) (car x))
          (t (new-nth (rest index) (sub1 index)))))

1 lisp> (new-nth '(a b c) 2)
***** 'S' is an unbound ID
```

```
***** Continuable error: retry form is 'S'
2 lisp break (1)> t
Backtrace from top of stack:
car new-nth new-nth
NIL
3 lisp break (1)> seq
(b c)
4 lisp break (1)> c
b
```

12.5 Details on the Break Loop

If the switch break is t, the function break is called by error or continuableerror before unwinding the stacks, or printing a backtrace. Input and output to/from break loops is done from/to the values (channels) of `breakin*` and `breakout*`. The channels selected on entrance to the break loop are restored upon exit.

breakin* = [Initially: nil] *global*
So rds chooses stdin*.

breakout* = [Initially: nil] *global*
Similar to breakin*.

Break is essentially a read-eval-print function, called in the error context. Any fluid may be printed or changed, function definitions changed, etc. The break uses the normal toploop mechanism (including history), embedded in a catch with tag *break*. The toploop attempts to use the parent loop's toploopread*, toploopprint* and toploopeval*; the breakeval function first checks top-level ids to see if they have a special breakfunction on their property lists, stored under breakfunction. This is expected to be a function of no arguments, and is applied instead of eval.

12.6 Some Convenient Error Calls

The following functions may be useful in user packages:

(rangeerror OBJECT:any INDEX:integer FN:function): None *expr*

```
(stderror
(bldmsg "Index %r out of range for %p in %p" INDEX OBJECT
FN))
```

(stderror MESSAGE:string): None Returned *expr*

```
(error 99 message)
```

(typeerror OFFENDER:any FN:function TYP:any): None Returned *expr*

```
(stderror (bldmsg
"An attempt was made to do %p on %r, which is not %w"
FN OFFENDER TYP))
```

(usagetypeerror OFF:any FN:function TYP:any USAGE:any): None Returned *expr*

```
(stderror (bldmsg
"An attempt was made to use %r as %w in %p, where %w is
needed"+ OFFENDER USAGE FN TYP))
```

(nonwordserror OFFENDER:any FN:function): None Returned *expr*

```
(typeerror OFFENDER FN "a words vector")
```

(nonpairerror OFFENDER:any FN:function): None Returned *expr*

```
(typeerror OFFENDER FN "a pair")
```

(nonlisterror OFFENDER:any FN:function): None Returned *expr*

```
(typeerror OFFENDER FN "a list or NIL")
```

(noniderror OFFENDER:any FN:function): None Returned *expr*

```
(typeerror OFFENDER FN "an identifier")
```

(nonnumbererror OFFENDER:any FN:function): None Returned
expr
```
(typeerror OFFENDER FN "a number")
```

(nonintegererror OFFENDER:any FN:function): None Returned
expr
```
(typeerror OFFENDER FN "an integer")
```

(nonpositiveintegererror OFFENDER:any FN:function): None*expr*
```
(typeerror OFFENDER FN "a non-negative integer")
```

(noncharactererror OFFENDER:any FN:function): None Returned *expr*
```
(typeerror OFFENDER FN "a character")
```

(nonstringerror OFFENDER:any FN:function): None Returned
expr
```
(typeerror OFFENDER FN "a string")
```

(nonvectorerror OFFENDER:any FN:function): None Returned
expr
```
(typeerror OFFENDER FN "a vector")
```

(nonwordserror OFFENDER:any FN:function): None Returned
expr
```
(typeerror OFFENDER FN "a words vector")
```

(nonsequenceerror OFFENDER:any FN:function): None Returned
expr
```
(typeerror OFFENDER FN "a sequence")
```

Debugging Tools

13.1 The Debug Module

debug is a loadable option. This module provides a basic debugging tool which provides a breakpoint function, and functions for breaking and tracing functions and methods selected by the user. debug is loaded automatically if one of the functions tr, trst or br is called.

13.1.1 Overview of Functionality

The function breakpoint allows the user to set a breakpoint anywhere within his code. Whenever a breakpoint is called a continuable break loop is entered. This allows the user to examine or modify variables, view a backtrace, and continue execution.

Tracing gives a record of the sequence of calls made to a set of functions specified by the user. When a traced function is entered a message is printed giving the function name, level of recursion, and values of any arguments. Upon function exit another message is printed, giving the value returned by the function.

Breaking a function means that the system enters a continuable break loop both before and after evaluation of the body of the function. Within the break loop the user may inspect and change values of parameters and non-local variables, and then continue execution. A break on entry occurs after the function's formal parameters have been bound to the actual arguments but before any of the forms in the function body have been evaluated. The break on exit occurs after the body of the broken function has been evaluated.

13.1.2 Using Break and Trace

(tr [NAME: id]): list *macro*
If no arguments are given then a list is returned which indicates which functions are currently being traced. Otherwise, each argument is treated as a reference to a function, and if possible, the function is set up for tracing. The list which is returned represents the functions which have been added to the set of functions being traced.

(trst [NAME: id]): list *macro*
If no arguments are given then a list is returned which indicates which functions are currently being traced. Otherwise, each argument is treated as a reference to a function, and if possible, the function is set up for tracing. The list which is returned represents the functions which have been added to the set of functions being traced. Trst can be used only for functions which are not compiled. Additionally to the entry and exit messages, for each *setq* assignment inside the function the variable name and the value are printed.

(br [NAME: id, method-specification]): list *macro*
If no arguments are given then a list is returned which indicates which functions are currently broken. NAME is treated as a reference to a function, and if possible, the function is set up for breaking. Subsequent arguments will be processed. The list which is returned represents the functions which have been added to the set of broken functions.

(untr [NAME:id]): list *macro*
If no arguments are given then tracing is removed for each traced function. NAME is treated as a reference to a function, and if possible, tracing is disabled for this function. Subsequent arguments will be processed. The list which is returned contains the names of the functions for which tracing has been removed.

(unbr [NAME:id,]): list *macro*
If no arguments are given then breaking is turned off for each broken function. NAME is treated as a reference to a function, and if possible, breaking is disabled for this function. The list which is returned contains the names of the functions for which breaking has been removed.

13.1.3 Sample Session

Trace and break printout goes to the standard output channel. Because multiple nested calls are usually shown, indentation and vertical bars are used to line up function entry and return messages for the same function.

```
(de add3 (a1 a2 a3)
  (+ a1 (add2 a2 a3)))

(de add2 (a1 a2)
  (+ a1 a2))

(de fact (n)
  (cond ((zerop n) 1)
        (t (times n (fact (sub1 n))))))

1 lisp> (br add3)
In a break, t prints backTrace; c Continues;
q Quits a level; a Aborts to top.
(ADD3)
2 lisp> (tr 'add3)
***** (QUOTE ADD3) is not a valid method-specification.
NIL
3 lisp> (tr add3 add2)
(ADD2 ADD3)
4 lisp> (add3 5 6 7)
ADD3 entry:
|     A1: 5
|     A2: 6
|     A3: 7
|  ADD3 entry BREAK:
|  NMODE Break loop
5 lisp break (1)> c     % Continue
|  |  ADD2 entry:
|  |  |     A1: 6
|  |  |     A2: 7
|  |  ADD2 value = 13
```

```
|  ADD3 exit BREAK, value = 18:
|  NMODE Break loop
6 lisp break (1)> c
ADD4 value = 18
18

1 lisp> (tr fact)
(FACT)
2 lisp> (fact 4)
FACT entry:
|     N: 4
|  FACT reentry (# 2):
|  |        % (# 2) represents the depth of recursion
|  |     N: 3
|  |  FACT reentry (# 3):
|  |  |     N: 2
|  |  |  FACT reentry (# 4):
|  |  |  |     N: 1
|  |  |  |  FACT reentry (# 5):
|  |  |  |  |     N: 0
|  |  |  |  FACT value (# 5) = 1
|  |  |  FACT value (# 4) = 1
|  |  FACT value (# 3) = 2
|  FACT value (# 2) = 6
FACT value = 24
24
3 lisp> (trace)
(FACT ADD2 ADD3 ADD4)
4 lisp> (untrace)
(ADD4 ADD3 ADD2 FACT)
5 lisp> (restr)
T
```

Tracing a macro will happen at macroexpand time. The value returned will be the macroexpanded form, and thus is quite useful in determining if your macro expanded properly. Note that when code is complied the application of a macro is expanded and this expansion replaces the the application.

```
1 lisp> (trace plus)
(PLUS)
2 lisp> (plus 2 9 -3 7 8)
PLUS entry:
|     ARG1 :(PLUS 2 9 -3 7 ...)
```

```
PLUS value = (PLUS2 2 (PLUS2 9 (PLUS2 -3 (PLUS2 7 8))))
23

(de fact (n)
  (cond ((zerop n) 1)
        (t (breakpoint "Fact; n<>0 branch of cond, n=%p" n)
           (times n (fact (sub1 n))))))

1 lisp> (fact 2)
User Breakpoint: Fact; n<>0 branch of cond, n=2
NMODE Break loop
2 lisp break (1)> c
User Breakpoint: Fact; n<>0 branch of cond, n=1
NMODE Break loop
3 lisp break (1)> c
2
```

13.1.4 Redefining a Broken or Traced Function

The basic definition of a function is the definition without any of the break or tracing side effects. This basic definition may be accessed by the function getd. The basic definition may also be redefined by using the function putd without interfering with the break/trace side effects, as long as the parameter list stays the same. If you intend to change the number or order of parameters you should first remove the break/trace wrappers from it with untrace and/or unbr.

Miscellaneous Utilities

14.1 Simulating a Stack

The following macros are in the USEFUL package. They are convenient for adding and deleting things from the head of a list.

(push ITM:any STK:list): any *macro*
(push item stack) is equivalent to
(setf stack (cons item stack))

(pop STK:list): any *macro*
(pop stack) does (setf stack (cdr stack)) and returns the item popped off stack. An additional argument may be supplied to Pop, in which case it is a variable which is SetQ'd to the popped value.

14.2 Ring Buffers

The code-address-to-symbol function takes an integer argument, and attempts to find out what function that address is in. If the address does not map to any function, the name of the most recently loaded function is returned.

(code-address-to-symbol ADDRESS:integer): id *expr*

for example:

```
(code-address-to-symbol 16#38393)
```

This function is available by evaluating (load addr2id).

14.3 Word Vector Operations

These functions are defined in $pu/vector-fix.sl resp the vector-fix module.

(mkwords N:integer): vector *expr*
Allocates a vector of N words with all elements initialized to zero.

(truncatevector V:vector I:integer): vector *expr*
Truncates V to I elements.

(truncatewords V:words I:integer): vector *expr*
Truncates V to I elements.

(getwords WRD:words I:integer): any *expr*
Retrieves the I'th entry of WRD.

(putwords WRD:words I:integer VAL:any): any *expr*
Store VAL at I'th position of WRD.

(upbw V:words): integer *expr*
Returns the upper limit of words V.

Compiler

15.1 Introduction

The evaluation of an expression which has been compiled may be different from interpretation. Some of the more important differences are listed here. Subsequent sections of this chapter will discuss these differences in more detail.
The application of a macro is replaced by its expansion. Therefore, the definition of the macro must preceed its use. Since the call on the macro has been replaced, a redefinition of the macro will have not have an effect on the compiled expression.
A constant expression is replaced by its value (see the section Constant Declaration for more information)
There are some functions which are defined in two ways. One of these versions will reduce the execution time by not verifying that the arguments are of correct type. Which function is called from the compiled code is based upon the value of a switch. If the switch is non-nil then the faster version replaces the slower one. Note that an instance of the faster version is not replaced by the slower one when the switch is set to nil (see the section Switches that Control the Compiler for more information).
An attempt is made to convert recursive control structures to iterative ones.

15.2 Compiling Files

It is best to compile files using `pslcomp`, a minimal version of PSL capable of compiling files. The successful compilation of a file should not depend upon optional modules which have been loaded. On some computer systems the names of the files to be compiled can be listed as command line arguments. When PSLCOMP is brought up the following banner is displayed.
Portable Standard Lisp Compiler
Usage: PSLCOMP source-file ...
If an error occurs, the message

```
***** Error during compilation of NAME
```

is printed and the compilation is aborted.

(compile-file FILE:string): {t, nil, abort} *expr*

This function is used to compile files. A binary file is created with the same name except that the file type is changed to **b**. Beware, it is possible to overwrite the source file if the operating system truncates file names (this can be avoided by using compile-file-aux, described below). If the file name is without a suffix then **sl** is assumed. This function will return t if the compilation was successful, abort if after an error was signalled the user aborted the compilation, or nil if the source file was not found. When the source file cannot be found the message

```
***** Unable to find source file for: FILE
```

will be printed. Prior to the compilation the message

```
----- Compiling SOURCE-FILE to BINARY-FILE
```

is displayed. In addition, one of the following two messages will be printed when compilation has stopped.

```
----- Compilation of NAME completed

----- Compilation of NAME aborted
```

(compile-file-aux SOURCE:string BINARY:string): {t, nil, abort} *expr*

Since this function is used by compile-file to compile files, most of the comments concerning compile-file apply to compile-file-aux. Although one application of compile-file may be used to compile a number of files, with this function only one file can be compiled. The advantage of using compile-file-aux is that the name of the binary file is given explicitly.

During the compilation of a file, if an atomic expression is read at the top level the

```
*** Warning: non-list form ignored: ATOM
```

will be displayed and the expression will be ignored. A list whose first element is not atomic is saved for insertion into the an initialization function which is added to the binary file by the compiler (its name is **fasl**initcode**).

The processing of a list whose first element is an id depends in large part upon values found on the property list of the id. The EVAL flag will cause the expression to be evaluated, using the current binding of toploopeval*, and compiled. The expression is only evaluated, again using the current binding of toploopeval*, if the id is flagged IGNORE. All other expressions are compiled. Compilation will modify the environment, in particular, all macro definitions are installed. Note that although defflavor is a macro it does not create a macro definition. It will however modify the property list of the flavor name.
As expressions are processed a list of expressions is built. When a function definition is made an application of putd is added to the list. Other expressions which are intended to be evaluated at load time were also added. This sequence of instructions, which preserves the order in which expressions were read, is compiled as the body of the initialization function (named **fasl**initcode**).

15.2.1 Order of Functions for Compilation

Functions whose type is not expr must be defined before their use in a compiled function, since the type of a function affects how it's application will be treated by the compiler. The application of a macro is replaced by the expansion of the macro. For functions whose type is fexpr it is necessary to gather the unevaluated arguments into a list. Calling a function which is an nexpr requires that the arguments be evaluated and gathered into a list. When the compiler cannot determine the type of a function, because it has not been defined, it assumes that the type of the function is expr. Note that it is not possible to define a recursive function, whose type is not expr, without first defining a dummy version.

15.3 Compiling Functions into Memory

Functions can be compiled directly into memory by turning on the switch comp, by evaluating (on comp). This compilation is accomplished in part by a call on the function compd from within the function putd.

(compd NAME:id TYPE:ftype BODY:lambda): NAME:id *expr*
The function compd is analogous to the function putd.

Once comp is set to t subsequent function definitions are automatically compiled and a message of the form

```
*** (NAME): base <number>, length <number> bytes
```

is displayed. Unfortunately this information is of little use unless one happens to be doing something like debugging the compiler.

(compile NAMES:id-list): any *expr*
This function is used to compile resident functions. It is possible to compile a function which is being traced. There are two warning messages which may be printed.

```
*** NAME already compiled
```

This indicates that the function associated with the function name is already compiled. If there is no function definition associated with the name then the following message will be displayed.

```
*** No definition for NAME
```

15.4 Compiler Errors and Warnings

In addition to the error messages mentioned in this chapter, the following error messages may be displayed. In general, the five star prefix is used for error messages, the three star prefix for warnings. The display of warning messages can be suppressed by evaluating (off msg).

```
***** Too many arguments N
```

The restriction on the number of arguments is based in part on the fact that arguments to compiled functions are passed in general purpose registers rather than directly on a stack. The compiler will generate code for the offending definition. Although the code appears to be correct, attempting to apply the function will result in error.

```
***** Attempt to compile non-lambda expression FORM
```

This error occurs within the context of a call on compd, when the expression is not a valid lambda expression.

```
*** NAME has not been defined, because it is flagged LOSE
```

If a function name is flagged LOSE then it cannot be redefined. Attempting to compile an interpretive definition is considered redefinition in this context.

```
***** Ill-formed function expression FORM
```

The offending expression is an instance of function application. This message will appear if the function is atomic and not an id, or if the function is a list which is not a valid lambda expression.

```
***** Odd number of arguments to SETQ FORM
```

The number of arguments passed to a SETQ must be even.

```
***** Incorrect number of arguments to NAME
```

The number of arguments passed to either apply, idapply, codeapply, cons, go, or return was not correct.

15.5 Differences between Compiled and Interpreted Code

In the process of compilation, many functions are Open-Coded, and hence cannot be redefined or traced in the compiled code. Such functions are noted to be Open-Coded in the manual. The call on the function is replaced by a sequence of instructions which make up the body of the function. This trick gives faster execution, but it requires more space. A call on a macro is another instance of open coding. A PSL program usually consists of a large number of relatively small functions. Calling a function requires manipulation of a parameter stack for variable binding and a control stack for return information. The time spent inside a function may be small compared to the time spent maintaining the stacks. Some commonly used PSL functions which are open-coded are and, or, apply, idapply, codeapply, return, go, cons, cond, case, prog, progn, and prog2.

Unless variables are declared, or detected, to be fluid or global they are compiled as local variables. This causes their names to disappear, and so they are not visible on the binding stack. In addition these variables will not be available to functions called in the dynamic scope of the function containing their binding.

The compiler attempts to do the following conversions

(function (lambda ...)) − >	the lambda expression is compiled into a gensymed name. If the functional form of one of the mapping functions (map, mapc, maplist, mapcar, mapcon, and mapcan) is explicitly tagged function then the call is open coded.
(setq na va nb vb ...) − >	(setq na va), (setq nb vb) ...
(nth sequence n) − >	(car sequence) if n = 1, (cadr sequence) if n = 2, (caddr sequence) if n = 3, (cadddr sequence) if n = 4
(pnth sequence n) − >	(identity sequence) if n = 1, (cdr sequence) if n = 2, (cddr sequence) if n = 3, (cdddr sequence) if n = 4, (cddddr sequence) if n = 5

The following transformations are made when the list function is applied to less than six arguments. When there are more that five arguments the compiler uses the cons expansion for additional arguments.

(list a) − >	(ncons a)
(list a b) − >	(list2 a b)
(list a b c) − >	(list3 a b c)
(list a b c d) − >	(list4 a b c d)
(list a b c d e) − >	(list5 a b c d e)
(apply (function foo) ...) − >	(foo ...)
(assoc ...) − >	(atsoc ...)

If the last clause within a cond expression is not of the

form (t ...) then the compiler will add the additinal clause (t nil).

(difference n 1) − >	(sub1 n)
(equal ...) − >	(eq ...)
(geq ...) − >	(not (lessp ...))
(intern (compress ...)) − >	(implode ...)
(intern (gensym ...)) − >	(interngensym ...)
(leq ...) − >	(not (greaterp ...))
(lessp n 0) − >	(minusp n)
(member ...) − >	(memq ...)
(neq ...) − >	(not (equal ...))
(not ...) − >	(null ...)
(plus2 n 1) − >	(add1 n)
(null (eq ...)) − >	(neq ...)
(null (atom ...)) − >	(pairp ...)

Prior to the application of some functions, such as car and cdr, there is no verification that the types of the arguments are correct.

Since compiled calls on macros, fexprs, and nexprs are different from the default exprs, these functions must be defined, or declared, before compiling the code that uses them. While fexprs and nexprs may be subsequently redefined, as new functions of the same type, macros are executed by the compiler to get the replacement form, which is then compiled. The interpreter would pick up the most recent definition of any function, and so functions can switch type as well as body.

Constants which appear in code that is to be compiled may be collapsed. For example, the following code references two distinct strings which happen to contain the same characters. The compiled code will only reference one string.

```
  (de fubar ()
    (let ((this "foo")
          (that "foo"))
    (eq this that))

1 lisp> (foo)
nil
2 lisp> (compile '(foo))
nil
3 lisp> (foo)
t
```

As noted below, the switches *r2i and *nolinke may cause function calls to be replaced by jumps. This means that the backtrace of compiled functions may differ from that of interpreted functions. In addition a sequence of interpreted function calls could consume stack space while compiled function calls might not.

15.6 Constant Declaration

(define-constant NAME:id VALUE:any): any *macro*
The identifier NAME is declared a constant and its valuecell is set to VALUE. This provides a means for informing the compiler that the value of an id is constant. If a constant id is used in such a way that during evaluation its value would be accessed the compiler can then simply replace the id by its value. It is possible to use an id which has been declared a constant as a local variable within a prog or as a parameter name. It is possible to redefine the value of a constant.

Request to set constant NAME to a different value.

(constant? NAME:id): boolean *expr*
Returns t if NAME has been declared a constant with define-constant, otherwise nil.
The compiler attempts to replace function applications by the corresponding value whenever each of the arguments is either constant or an application which may be replaced by its value. The following is a list of all functions whose application may be evaluated during compilation. Clearly some functions like cons cannot be evaluated at compile time.

```
// /= < << <= = > >= >> ^ | ~ ~=
```

abs aconc acos acosd acot acotd acsc acscd add1 alphanumbericp alphap asec asecd asin asind ass assoc atan atan2 atan2d atand atom atsoc

bldmsg bothcasep

car cdr ceiling char-downcase char-equal char-font char- greaterp char-int char-lessp char-upcase char< char= char> constantp cos cosd cot cotd csc cscd

degreestodms degreestoradians difference digit digit-char digitp divide dmstodegrees dmstoradians

eq eqcar eqn eqstr equal expt

factorial fix fixp float floatp floor

geq getv

land lastcar lastpair length leq lessp list lnot log log10 log2 lor lowercasep lshift lxor

max2 member memq min2 minus minusp mkquote mod

ne neq not nth null

onep

pairp plus2 pnth

radianstodegrees radianstodms recip remainder rest reserse round

sec secd sin sind size sqrt string-not-greaterp string-not- lessp string-repeat string-search string-search-equal string- search-from-string-search-from-equal string-size string-trim string-upcase string-upper-bound string< string<= string<> string= string> string>= stringp sub sub1 substring-equal sub-string=

tan tand times2

upbv uppercasep

vector-empty? vector-fetch vector-size vector-upper-bound vector2l ist vec-tor 2string vectorp

There is another facility in PSL for defining constants. You are encouraged to use the function define-constant. The following two functions are documented here for completeness.

(defconst [U:id V:any]): undefined *macro*

Defconst provides for the definition and use of symbolic constants. Each U is defined to represent the corresponding value V. The value V is stored on the property list of U, not in the valuecell. The macro const is used to retrieve the value.

(const U:id): any *macro*

Const gives access to the value defined by the function defconst. The value will be evaluated.

```
1 lisp> (defconst foo (add1 2))
(add1 2)
```

```
2 lisp> (const foo)
3
```

15.7 Fluid and Global Declarations

Fluid and global declarations must be used to indicate variables that are to be used as non-local variables in compiled code. The values associated with local variables are stored on a stack. If a variable has been declared global or fluid then the compiled code will instead access the value cell of that variable. In addition the content of the value cells of special variables being used as formal parameters must be preserved. The compiler currently defaults variables bound in a particular function to local. Note that local variables exist as anonymous stack locations, therefore called functions cannot see them. An undeclared non-local variable is declared fluid by the compiler with the warning:

```
*** NAME declared fluid
```

The appearance of this message may indicate an additional error. If a previous function used this name in the parameter list then it may have to be recompiled. The local variable in the previous function will not refer to the same variable as the one recently declared fluid, even though the names are the same in the source code listing.
Previous documentation has given users the impression that it is illegal to bind an id which has been declared global. This is not true, an attempt to bind a global variable will cause the following message to be displayed.

```
*** Illegal to bind global VARIABLE but binding anyway
```

The lack of distinction between global and fluid in PSL appears to be due to the fact that shallow binding is employed in PSL. In a deep bound system one has to search down a binding stack to locate the value of a fluid id, which could get expensive. A global declaration within a deep bound system would allow the interpreter to assume that the value of an id is in a fixed location. Of course in a shallow bound system the value of a id is always found in the value cell of that id.

15.8 Control Over the Time When Something is Done

Which expressions are evaluated during compilation only, which are written to the file to be evaluated at load time, and which do both can be controlled

by the EVAL and IGNORE properties of function names. In addition, the following functions may also be used.

(commentoutcode U:form): nil *macro*
Comment out a single expression.

(compiletime U:form): nil *expr*
Evaluate the expression U at compile time only.

(bothtimes U:form): U:form *expr*
Evaluate the expression U at both compile time and load time.

(loadtime U:form): U:form *expr*
Evaluate the expression at load time only.

15.9 Switches That Control the Compiler

The compilation process is controlled by a number of switches. In addition to comp there are the following switches.

***fast-vectors** = [Initially: t] *switch*

***fast-evectors** = [Initially: t] *switch*

***fast-strings** = [Initially: nil] *switch*

***fast-integers** = [Initially: nil] *switch*

There are some functions which are defined in two ways. One of these versions will reduce the execution time by not verifying that the arguments are of correct type. The name of the slower version will have two properties, FAST-FLAG and FAST-FUNCTION. The FAST-FLAG property references a switch whose value determines which function will be called from compiled code. An instance of the slower function is replaced by the faster one when the switch is set to t. Of course, to make this conversion the compiler has to know the name of the faster function. This name is referenced by the FAST-FUNCTION property.

The property list of the function vector-fetch looks something like

```
(... (fast-flag . *fast-vectors) (fast-function . igetv) ...)
```

The expression (vector-fetch vector index) will be replaced by the expression (igetv vector index) if the switch *fast-vectors is set to t.

***r2i** = [Initially: t] *switch*

If non-nil, recursive calls are converted to jumps whenever possible. The effect of this is that tracing this function will not show the recursive calls since they have been eliminated. The function RECURSIVE-FACTORIAL contains a recursive call which can be converted to a jump. The result of compiling RECURSIVE-FACTORIAL is almost identical to that of compiling ITERATIVE-FACTORIAL when r2i is non-nil.

```
(de factorial (n)  (iterative-factorial (1 n)))


(de recursive-factorial (accumulator count)
  (if (<= count 1)
     accumulator
     (recursive-factorial (* accumulator count)
                          (sub1 count))))


 (de iterative-factorial (accumulator count)
  (prog ()
   loop
     (when (<= count 1) (return accumulator))
     (setf accumulator (* accumulator count))
     (decr count)
     (go loop)))
```

***nolinke** = [Initially: nil] *switch*

If nil, when the last form to be evaluated is a function call then the compiler will generate an instruction which will perform two actions during evaluation. Prior to jumping to the entry point of the function being called, the space allocated on the stack for local variable storage is reclaimed. Although the amount of space reclaimed may be small, this does provide the called function with more stack space. In contrast, when this switch is non-nil a pair of instructions would be used, a call on the function followed by an exit. In this case the stack space is not reclaimed until the exit.

***ord** = [Initially: nil] *switch*

If non-nil, the compiler is forced to generate code which evaluates arguments of a function call in a left to right order. It is possible to generate code which is more effecient when this switch is set to nil. Arguments are currently passed in registers. As the calling function computes each argument the result is stored on the stack. References to constants need not be generated and then pushed onto the stack, the other arguments can be compiled first, then just before the function is called the appropriate register is loaded with the constant. A similar argument can be made for variable references. However, allowing for side effects we must be sure that the postponed value would be the same as that fetched at the proper time.

***plap** = [Initially: nil] *switch*

If non-nil, the portable intermediate code produced by the compiler is printed. This is useful for examining compiler output prior to assembly, in particular it can be helpful in debugging macros.

***pgwd** = [Initially: nil] *switch*

If non-nil, the actual assembly language mnemonics are displayed. This flag is useful for examining the code which is generated for the initialization function **fasl**initcode**.

***pcmac** = [Initially: nil] *switch*

If non-nil, both the portable intermediate code and the assembly mnemonics are displayed. After each intermediate instruction the assembly mnemonic is shown.

***pwrds** = [Initially: t] *switch*

If non-nil, the address and size of compiled functions are displayed as they are defined.

15.10 Conditional Compilation

(if_system SYS-NAME:id TRUE-CASE:any FALSE-CASE:any): any *macro*

This is a conditional macro for choosing system dependent code during compilation. The expression for the false case defaults to nil if it is ommited. The id must be a member of the list whose name is system_list*. An example of its use follows.

```
(if_system  tops20  (setq buildfileformat* "PL:%W"))
```

15.11 Implementation Details

The output from the compiler is a list of instructions in the order which they will be executed. Each instruction is a list, an operation followed by as many items as are required by that operation. The compiled code may be executed by simulating the actions of the machine on each element of the sequence. In the interest of efficiency, the compiler ouput is translated into a sequence of actual machine instructions. An area of memory (which is not garbage collected), is allocated to receive the processed compiler output. This area is called the Binary Program Space or BPS. The function lap translates the output from the compiler into machine instructions.
The functions faslout, faslend, and faslabort are used by Compile-file-aux to compile files. It should not be necessary for you to used these functions, they are included for completeness. Evaluation of

```
(faslout name)
```

will cause the message
FASLOUT: (DSKIN files) or type in expressions
When all done execute (FASLEND)
to be displayed. Subsequent expressions, typed in or read from files, are not processed in the normal fashion. The following piece of code demonstrates how this feature is implemented.

```
(de toploop
    ...
  (cond ((not *defn) (top-loop-eval-and-print input-value))
        (dfprint* (funcall dfprint* input-value))
        (t (prettyprint input-value)))
    ... )
```

When the switch *defn is nil the input is evaluated and the result is printed. Setting *defn to t and associating a function with the global dfprint* allows for the redefinition of what is meant by evaluating the input and printing the result of that evaluation. It the function bound to dfprint* which is responsible for the compilation of expressions. Faslend is provided to discontinue compilation, faslabort is used to abort the compilation. The only context in which the application of these functions makes sense is when a previous invocation of faslout has not been terminated.

```
***** FASLEND not within FASLOUT
```

Miscellaneous Useful Features

16.1 Exiting PSL

(quit): Undefined *expr*
Return from LISP and terminate the PSL process.

(exitlisp): Undefined *expr*
Return from LISP and terminate the PSL process.

(exitwithstatus STATUS:integer): Undefined *expr*
Return from LISP and terminate the PSL process. STATUS is passed to the calling command shell as return value.

16.2 Saving an Executable PSL

(savesystem MSG:string FILE:string FORMS:form-list): Undefined *expr*

> This records the welcome message (after attaching a date) in the global variable lispbanner* used by standardlisp's call on toploop, and then calls dumplisp to compact the core image and write it out as a machine dependent executable file with the name FILE. FILE should have the appropriate extension for an executable file. Savesystem also sets *user-mode to t.

The forms in the list FORMS will be evaluated when the new core image is started. For example

```
(savesystem "PSL 4.2" "PSL" '((read-init-file "psl")))
```

lispbanner* = [Initially:] *global*

Records the welcome message given by a call to savesystem from PSL. Also contains the date, given by the function date.

(dumplisp FILE:string): Undefined *expr*

The core image is written as an executable file, with the name FILE. Better use the function savesystem.

16.3 Init Files

Init files are available to make it easier for the user to customize PSL to his/her own needs. When psl or pslcomp is executed, if a file .pslrc or .pslcomprc on UNIX boxes or psl.rc or pslcomp.rc on DOS or Windows systems is on the users home directory, it will be read and evaluated.
The remainder of this section describes functions defined in the module **initfile**. These functions can be used to implement init files.

(user-homedir-string): string *expr*

Returns a full pathname for the user's home directory.

(init-file-string progname:string): string *expr*

Returns the full pathname of the user's init file for the program progname.

(read-init-file progname:string): nil *expr*

Reads and evaluates the init file with name progname. Read-init-file calls init-file-string with argument progname. Under UNIX the initfilename is concatenated from "." progname and "rc", such that a typical initfilename is .pslrc for psl

```
(read-init-file "psl")
```

16.4 Miscellaneous Functions

(reset): Undefined *expr*

Return to the top level of PSL, unwind-protect forms get a chance to run.

(time): integer *expr*
CPU time in milliseconds since login time.

(date): string *expr*
The date in the form "day-month-year"

```
1 lisp> (date)
"21-Jan-1997"
```

(date-and-time): string *expr*
A string is returned which describes the date, followed by the time. The string is of the form
"day-month-year hours:minutes:seconds"

```
1 lisp> (date-and-time)
"21-Jan-1997 17:04:14"
```

16.5 Garbage Collection

(reclaim): nil *expr*
Reclaim is the user level call to the garbage collector. Active data in the heap is made contiguous and all tagged pointers into the heap from active local stack frames, the binding stack and the symbol table are relocated. If *gc is t, prints some statistics. Increments gcknt* and updates gctime*.

(known-free-space): integer *expr*
Returns the number of items available in the heap.

(free-bps): integer *expr*
Returns the number of items available in the BPS (Binary Program Space).

***gc** = [Initially: t] *switch*

*Gc controls the printing of garbage collector messages. If nil no indication of garbage collection occurs. If non-nil various messages will be displayed.

```
1 lisp> (reclaim)
*** Gargabe collection starting
*** GC 1:  8-Nov-97 15:49:54, 50 ms (50 %), 218 recovered, 260679 free
nil
```

As the example shows, the amount of time (in milliseconds), required to do the garbage collection, the number of items recovered, and the number of items available in the heap are displayed. The first number which follows GC represents the value of gcknt*. The inclusion of the date into the message has been found useful for calculations which take extremely long.

gctime* = [Initially: 0] *global*

The total time (in milliseconds) spent in garbage collection. When the garbage collector is invoked, the time spent performing the collection is added to the value of gctime*

gcknt* = [Initially: 0] *global*

Records the number of times that the garbage collector has been invoked. Gcknt* may be reset to another value to record counts incrementally, as desired.

Cross Reference Tools

17.1 Introduction

This chapter describes tools which build cross references, and which aid in analyzing LISP code for building other cross reference - like tools.

17.1.1 RCREF - Cross Reference Generator for PSL Files

rcref is a Standard LISP program for processing a set of PSL function definitions to produce:

1. A "Summary" showing:
    - A list of files processed.
    - A list of "entry points" (functions which are not called or are called only by themselves).
    - A list of undefined functions (functions called but not defined in this set of functions).
    - A list of variables that were used non-locally but not declared GLOBAL or FLUID before their use.
    - A list of variables that were declared GLOBAL but used as FLUIDs (i.e. bound in a function).
    - A list of FLUID variables that were not bound in a function so that one might consider declaring them GLOBALs.
    - A list of all GLOBAL variables present.
    - A list of all FLUID variables present.
    - A list of all functions present.
2. A "global variable usage" table, showing for each non-local variable:
    - Functions in which it is used as a declared FLUID or GLOBAL.

- Functions in which it is used but not declared before.
- Functions in which it is bound.
- Functions in which it is changed by setq.

3. A "function usage" table showing for each function:

   - Where it is defined.
   - Functions which call this function.
   - Functions called by it.
   - Non-local variables used.

The output is alphabetized on the first seven characters of each function name.
RCREF also checks that functions are called with the correct number of arguments.

17.1.2 Restrictions

Algebraic procedures in Reduce are treated as if they were symbolic, so that algebraic constructs actually appear as calls to symbolic functions, such as aeval.
RCREF adds the eval flag to some functions, so it probably cannot coexist with the compiler in a core image.
SYSLisp procedures are not correctly analyzed.

17.1.3 Usage

RCREF should be used in PSL:RLisp. To make a file FILE.CRF which is a cross reference listing for files FILE1.EX1 and FILE2.EX2 do the following in RLisp:
LOAD RCREF;
OUT "file.crf"; ON CREF;
IN "file1.ex1","file2.ex2";
OFF CREF;
SHUT "file.crf".
To process more files, more IN statements may be added, or the IN statement may be changed to include more files.

17.1.4 Options

***crefsummary** = [Initially:NIL] *switch*

If the switch CREFSUMMARY is ON then only the summary (see 1 above) is produced.
Functions with the flag NOLIST are not examined or output. Initially, all PSL functions are so flagged. (In fact, they are kept on a list NOLIST!*, so if you wish to see references to ALL functions, then CREF should be first loaded with the command LOAD RCREF, and this variable then set to NIL). (RCREF is now autoloading.)

nolist* = [Initially: the following list] *global*

(and cond list max min or plus prog prog2 progn times lambda abs add1 append apply assoc atom car cdr caar cadr cdar cddr caaar caadr cadar caaddr cadaar cadadr caddar cadddr cdaaar cdaadr cdadar cdaddr cddaar cddadr cdddar cddddr close codep compress cons constantp de deflist delete df difference digit divide dm eject eq eqn equal error errorset eval evlis expand explode expt fix fixp flag flagp float floatp fluid fluidp function densym get getd getv global globalp go greaterp idp intern length lessp linelength liter lposn map map mapc mapcan mapcar mapcon maplist max2 member memq minus minusp min2 mkvect nconc not null numberp onep open pagelength pair pairp plus2 posn princ print prin1 prin2 prog2 put putd putv quote quotient rds read readch remainder remd remflag remob remprop return reverse rplaca rplacd sassoc set setq stringp sublis subst sub1 terpri times2 unfluid upbv vectorp wrs zerop)
It should also be remembered that in RLisp any macros with the flag EXPAND or, if FORCE is on, without the flag NOEXPAND, are expanded before the definition is seen by the cross-reference program, so this flag can also be used to select those macros you require expanded and those you do not. The use of ON FORCE; is highly recommended for CREF.

17.2 Scanalyzer

17.2.1 Introduction

The scanalyzer module is a tool for analyzing PSL functions, expressions, and source files. It makes it easy to analyze PSL code and to write preprocessors for PSL. It supports preprocessing of PSL code in much the way

that macroexpand does, but is more flexible and extensible. The scanalyzer understands the various PSL function types, including macros, and has some understanding of each of the PSL special forms. Scanalyzer lives in SCANALYZER.SL, some support functions (including scanalyze-file) live in XREF-SUPPORT.SL.

Apologies to John Brunner, author of "Stand on Zanzibar", for the name of this facility.

17.2.2 Philosophy

The meaning of a piece of LISP code can depend a great deal on the context in which it is compiled or evaluated. Virtually everything in a LISP system can change dynamically, and analyzing the meaning of code in the right context is a significant problem.

This facility focuses on analysis of code that is compiled, and it does so for two reasons. One is that most large systems written in PSL run compiled. The other is that once LISP code is compiled, the meaning of the object code is much more static than the meaning of the corresponding interpretive code.

Reliable system operation depends on modules being compiled in an environment that is known and fixed as much as possible. To do this, we compile each module with a compiler that is started afresh. After one module has been compiled we do not compile other modules with the same compiler.

To precisely analyze a module, we can start a compiler. To it we add the analysis facility, which should not change the compilation context. We then analyze the module, just as we would compile it. The analysis facility must be sensitive to the same contextual information that the compiler is.

Many modules have some side-effects on the context of compilation – defining a macro is just one of the things that affects compilation context. To be sure that a module is analyzed in the right context, the analysis should have the same side-effects on compilation context as the compilation would have.

17.2.3 Functions

(scanalyze-file INPUT-FILE:string, pathname OUTPUT-FILE:string, pathname): string *expr*

Applies the function scanalyze-form (defined in section 2.3.3) to every form in input-file. Scanalyze-file assumes that information accumulates in the list xref-assertions. After all the forms have been analyzed, scanalyze-file dumps all the assertions into output-file as a list (reversing the list first). The assertion ($< input-file >$ IS-SOURCE-FILE) is added to the front of the assertion list.
File name defaulting: As its arguments scanalyze-file accepts both filename strings and pathnames. In any case the output-file argument is defaulted from the input-file argument. Any missing components of the name in output-file are defaulted using the function merge-pathname-defaults to be the same as in input-file, except for the file type (suffix). If that is not specified in output-file it is set to ".XD", which is the tentative standard for Xref-Databases.
For example, in the call (scanalyze-file "pk:load.sl" "mydir:"), the output filename generated is "mydir:load.xd". In the call (scanalyze-file "#5:/pslroot/foo/test.sl" "mydir:test-out.dat", the output filename generated is "mydir:test-out.dat".

(scanalyze-form FORM:form): form *expr*

This expects an s-expression as its argument and analyzes it as a top-level form in a source file. By recognizing which forms would be evaluated at compile-time, load-time, or at both times, this keeps the context for analysis the same as the context would be for compilation. Those forms that would be compiled are passed to the function Scanalyze.

Scanalyze-Form recognizes forms such as compiletime bothtimes, and loadtime. It evaluates each at compiletime if the compiler would and analyzes each if the compiler would compile it. Actually, it recognizes the IGNORE and EVAL flags used internally by the compiler. This means that if analysis is started in the same context as a compilation, during analysis macros are defined in the same way in both cases, as are wconsts, if systems expand the same way, the same "compiletime-loaded" modules are loaded, etc.. In PSL this means that the analysis context very closely matches the compilation context.

(scanalyze FORM:form ENV:form): form *expr*
Analyzes the given form (s-expression) in the given environment. The value returned depends on whether the caller has set up values for functional variables or properties that this function is sensitive to. If the caller has set up no special actions, the value returned should be the same that the function MACROEXPAND would return.

17.2.4 Environment Arguments

The caller can set up special actions that are in two categories. These are "analysis hooks" and "preprocessing hooks".
The ENV arguments to various things are environments. An environment is currently a list of "frames". Each frame is a list of 2 elements. The first identifies the type of frame, currently always 'LOCALS. The second is the "contents", for locals a list of the local variable names.

17.2.5 Analysis Hooks

The scanalyzer defines a number of variables that the user may give functional values to. Some support the user of this module in analyzing code. Others provide the means to preprocess or transform code. The hooks are functional variables of 2 args (expression and environment) to attach an analysis function to. Each of these must either be NIL or a list of functions. The first three of these hooks are called before any preprocessing or expansion of the form is done; the last is called after all preprocessing and expansion of a form is complete. Scan-Macro-Hooks Called before each macro is expanded. Scan-Fn-Hooks Called before descending into any expr to analyze it. Scan-Non-Pair-Hooks Called with each non-pair examined.
After-Expansion-Hooks Called with each fully expanded expression. Not applied to forms such as macro calls, because they are expanded further before being returned.

17.2.6 Properties

These are checked respectively before and after an expression has been expanded. After expansion means that the call has been expanded until it is not a macro call, and all subexpressions have been expanded also. If the car of an expression is an id that has the appropriate property, each member of the list that is the property value will be funcalled. These properties are checked at the same points where the scan-xxx hooks are checked. Thus it is of no use to give a macro a post-expand-scanners property. The properties are:

- Pre-Expand-Scanners
- Post-Expand-Scanners
Pre-expand-scanners take precedence over scan-macro-hooks and scan-fn-hooks.

17.2.7 Information Made Available By Scanalyzer

Current-Function This has the name of the current function where that can be determined. Forms such as DE and DEFUN that expand into calls on PUTD with constant function names cause this to be rebound inside analysis of the function body argument.
Current-Top-Level-Form Contains the argument that was passed to the call on scanalyze-form currently being executed.
Top-Level-Code? Is T for expressions not nested in any invocation of the function FUNCTION. Expressions for which this variable is T are known to be performed at initialization time.

17.2.8 Expansion And Preprocessing Hooks

Special-Expander Should be a functional variable of 2 arguments when set to a non-NIL value. If it is, and if the similar functional value of expand-specially? is non-NIL, the special-expander function will be used to expand the form.
Expand-Specially? If non-NIL, we use special-expander to expand the form.
Non-Pair-Expander Either NIL or a functional value of 2 arguments used to expand expressions that are "atomic".
If any function or macro, etc.has an Expr-Expander property, its value must be a 2 argument function that scans and expands expressions whose first element is that id. This module provides several such functions so that built-in special forms can be analyzed.
Special-Expander and Non-Pair-Expander have first priority. Then comes any Expr-Expander property. Then normal processing by type of function.

17.2.9 Cross Reference Support

The fluid xref-assertions holds the assertions generated by xref-assert and xref-assert-list, dumped out to a file by scanalyze-file.

(xref-assert A:assertion): assertion *expr*

(xref-assert-list L:list): assertion-list *expr*

These functions add the single assertion or list of assertions to xref-assertions.

(record-usage EXPR:form ENV:form): nil *expr*
Suitable as a function to be applied to each (non-atomic) expression and subexpression being analyzed. It is especially suitable as an after-expansion hook. Except for various special cases described below, record-usage makes an assertion for each expression that is a function call. The assertion is of the form, (<fn1> CALLS <fn2>) where <fn2> is the function being called (car of the expression). <Fn1> is the current-function, or if that is NIL, it is the current-file.

Record-usage ignores non-pair expressions. Expressions that are pairs (function calls) are checked for an openfn or opencode property, which would imply that they are compiled in-line. No assertion is made for these or for expressions where the function part (car) is not an id.
If the function being called has a usage-asserter property, the value of that property is funcalled to make whatever assertions it will. Otherwise if the function is flagged dont-record-usage, no assertion is made.

(record-macro-usage EXPR:form ENV:form): nil *expr*
record-macro-usage is like record-usage, but it is for macros and cmacros. This function is suitable as a scan-macro hook. The assertions generated by this function in the usual case are (<fn1> USES-MACRO <macro>) or for cmacros, (<fn1> USES-CMACRO <cmacro>).

(setup-some-xref-actions): t *expr*
The programmer who likes some of the facilities in this module, but doesn't want everything here can just load the module. If you want to use the facilities and defaults available in the module, call this function. In addition to setting up record-usage and record-macro-usage as analysis hooks, it flags some sets of functions as not having their usage recorded. It also sets up some special usage asserters for certain functions.
The value of each of these variables is a list of functions (or macros or both). The "setup" function, above, flags each function on each of these lists as dont- record-usage.

kernel-fns = [Initially: A list of all defined
functions in a PSL kernel.] *global*

useful-fns = [Initially: A list of frequently-used functions
and macros defined in the USEFUL library module.] *global*

object-fns = [Initially: A list of frequently-used functions
and macros in the OBJECTS module.] *global*

The following are special usage asserters for certain functions and macros. See the source code for details.

- Load-Usage-Asserter
- Imports-Usage-Asserter
- Send-Usage-Asserter
- Defmethod-Usage-Asserter

Prettyprinting

18.1 Introduction

To access the prettyprinter described in this chapter you must load the module PP.
A prettyprinter takes as input a stream of characters and prints them with aethetically appropriate indentations and line breaks. For example

```
(if (vectorp data) (vector-size data) (if (stringp data)
    (string-length data) (length data)))
```

```
(if (vectorp data)
  (vector-size data)
  (if (stringp data)
    (string-length data)
    (length data)))
```

18.2 Prettyprinting Files and Data

The width of an output line is defined by default to be 75. You can modify this by passing the desired width to the function line-width.

(pp-file SOURCE:string DESTINATION:string):
undefined *expr*

This function is used to prettyprint the contents of one file into another. There must be two arguments, the first is a string representing the source file name, the second is a string representing the destination file name. The first argument to top level applications of de, df, dm, ds, dn, defmethod, defflavor, and defmacro is printed once the form has been written to the file.

(pp-object E:form): undefined *fexpr*
This function will prettyprint the PSL data object E. Pp-object will also accept two optional arguments. The output may be directed to a file by including a string which represents the file name. Indentation is specified with an integer. Both of these optional arguments must be preceded by special identifiers, :indent for indentation, :sink for redirection of output.

18.3 Formats

Suppose we define the following data structure (in the discussion which follows it will be assumed that the selector functions form, root, and suffix have been defined).

```
(NAMED-FORM <form> <root> <suffix>)
```

Furthermore, we would like

```
(named-form (+ left right) argument 1)
```

to be printed in the form:

```
argument1: (+ left right)
```

The format definition which follows could be used to specify that named-forms should be printed out in the above way. Deformat will convert everything which follows its second argument into a formatting function. In the example below, this formatting function will be used to format lists which have the identifer named-form as their first element. When such a list is passed to it, the function creates a sequence of format instructions, specifying what should be printed corresponding to the list.

```
(deformat (named-form) (item)
  (template '({ * * ":" [i 1 2] * }) (root item)
  (suffix item) (form item)))
```

The function template creates a sequence of formatting instructions for its arguments based on directions specified by the template (the first argument). The template in this example can be understood as follows: The and specify that the components between them should be treated as a single logical unit when they are printed and that an outer pair of parentheses should be printed. The three *'s show where the three components of the data structure should be printed. The ":" specifies that a colon should be printed after

the suffix. The vector [I 1 2] corresponds to a conditional line break. If there is sufficient room for the form then a single space is printed (this is specified by the second element of the vector). Otherwise a line break followed by an indentation of two spaces is inserted.

A template may consist of the following symbols:

- The prettyprinter will break onto different lines as few logical blocks as possible. The left angle bracket (<),denotes the beginning of a block, the right angle bracket (>), denotes the end of a block. It is assumed that these brackets balance.
- Left and right braces (and) are also to define logical blocks which should be printed within a pair of parentheses.
- There are two different types of blanks, the terms consistent and inconsistent are used to describe them. Note that blanks may only appear within a logical block. If each blank within a block is consistent and the block will not fit on the current line of output then each sub-block of the block will be placed on a new line, if each blank is inconsistent then new lines will be forced only when it is necessary. For example, the first result will occur if the blanks are inconsistent, the second if they are consistent.

```
(one two three
 four)

(one
 two
 three
 four)
```

In general if a block will not fit on the current line of output then each consistent blank corresponds to a new line, inconsistent blanks will print as spaces unless it is necessary to force a new line. There is a length and an offset associated with each blank, if a blank is treated as a space then the length represents the number of spaces which will be printed. If a blank is treated as a break to a new line then the offset is used for indentation on the new line. Indentation is from the horizontal position of the first character of the innermost logical block. For example,

```
(template '({ * { (* [c 1 1]) } })
          '(function (arg-one arg-two)))
```

will result in

```
(function (arg-one
           arg-two))
```

assuming that it is necessary to force a line break. The consistent blank within the call the template specifies an offset of one. The first character of the innermost logical block is the second left parenthesis, from its position the printer indented a single space. A blank is represented by a vector, the first entry should be either an I or C, these characters designate the type of the blank, I for inconsistent and C for consistent. The second entry is a positive integer which represents length, and the third is a positive integer representing indentation. The algorithm expects to find a blank after each occurence of a right angle bracket, if the formats which have been provided do not follow this convention then an inconsistent blank with length and offset set to zero will be inserted after the right angle bracket.

- An asterisk (*) is used to make a recursive call on dispatch. Dispatch is responsible for applying the appropriate formatting function to an expression
- Strings, the text of the string is included in the output.
- A sublist of format symbols may also appear within the list of format symbols. This list may contain blanks or asterisks and is used repetitively until a corresponding list of expressions is exhausted.

18.4 Dispatch

```
(de dispatch (item depth)
  (cond ((instance item)
         (apply (find-instance-template item)
                (list item depth)))
        ((atom item)
         (apply (find-atom-template item)
                (list item depth)))
        ((> depth *maxdepth*) (add-tokens "#"))
        ((has-template item)
         (apply (get-template item)
                (list item (add1 depth))))
        (t (default-template item (add1 depth)))))))
```

It is useful to know how this function associates a format function with an expression. If the expression is an instance of a flavor then the dispatch function will search for a format function associated with the name of the

flavor. For other atomic expressions, the type is used to locate a format function. Currently there are format functions for vectors, strings, numbers, code-pointers and identifiers. If the expression is a list and the first element of this list is an identifier then an attempt is made to find a format function associated with the identifier. Before resorting to a default a check is made for function application for which there is a separate default format function. Function application is recognized when the first element of the list is a lambda expression, an application of getd to the first element returns a non-nil value, or the first element is an id which is a member of a global list named *function_names*. This list of function names is useful for prettyprinting a file since a file will normally contain a number of applications of functions defined within the file itself. The fexpr add-functions accepts any number of function names and adds then to *function_names*.
With respect to lists, any sub-expression below a maximum depth is printed as `'#'`. Once the maximum length of a list has been reached the display of the remaining elements will be '...'. The maximum depth (initially 10) and length (initially 25) can be modified with the functions new-depth and new-length.

18.5 Specifying Formats

(deformat NAME:pair PARAM:id-list [BODY:forms]): nil *fexpr*
Deformat is used to define format functions for PSL expressions. The car of NAME should be a list of identifiers (it may also be a single identifier), the cdr should be nil for atomic expressions or an integer representing a minimal length for lists. PARAM is a list containing a single id, when the format function is used PARAM will be bound to the expression being formatted. Formats are defined by calls on template (described below).

(template FORMAT:list [COMPONENT:form]): any *macro*
FORMAT is a list of format symbols (described above). Each COMPONENT should correspond to a call on dispatch (*), in FORMAT. If a sub-list of format symbols in included within FORMAT then there must be a corresponding list of components.

This first example illustrates a format for quoted expressions. For example, 'ANY instead of (QUOTE ANY). A minimal length of two is given, since the list (QUOTE) should not be printed as '. The argument to quote is formatted by a separate call on dispatch.

```
(deformat (quote . 2) (item)
  (template '("'" *) (second item)))
```

The elements of a vector should be enclosed within square brackets. Since the length of vectors will vary a list of format symbols is supplied to be used repetitively for each element. This list will be matched against a corresponding list of components, (vector2list item). Each element is formatted by a recursive call on dispatch. Elements are separated by inconsistent blanks, therefore if the entire vector does not fit on a single line the ouput will be something like

```
[one two three
 four]

(deformat (vector) (item)
  (template '(< "[" (* [i 1 1]) "]" >) (vector2list item)))
```

Given the expression (setq n-one v-one n-two v-two)

```
(setq n-one v-one
      n-two v-two)
```

is preferred over

```
(setq n-one v-one n-two
 v-two)
```

The following can be used to obtain the preferred output.

```
(deformat ((set setq) . 3) (item)
  (template '({ * " " < (* [i 1 1] * [c 1 0]) > })
            (first item) (rest item)))
```

The first form of ouput is obtained by using inconsistent blanks after the names (n-one and n-two) and consistent blanks after the values (v-one and v-two). When this expression will not fit on a single line the consistent blanks will correspond to line breaks and unless a value is too large to fit on an output line, the inconsistent blanks will correspond to spaces. The zero offset associated with the consistent blanks corresponds to the block of names and values. This is why the names line up within a column. If the inner logical brackets were omitted the output would look something like

```
(setq n-one v-one
 n-two v-two)
```

assuming that the expression would not fit on a single line. Notice that the format for a single pair (name and value) is used repetitively.
A format function may contain more than one call on template. The two functions below are equivalent.

```
(deformat (lambda . 3) (item)
  (template '({ * [i 1 1]) (first item))
  (if (null (second item))
    (template '("()" [c 1 1] (* [c 1 1]) })
              (rest (rest item)))
    (template '((* [c 1 1]) }) (rest item))))

(deformat (lambda . 3) (item)
  (template '({ * [i 1 1] (* [c 1 1]) })
            (first item)
            (cons (if (null (second item)) '!(!) (second item))
                  (rest (rest item)))))
```

The Objects Module

19.1 Introduction

The **objects** module provides simple support for object-oriented programming in PSL. It is based on the "flavors" facility of the LISP machine, which is the source of its terminology. The LISP Machine Manual contains a much longer introduction to the idea of object oriented programming, generic operations, and the flavors facility in particular. This discussion briefly covers the basics of using objects to give you an idea of what is involved; then it explains the details.

19.1.1 Terminology

An object datatype is known as a flavor. It can be thought of in two parts:

1. A template that defines the characteristics of the flavor.

2. A set of operations that can be performed on the flavor.

The template that defines the characteristics of a flavor is known as the flavor definition. It is created by the macro defflavor. Each operation that can be performed on an object is known as a method definition. They are created by the macro defmethod. Invoking an operation on an object is known as sending an object a message. A created object of a given flavor is known as an instance of that flavor.
Example of a flavor definition is:

```
(defflavor complex-number
           (real-part
            (imaginary-part 0.0)
            )
           ()
           gettable-instance-variables
           initable-instance-variables
           )
```

A flavor definition specifies the fields, or in our terminology, the instance variables, that each object of that flavor is to have. In the example above, the instance variables are REAL-PART and IMAGINARY-PART. The mention of the instance variable IMAGINARY-PART indicated that by default the imaginary part of a complex number will be initialized to 0.0. There is no default initialization for REAL-PART.
Instance variables may be strictly part of the implementation of a flavor, totally invisible to users. Typically though, some of the instance variables are directly visible in some way to the user of the object. The flavor definition may specify "initable-instance-variables", "gettable-instance-variables", and "settable-instance-variables". These options mean that the instance variables specified are able to be initialized (initable), able to be accessed by name (gettable), and able to be assigned and accessed (settable). None, some of, or all of the instance variables may be specified in each option. In the example above, both REAL-PART and IMAGINARY- PART may be initialized and may be accessed by name.
With the objects package the programmer completely controls what operations are to be done on objects of each flavor, so this is a true object-oriented programming facility. Also, operations on flavored objects are generic. This means that the operations can be done on an object of any flavor, as long as the operations are defined for that flavor of object. The same operation can be defined for many flavors; and whenever the operation is invoked, what is actually done will depend on the flavor of the object it is being done to.
To see the power of generic operations, consider the following example:
Say we wish to write a scanner that reads a sequence of characters out of some object and processes them. It does not need to assume that the characters are coming from a file, or even from an I/O channel.
Suppose the scanner gets a character by invoking the GET-CHARACTER operation. In this case any object of a flavor with a GET-CHARACTER operation can be passed to the scanner, and the GET-CHARACTER operation defined for that object's flavor will performed to fetch the character. This means that the scanner can get characters from a string, or from a text editor's buffer, or from any object at all that provides a GET-CHARACTER operation. The scanner is automatically general.

19.2 Creating Objects

The function make-instance provides a convenient way to create objects of any flavor. The flavor of the object to be created and the initializations to be done are given as parameters in a way that is fully independent of the internal representation of the object.

19.2.1 Methods

The function "=>", whose name is intended to suggest the sending of a message to an object, is usually used to invoke a method.
Examples:

```
(=> my-object zap)
(=> thing1 set-location 2.0 3.4)
```

The first argument to => is the object being operated on: my-object and thing1 in the examples. The second argument is the name of the method to be invoked: zap and set-location. The method name is not evaluated. Any further arguments become arguments to the method, and are evaluated. (There is a function send which is just like => except that the method name argument is evaluated just like everything else.)
Once an object is created, all operations on it are performed by methods defined for its flavor. The flavor definition also defines some methods. For each "gettable" instance variable, a method of the same name is defined which returns the value of that instance variable. For "settable" instance variables a method named "set-<variable name>" is defined. Given a new value, this method sets the instance variable to have that value.

19.2.2 Protection of Objects

Most LISP's, and PSL in particular, leave open the possibility for the user to perform illicit operations on LISP objects. Instances of a flavor are represented as ordinary LISP objects (evectors at present), so in a sense it is quite easy to do illicit operations on them: just operate directly on its representation (do evector operations).
On the other hand, there are major practical pitfalls in doing this. The representation of a flavor of objects is generated automatically, and there is no guarantee that a particular flavor definition will result in a particular representation of the objects. There is also no guarantee that the representation of a flavor will remain the same over time. It is likely that at some point evectors will no longer even be used as the representation.
In addition, using the objects package is quite convenient, so the temptation to operate on the underlying representation is reduced. For debugging, one can even define a couple of extra methods "on the fly" if need be.

19.3 Reference Information

19.3.1 Loading Objects

Note: This file defines both macros and ordinary LISP functions. It must be loaded before any of these functions are used. The recommended way of doing this is to put the expression: (BothTimes (load objects)) at the beginning of your source file. This will cause the package to be loaded at both compile and load time.

19.3.2 Flavor Definition

Flavors are defined by using the macro defflavor This macro has the form:

(defflavor flavor-name (var1 var2 ...) (inheritance-flavor-list) option1 option2 ...):list *macro*

flavor-name is the name of the flavor being defined. var1 var2 ... are the instance variables of the flavor. Each must be a symbol that is the name of the variable, or a list of two elements. The list form must have the name of the instance variable (id) as its first element and a default initialization form as its second argument. Do not use names like "IF" or "WHILE" for instance variables: they are translated freely within method bodies (see defmethod). The translation process is not very smart. For example, it doesn't distinguish between formal parameters to methods with the same name as instance variables. For safety, if formal parameters are found that match instance variables, an error message is issued. Although the translation process isn't smart, it at least understands the nature of quote.
The inheritance-flavor-list must be a list of one element or the empty list. When the empty list, it specifies that no inheritance is used in defining this flavor (for a description of inheritance, see section 10.3). When a one element list, the element must be the name of another flavor to inherit from. This other flavor must have already been compiled or loaded or an error message will be issued.
option1 option2 ... are the options that effect the instance variables and methods of the flavor. Recognized options are:

```
(INITABLE-INSTANCE-VARIABLES var1 var2 ...)
(GETTABLE-INSTANCE-VARIABLES var1 var2 ...)
(SETTABLE-INSTANCE-VARIABLES var1 var2 ...)
(FAST-ACCESS-FOR-METHODS method1 method2 ...)
```

INITABLE-INSTANCE-VARIABLES	[make all instance variables INITABLE]
GETTABLE-INSTANCE-VARIABLES	[make all instance variables

	GETTABLE]
SETTABLE-INSTANCE-VARIABLES	[make all instance variables SETTABLE]
FAST-ACCESS-FOR-METHODS	[access all methods fast]

An empty list of variables is taken as meaning all variables rather than none, so (GETTABLE-INSTANCE-VARIABLES) is equivalent to GETTABLE-INSTANCE-VARIABLES.
Initable instance variables may be initialized using options to make-instance or instantiate-flavor. See below.
For each gettable instance variable a method of the same name is generated to access the instance variable. If instance variable LOCATION is gettable, one can invoke (=> <object> LOCATION).
For each settable instance variable a method with the name SET-<name> is generated. If instance variable LOCATION is settable, one can invoke (=> <object> SET-LOCATION <expression>). Settable instance variables are always also gettable and initable by implication. If this feature is not desired, define a method such as SET-LOCATION directly rather than declaring the instance variable to be settable.
FAST-ACCESS-FOR-METHODS is used to speed up message sending for a select set of methods. Methods specified in this option are accessed by a different means then "normal" methods. This option should only be used when:

- It is known that message sending speed is critical.
- A select set of methods of a flavor are known to be called very frequently compared to the rest of the methods of that flavor.

The order methods are specified with this option determines the order they are accessed (speed of method lookup). Thus, the first method specified will be accessed the quickest and the last method specified the slowest. If no methods are given (the second form above), the objects package will decide on its own order. Specifying no methods should be used only if a flavor has a few methods (i.e., less then 10). If any of the methods specified are not methods of the flavor, a warning message will be issued and fast access for it will be ignored.
As an example, consider the flavor TUNAFISH which has 70 methods. It is determined that much time is spent calling five low level TUNAFISH methods whose names are A, B, C, D, and E. Specifying:

```
(FAST-ACCESS-FOR-METHODS A B C D E)
```

in the flavor definition of TUNAFISH will cause method A to be accessed faster then any other method, B to be accessed slower then A, C slower then B, D slower then C, and finally E the slowest of the five methods. The remaining 65 methods of TUNAFISH will be accessed as slow or slower then E.
As its value, defflavor returns a list of the form:

```
(flavor <flavor-name>)
```

Where <flavor-name> is the name of the flavor being defined. Examples of Flavor Definition

```
(defflavor complex-number
          (real-part imaginary-part)
          ()
          gettable-instance-variables
          initable-instance-variables
          )

(defflavor complex-number
          ((real-part 0.0)
           (imaginary-part 0.0)
           )
          (number)
          gettable-instance-variables
          (settable-instance-variables real-part)
          )
```

19.3.3 Method Definition

As you may recall, methods are defined using the macro defmethod. It has the form:

```
(defmethod (flavor-name method-name) ([arg1 arg2 ...])
<expression1> <expression2>...): list                    macro
```

flavor-name is the name of the flavor a method is begin defined for. It must be an id.
method-name is the name of this method. It must also be an id.
arg1 arg2 ... are the arguments of the method. There may be zero or more and they must all be ids.
<expression1> <expression2>... make up the body of the method. This body can refer to any instance variable of the flavor by using the name just like an ordinary variable. They can also set them using setf. All occurrences

of instance variables (except within evectors or quoted lists) are translated to an invocation of the form (EGETV SELF n).
The body of a method can also freely use SELF like it were another instance variable. SELF is bound to the object that the method applies to. SELF may not be setqed or setfed.
The value returned by defmethod is a list of the form:

```
(method <flavor-name> <method-name>)
```

Where <flavor-name> and <method-name> are simply the names of the flavor being defined and method being defined respectively.

Examples of Method Definition:

```
(defmethod (complex-number real-part) () real-part)

(defmethod (complex-number set-real-part) (new-real-part)
(setf real-part new-real-part))

(defmethod (toaster plug-into) (socket)
(setf plugged-into socket)
(=> socket assert-as-plugged-in self))
```

19.3.4 Object Creation

The most common way instances of a flavor are created is by using the function make-instance. It has the form:

(make-instance flavor-name:id [initialization-sequence]):
flavored object *expr*

make-instance takes as arguments a flavor name and an optional initialization sequence.
Flavor-name must be an id that represents an existing flavor.
The optional initialization-sequence, when specified, consists of alternating pairs of instance variable names and corresponding initial values. Note that the instance variable names are quoted, in the example above, because all the arguments are evaluated. make-instance returns an object that is "initialized."

Examples of make-instance

```
(make-instance 'complex-number)
(make-instance 'complex-number 'real-part 0.0
                        'imaginary-part 1.0)
```

An object returned by make-instance is initialized as follows:
First, all instance variables with initialization specified in the call to make-instance are initialized to the value given. After this the objects DEFAULT-INIT method is envoked. This method is constructed by the flavor definition (defflavor). It initializes any instance variables not specified in the call to make-instance but having default initializations specified in the flavor definition. It then sets all remaining uninitialized instance variables to the value *UNBOUND*. The default initialization and initialization to *UNBOUND* is performed by the DEFAULT-INIT method.
If a method named INIT is defined for this flavor of object, it is invoked automatically after the initializations just discussed. The INIT method is passed, as its one argument, a list of alternating variable names and initial values. This list is the result of evaluating the initializations given to make-instance. For example, if we call:

```
(make-instance 'complex-number 'real-part (sin 30)
                               'imaginary-part (cos 30))
```

then the argument to the INIT method (if any) would be

```
(real-part .5 imaginary-part .866).
```

The INIT method may do anything desired to set up the desired initial state of the object.
At present, this value passed to the INIT method is of virtually no use to the INIT method since the values have been stored into the instance variables already. In the future, though, the objects package may be extended to permit keywords other than names of instance variables to be in the initialization part of calls to make-instance. If this is done, INIT methods will be able to use the information by scanning the argument.
For details on how initialization is done when inheritance is in effect, see Using Inheritance, section 10.3.
Another less common way to create an object is through the use of the function instantiate-flavor. It has the form:

(instantiate-flavor flavor name:any [U:list]):
flavored object *expr*

This is the same as make-instance, except that the initialization list is provided as a single, required argument. The list must be a list of alternating keyword names and initial values.

Example:

```
(instantiate-flavor 'complex-number
        (list  'real-part  (sin  30) 'imaginary-part (cos 30)))
```

19.3.5 Message Sending

There are several ways to invoke a method:

(=> object:object method:id): any *expr*

Examples:

```
(=> r real-part)

(=> r set-real-part 1.0)
```

The message name is not quoted. Arguments to the method are supplied as arguments to =>. In these examples, r is the object, real-part and set-real-part are the methods, and 1.0 is the argument to the set-real-part method.

(send object:object method:id): any *expr*

Examples:

```
(send r 'real-part)

(send r 'set-real-part 1.0)
```

The meanings of these two examples are the same as the meanings of the previous two. Only the syntax is different: the message name is quoted.

(send-if-handles object:object method:id [U:any]): any *expr*
Conditionally Send a Message

Examples:

```
(send-if-handles r 'real-part)

(send-if-handles r 'set-real-part 1.0)
```

SEND-IF-HANDLES is like SEND, except that if the object defines no method to handle the message, no error is reported and NIL is returned.

(lexpr-send object:object method:id [U:any] V:list):any *expr*
Send a message (explicit "Rest" Argument List)

Examples:

```
(lexpr-send foo 'bar a b c (list x y))
```

The last argument to LEXPR-SEND is a list of the remaining arguments.

(lexpr-send-if-handles object:any
method:id [U:any] V:list):any *expr*

This is the same as LEXPR-SEND, except that no error is reported if the object fails to handle the message.

(lexpr-send-1 object:object method:id V:list):any *expr*

Send a Message (Explicit Argument List)
Examples:

```
(lexpr-send-1 r 'real-part nil)
```

```
(lexpr-send-1 r 'set-real-part (list 1.0))
```

Note that the message name is quoted and that the argument list is passed as a single argument to LEXPR-SEND-1.

(lexpr-send-1-if-handles object:any
method:any V:list): any *expr*

This is the same as LEXPR-SEND-1, except that no error is reported if the object fails to handle the message.

(ev-send object:any method:any V:list): any *expr*

EV-SEND is just like LEXPR-SEND-1, except that it is an EXPR instead of a MACRO. Its sole purpose is to be used as a run-time function object, for example, as a function argument to a function.

19.3.6 Printing Objects

Objects can print out according to their flavor. If a flavor has a channelprin method, that method will be used by prin1, prin2, etc. to print objects of that flavor.

(=> any CHANNELPRIN channel level prin1?): any *method*
A channelprin method must accept three arguments: a channel (currently a small integer); either nil or a number which will indicate the current level of nesting within lists, vectors, or other objects; and a boolean which will indicate whether prin1- or prin2-style printing is being done.

Ordinarily, an object should print in the form #<flavor info ... >. If the level of printing is not nil, printing of lists, vectors, objects, etc. within the channelprin method should be done with prinlevel bound to 1 less than its value at the time of entry to the method.

19.3.7 Useful Functions on Objects

(object-type U:any): id, NIL *expr*
The object-type function returns the type (an id) of the specified object, or nil, if the argument is not an object. At present this function cannot be guaranteed to distinguish between objects created by the objects package and other LISP entities, but the only possible confusion is with vectors or evectors.

(flavor-defined? flavor-name:id): t, nil *expr*
Returns t if flavor-name is a currently defined flavor. Returns nil otherwise.

(declare-flavor $< flavorname >< var1 >< var2 >$**...): undefined** *macro*

(undeclare-flavor $< var1 >< var2 >$**...): undefined** *macro*
`*** Read these warnings carefully! ***`
This facility can reduce the overhead of invoking methods on particular variables, but it should be used sparingly. It is not well integrated with the rest of the language. At some point a proper declaration facility is expected and then it will be possible to make declarations about objects, integers, vectors, etc., all in a uniform and clean way.

The declare-flavor macro allows you to declare that a specific symbol is bound to an object of a specific flavor. This allows the flavors implementation to

eliminate the run-time method lookup normally associated with sending a message to that variable, which can result in an appreciable improvement in execution speed. This feature is motivated solely by efficiency considerations and should be used ONLY where the performance improvement is critical.
Details: if you declare the variable X to be bound to an object of flavor FOO, then **within the context of the declaration** (see below), expressions of the form (=> X GORP ...) or (SEND X 'GORP ...) will be replaced by function invocations of the form (FOO GORP X ...). Note that there is no check made that the flavor FOO actually contains a method GORP.If it does not, then a run-time error "Invocation of undefined function FOO"GORP" will be reported.
WARNING: The declare-flavor feature is not presently well integrated with the compiler. Currently, the declare-flavor macro may be used only as a top-level form, like the PSL FLUID declaration. It takes effect for all code evaluated or compiled henceforth. Thus, if you should later compile a different file in the same compiler, the declaration will still be in effect! **this is a dangerous crock, so be careful!** To avoid problems, I recommend that declare-flavor be used only for uniquely-named variables. The effect of a declare-flavor can be undone by an undeclare-flavor, which also may be used only as a top-level form. Therefore, it is good practice to bracket your code in the source file with a declare-flavor and a corresponding undeclare-flavor.

```
*** Did you read the above warnings??? ***
```

(object-get-handler object:object method-name:id): id *expr*

Note that the use of object-get-handler is not recommended. It should only be used in the most critical places where the time to perform a normal message send is too slow and the declare-flavor facility cannot be used.
Given an object and a method-name, object-get-handler returns the name of the function that implements this method. It is used to repetitively send the same message. For example:

```
(let
    ((func-name (object-get-handler obj 'banana)))
...
    (for
    ...
    (do (idapply func-name (list arg1 arg2 ... argn)))
    )
)
```

This is equivalent to sending the message:

```
(=> obj banana arg1 arg2 ... argn)
```

inside the for but the lookup of the function name for the message send is only performed once.

Miscellaneous Functions

(objects-version): NIL *expr*
This function prints a date used to check the version of the objects package in use. When bugs are reported, the version date should be given. Older versions of the objects package did not have this function defined.

19.4 Using Inheritance

Inheritance is a mechanism that allows a flavor to be defined in terms of other flavors. This allows objects to be created that are specializations or generalizations of other objects. The object package supports what is called single inheritance – only one flavor can be inherited from by another flavor. The flavor being inherited, however, could have inherited from one other flavor, and so on. Note that the LISP machine supports multiple inheritance allowing more than one flavor to be inherited from at a time.

The best way to explain single inheritance is through an example:
Suppose we have defined the flavor mother and then define the flavor daughter in which we specified the inheritance-flavor-list to be mother. The inheriting process causes daughter to contain all the instance variables as well as the methods that were defined for mother. That is,a daughter object can deal with any of the instance variables and receive any of the messages defined for flavor mother without having to recreate them.
We refer to flavor mother as the parent of flavor daughter, and daughter as a child of mother. daughter, in turn, can be the parent of other flavors. Each child inherits the instance variables and methods of its parent, its parent's parent, and so on.
When inheriting instance variables, a child flavor has as its instance variables the union of all those defined for its parents and those specifically defined for the new flavor. As a result, it would not make sense to give a child flavor an instance variable with the same name as one in one of its parent's. For this reason, duplicate instance variables are flagged as an error.

Inheriting methods, however, is different. It is possible to define a method in a child flavor with the same name as a method in one of the parents. The effect is to override the parent's method with the child's method, as far as the child flavor is concerned. In other words, when sending a message to an object, the method executed is the first one found by searching backward from it through its parents. First the methods defined for the child flavor are searched, then the child's parent's methods, and so on until one is found.
As an example of overriding methods, the following code defines a flavor geometric-object. Then, a new flavor square is created with geometric-object as its parent. Finally, a flavor colored-square is created with square as its parent.

```
(defflavor geometric-object
  (name
   size)
  ()                    % A flavor with no parents
  )

(defmethod (geometric-object display) ()
  (printf "%w is %w units big" name size))

(defmethod (geometric-object set-name) (new-name)
  (setf name new-name))

(defflavor square
  (side-length)
  (geometric-object)    % A flavor with one parent, geometric-object
  )

% Overriding for square the inherited method display

(defmethod (square display) ()
  (printf "%w is a square, %w units long on all sides" name
          side-length))

(defmethod (square new-length) (length)
  (setf side-length length)
  (setf size (* length length)))

(defflavor colored-square
  (color)
  (square)
  )
```

```
(defmethod (colored-square get-color) ()
  color)
```

After the definitions, flavor colored-square has the instance variables:

```
color
side-length
name
size
```

and has the methods:

- get-color (as defined for flavor colored-square)
- new-length (as defined for flavor square)
- display (as defined for flavor square)
- set-name (as defined for flavor geometric-object)

19.4.1 Warning on Inheritance Usage

The Common lisp objects package will be incompatible with the existing objects package as far as inheritance is concerned. In Common lisp, an inherited instance variable cannot be accessed by simply referring to it within a method. Also, the way variables are initialized is quite different.

To access inherited instance variables, you must do the equivalent of sending the object a message to get the instance variable's value. For example, referring to the instance variable banana in a method for the flavor fruit, a reference to banana in PSL becomes something like (=> self banana) in Common lisp. When you reference inherited instance variables in PSL and performance is important, you should clearly mark them. If performance is not an issue, make the instance variables gettable and refer to them by their method. This also applies to settable instance variables.

Uses of the INIT method may also have to be changed because the new objects package performs object initialization in a different way. Avoid the INIT method if you can, if you can't mark it for change for Common lisp objects.

In this way, transition to Common lisp will be much easier.

19.4.2 Using SELF and MYSELF with Inheritance

Suppose an additional method was defined for flavor geometric-object:

```
(defmethod (geometric-object display-me) ()
(=> self display))
```

Now suppose a display-me message is sent to an object of type square. Which method for the message 'display' should be used by display-me? Should it look up the method in the context of the flavor where the method was defined (geometric-object) or the context of the flavor that received the original message (square)? The answer is to provide a new symbol, MYSELF. MYSELF means to look up messages in the context of the object that received the original message. SELF looks up methods in the context of the flavor it is defined in. If display-me was rewritten as:

```
(defmethod (geometric-object display-me) ()
  (=> self display)
  (=> myself display))
```

and sent to an object of flavor square, then the first display would be that defined in geometric-object, and the second would be the one defined for square. If the display-me message was sent to an object of flavor geometric-object, self and myself would be identical.

19.4.3 Inheritance and Initialization

You may have noticed that there is ambiguity in the order a newly created object is initialized when the object uses inheritance. When inheritance is used, initialization is performed as describe above with the following additions:
When the son flavor's (i.e., the flavor given in the make-instance or instantiate-flavor call) DEFAULT-INIT method is executed, the first thing it does is call its father's DEFAULT-INIT method. The second thing it does is execute its father's INIT method, if it exists.
Notice that if the father flavor used inheritance, its DEFAULT-INIT method would first call its grandfather's DEFAULT-INIT and INIT methods, and so on.

19.4.4 Making Changes to Inherited Code

The inheritance scheme used in this facility is static – all of the work is done at compile-time. This has some implications for compiling object definitions that use inheritance:

- All of the parents of a flavor must be loaded or compiled before compiling the new child's definitions. If this is not the case, an error message will be issued.

- If a change is made to a parent flavor, including the addition of a new method or changing instance variables, all of the children will have to be recompiled to use it. You do not have to recompile the children, however, if an existing method of a parent is redefined.

19.5 Debugging Information

Any object may be displayed symbolically by invoking the method DESCRIBE, e.g. (=> x describe). This method prints the name of each instance variable and its value, using the ordinary LISP printing routines. Flavored objects are liable to be complex and deeply nested or even circular. This makes it often a good idea to set PRINLEVEL to a small integer before printing structures containing objects to control the amount of output.
When printed by the standard LISP printing routines, "flavored objects" appear as evectors whose zeroth element is the name of the flavor.
For each method defined, there is a corresponding LISP function named `<flavor-name>$<method-name>`. Such function names show up in backtrace printouts.
It is permissible to define new methods on the fly for debugging purposes.

Vectors and Such

20.1 Vectors

A vector is a structured entity in which random elements may be accessed with an integer index. A vector has a single dimension. Its maximum size is determined by the implementation and available space. A vector is denoted by enclosing its elements within square brackets.

```
[10 TEN]
[COLORS (RED BLUE)]
```

Built-in Vector Creation and Copying Functions

(mkvect UPLIM:integer): vector *expr*

Defines and allocates space for a vector with (add1 UPLIM) elements accessed as 0 ... UPLIM. Each element is initialized to nil. If UPLIM is -1, an empty vector is returned. An error occurs if UPLIM is less than -1 or if the amount of available memory is insufficient for a vector of this size.

```
***** A vector of size UPLIM cannot be allocated
```

(make-vector UPLIM:integer INITVAL:any): vector *expr*

Similar to mkvect, except that each element is initialized to INITVAL. Note the difference between this function and make-string, (see the section on creating and copying strings in Chapter 6). This function creates a vector of (add1 UPLIM) elements where make-string creates a string of UPLIM characters.

The vector created by this function will contain (add1 UPLIM) references to INTVAL as opposed to creating a copy of UPLIM for each entry.

```
1 lisp> (setq array (make-vector 1 (make-vector 1 0)))
```

```
[[0 0] [0 0]]
2 lisp> (vector-store (vector-fetch array 0) 0 1)
1
3 lisp> array
[[1 0] [1 0]]
```

(vector [ARGS:any]): vector *nexpr*

Create vector of elements from the list ARGS. The size of the vector will be equal to the number of elements in the list ARGS. Each element of the vector is initialized to the corresponding element from ARGS

(copyvectortofrom NEW:vector OLD:vector): NEW:vector *expr*

The elements of NEW are set to the corresponding elements of OLD The elements are not copied.

```
1 lisp> (setq a [[1 2 3]])
[[1 2 3]]
2 lisp> (setq b [0])
[0]
3 lisp> (copyvectortofrom b a)
[[1 2 3]]
4 lisp> (eq (getv a 0) (getv b 0))
t
```

(copyvector V:vector): vector *expr*

Create a new vector, with the elements initalized from the corresponding elements of V. The elements of V are not copied.

```
1 lisp> (setq a "A STRING")
"A STRING"
2 lisp> (setq b (vector a))
["A STRING"]
3 lisp> (setq c (copyvector b))
["A STRING"]
4 lisp> (eq (getv b 0) (getv c 0))
t
```

20.1.1 About the Basic Operations on Vectors

The functionality provided here overlaps what is provided in some other ways. The functions provided here have well-chosen names and definitions, they provide the option of generating efficient code, and they are consistent with the esthetic preferences of our community.

20.1.2 The Operations

This section documents functions in the library module **slow-vectors** . There is another library module called **fast-vectors**. The fast-vectors module provides alternate definitions for these functions. When the switch fast-vectors is non-nil the compiler will use these alternate definitions to produce effiecient code. However, there will not be any verification that arguments are of correct type (in addition, it is assumed that numeric arguments are within a proper range). If invalid arguments are used, then at best your code will not generate correct results, you may actually damage the PSL system. There are two side effects to loading fast-vectors. The slow-vectors module will be loaded and the switch fast-vectors will be set to t.

(vector-fetch V:vector I:integer): any *expr*
Accesses an element of a PSL vector. Vector indexes start with 0. The thing stored in that position of the vector is returned.

(vector-store V:vector I:integer X:any): None Returned *expr*
Stores into a PSL vector. Vector indexes start with 0.

(vector-size V:vector): integer *expr*
Returns the number of elements in a PSL vector. Since indexes start with index 0, the size is one larger than the greatest legal index. See also just below.

(vector-upper-bound V:vector): integer *expr*
Returns the greatest legal index for accessing or storing into a PSL vector. See also just above.

(vector-empty? V:vector): boolean *expr*
True if the vector has no elements (its size is 0), otherwise NIL.

20.1.3 Built-in Operations on Vectors

These predate the fast-vectors (f-vector) and slow-vectors (s-vector) library modules.

(getv V:vector INDEX:integer): any *expr*
Returns the value stored at position INDEX of the vector V. The type mismatch error may occur. An error occurs if the INDEX does not lie within 0 ... (upbv V) inclusive:

```
***** INDEX subscript is out of range
```

(putv V:vector INDEX:integer VALUE:any): any *expr*
Stores VALUE in the vector V at position INDEX, VALUE is returned. A type mismatch error will occur if V is not a vector. If INDEX is either negative or greater than (upbv V) then an error occurs.

```
***** Subscript 'INDEX' in PutV is out of range
```

(upbv U:any): nil, integer *expr*
Returns the upper limit of U if U is a vector, or nil if it is not.

20.2 Word Vectors

Word-vectors or w-vectors are vector-like structures, in which each element is a "word" sized, untagged entity. This can be thought of as a special case of fixnum vector, in which the tags have been removed.

(make-words UPLIM:integer INITVAL:integer): word-vector *expr*
Defines and allocates space for a Word-Vector with (add1 UPLIM) elements, each initialized to INITVAL.

(make-bytes UPLIM:integer INITVAL:integer): byte-vector *expr*
Defines and allocates space for a byte-vector with (add1 UPLIM) elements, each initialized to INITVAL.

20.3 General X-Vector Operations

An x-vector is either a vector, string, word-vector, or byte-vector. Each may have several elements, accessed by an integer index. A valid index for an x-vector X is from 0 to (size X). Thus an x-vector X will have (add1 (size X)) elements. The functions described in this section may also be applied to lists.

(size X:x-vector): integer *expr*
Returns the size of x-vector X, the size is the index of the last element.

(indx X:x-vector I:integer): any *expr*
Access the I'th element of an x-vector. An error occurs if I is either negative or exceeds the size of X.

```
***** Index 'I' out of range for X in INDX
```

(setindx X:x-vector I:integer A:any): any *expr*
Define A to be the I'th element of X. If the index I is outside the range of X then it is an error (see indx for a description of the message).

(sub X:x-vector B:integer S:integer): x-vector *expr*
Extract a subrange of an x-vector, starting at B, producing a new x-vector of size S. Note that an x-vector of size 0 has one entry.

(setsub X:x-vector I1:integer S:integer Y:x-vector): x-vector *expr*
Store subrange of Y of size S into X starting at I1. Returns Y.

(subseq X:x-vector LO:integer HI:integer): x-vector *expr*
Returns an x-vector whose size is (sub1 (- HI LO)), beginning with the element of X with index LO. In other words, returns the subsequence of X starting at LO and ending just before HI.

```
1 lisp> (setq a '[0 1 2 3 4 5 6])
[0 1 2 3 4 5 6]
2 lisp> (subseq a 4 6)
[4 5]
```

(setsubseq Y:x-vector):Y:x-vector *expr*

The size of Y must be (sub1 (- HI LO)) and Y must be the same type of x-vector as X. Elements LO through (sub1 HI) in X are replaced by the elements of Y. Y is returned and X is changed destructively.

```
1 lisp> (setq a "0123456")
"0123456"
2 lisp> (setsubseq a 3 7 "ABCD")
"ABCD"
3 lisp> A
"012ABCD"
```

(concat X:x-vector Y:x-vector): x-vector *expr*

Concatenate 2 x-vectors. Currently they must be of same type.

(totalcopy S:any): any *expr*

Returns a unique copy of the entire structure, i.e., it copies everything for which storage is allocated - everything but inums and ids. Like copy (Chapter 5) totalcopy will not terminate when applied to circular structures.

```
1 lisp> (setq x '("ONE" 2)
1 lisp>       y (totalcopy x)
1 lisp>       z (copy x))
("ONE" 2)
2 lisp> (eq (first x) (first y))
NIL
3 lisp> (eq (first x) (first z))
T
```

Operating System Contacts

21.1 Calling the Command Shell

The function system can be used to execute a system command.

(system COMMAND:string):undefined *expr*
starts a (system specific) command interpreter and passes the command to the interpreter.

E.g. unter the UNIX operating system a Bourne shell is started and COMMAND is interpreted following the conventions of this shell. Of course it is possible to use e.g.

```
(system "bash")
```

21.2 The Working Directory

The current working directory can be read by

(pwd):STRING *expr*
returns the current working directory in system specific format.

The current working directory can be set by

(cd DIR:string):BOOLEAN *expr*
sets the current working directory to DIR after expanding the filename according to the rules of the operating system. If this operation is not sucessful, the value Nil is returned.

21.3 Invoking Pipes

Pipes are known to almost all the operating systems where PSL is implemented for. It usually comes in two flavours, named pipes and anonymous pipes. PSL uses the anonymous pipes for unidirectional communication. One standard operation is the connection to the Gnuplot graphics system from the REDUCE computer algebra system. In the following we describe the pipes operation in PSL under Unix and Windows.

21.3.1 Pipes under Unix

The pipe interface can be made available by loading the module **pipes**.

(pipe-open CMD:string MODE:id):Channel *expr*

The Unix system tries to create the process defined by CMD using the usual Unix mechanisms like pathes etc. Depending on the MODE either the stdin or the stdout is connected to the PSL program during the evaluation. The resulting LISP I/O file is returned such that in subsequent read or write operation to this file the command input or output is used. MODE needs to be either `input` or `output`.
A pipe can be used as a normal LISP file. A close operation on the file will terminate the process. If the user wants to keep the process alive, instead of close the following function has to be used:

(abandonpipe Channel):undefined *expr*
This closes the pipe created by pipe-open, but the process will remain active.
Examples:

```
(load pipes)
(setq mail (pipe-open "mail hugo@zib.de" 'output))
(wrs mail)
(prin2 "This is the result of (expt 10 100) ")
(prin2t (expt 10 100))
(wrs nil)
(close mail)

(load pipes)
(setq unix (pipe-open "uname -s" 'input))
(rds unix)
```

```
(setq variant (read))
(rds nil)
(close unix)
```

21.3.2 Pipes under MS/DOS

The pipe interface can be made available by loading the module **pipes**.

21.3.3 Pipes under MS Windows

The pipe interface can be made available by loading the module **w-pipes**.

21.4 Socket Interface (Unix only)

The socket interface allows the PSL system to communiate through Unix socket as if they were normal LISP I/O channels. This can be used to write server or client processes in PSL easily. The module `pslsocket` has be be loaded in advance.

(socketopen HOST:(string or 0) SNumB:integer): pair of LISP channels *expr*

If HOST is 0 (the server mode) the PSL system waits until a BIND request is received on the socket SNumB. Otherwise HOST is interpreted as a string (the client mode) and a computer with this name is sought on the net. If the name could be resolved successfully, a BIND request is send to this computer with the socketnumber SNumB. If the operation worked o.k., a pair is returned consisting of two LISP channels, the CAR for input and the CDR for output, each of these can be used like LISP channels furtheron.

(socketflushbuffer C:channel):any *expr*

Sends the characters hanging around in the buffer via the socket associated with the LISP channel.

21.5 Shared Memory Interface (Unix only)

The PSL shared memory interface provides all function for operating with shared memory regions and semaphores (See e.g. Unix man pages for shmop, shmget, shmctl, semop, semctl, semget etc.) The definitions of these man

pages are used in the paragraph. Using the memory address map mechanism described below,it is easy to write one's own shared memory application.
In the rest of this paragraph we describe a simple model implementation of a 'pipe' using shared memory and a semaphore. This code is contained in the file $pu/shmem.sl.

(shm-open S:pair M:Mode):any *expr*
If S = 0, a new shared memory area is allocated. Otherwise S is expected to be a dotted pair of shmid and semid of an existing shared memory. Legal modes are **input_create, output_create, input and output**. A list consisting of channelnumber shmid and semid is returned.

(independentdetachshm C:channel): any *expr*
Detaches the shared memory region used by C, closes the channel.

(readfromshm C:channel):any *expr*
Waits until the shared memory region is readable, reads an expression and resets the mode to writable. Returns the expression.

(writetoshm C:channel E:expression):list *expr*
Waits until the shared memory region is writeable, prints the expression using prin2. Returns the value of prin2.

The following C program works together with the PSL part such that it prints the messages read from shared memory when they are ready for printing. It must be started with two paramemters, namely the shmid and the semid to synchronize with the PSL part.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

struct sembuf ;

struct sembuf sembu;
struct sembuf *sembuptr;
```

```
main (argc,argv)
     int argc;
     char *argv[];

{ int sema , shmemid;
  char * shmaddress;

  sembuptr = &sembu;

  sscanf(argv[1],"%d",&shmemid);
  sscanf(argv[2],"%d",&sema);

  /* open shared memory */

 printf("the data is : %d %d\n",shmemid,sema);
  shmaddress = shmat(shmemid,0,0);

  while (1)
  { waitsema(sema) ; /* wait for a 0 */
    printf("the message is : %s \n",shmaddress +4);
    setsema (sema,2) ; /* ok, eaten */
  }
}

setsema (sema,val)
int sema,val;
{ semctl(sema,0,SETVAL,val); }

waitsema (sema)

int sema;
{
  sembu.sem_num =0; sembu.sem_op = 0; sembu.sem_flg =0;

  semop (sema, sembuptr , 1);
}
```

21.6 Miscellaneous Features

It is a design goal to reduce the number of system calls which have to be linked in to the kernel to a minimum. The necessary information can be read

elsewhere, e.g. through a pipe. For some example, see the chapter on pipes above.
Nevertheless, some system calls have be be linked in, because they handle process specific information which is not available to a spawned process.
The follwoing two system calls can be used to create unique file names.

(getpid):Integer *expr*
Returns the system specific number under which the PSL process is registered in the currently running system. If PSL is running under the control of a GUI, this number corresponds to the PSL process, not the GUI process,

(gethostid):any *expr*
The system and hardware dependent word (say), which identifies the processor is returned. It is normally a good idea to store this word in a place where the garbage collector will not come along, e.g. in a WARRAY.

Calling Programs in Foreign Languages

to be done

Utilities for Profiling

To be added.

PSL Command Line

Memory Management

eventuell zu MISC

Addendum

Since its earlier versions starting in the 80's, Portable Standard LISP has undergone several changes.

PSL was developed for mainframe systems e.g. of /370 and Vax type running proprietary operating systems of the manufacturers. Today most PSL versions run on workstations with RISC processors running a Unix type operating system.

Today PSL is mostly used a deliverly vehicle for LISP based systems, e.g. the computer algebra system REDUCE and not as a LISP dialect of its own right.

The support and maintainance for PSL is mostly done at ZIB in Berlin.

Recently, the internal representation of identifiers was changed in order to support a 256 character set and case sensitivity. This causes a binary incomptibility of .b files with older versions such as PSL 3.4.

This document describes the changes and newer developments for PSL done at ZIB in order to improve performance and to work with new application types such as parallel processing.

26.1 Installation

This section gives instructions for the Installation of PSL 4.2 on your UNIX system. It does not apply for other operating systems such as VMS, MVS, DOS etc, and it does not apply for distribution on diskettes (e.g. all PC systems, IBM RS/6000, NeXT). Please read special instructions (below) for the system mentioned above.

The Installation procedure described below uses `csh` syntax. If you don't have csh installed on your system, please change the commands accordingly.

PSL is stored as a single file tree with two directories on the top level, called bin and dist, where the later contains all sources and some more binary files.

The installation is relatively simple.

26.1.1 Reading the tape

The tape contains a compressed file tree. Please cd to the directory where you want to store the files (called PSL root directory later on) and enter

```
dd if=/dev/tape ibs=51200 | uncompress | tar -xf -
```

Please replace /dev/tape in the above command with the correct name of your tape drive.

26.1.2 Reading the tape for IBM RS/6000

The PSL file tree for the IBM RS/6000 is configured in the same way as for other Unix type workstations. The installation needs an additional step.
The installation procedure described below assumes that you use the C-Shell. If you use different shells please change commands accordingly.
To read the tape, make a directory with a name like .../psl, cd to that directory, and do, after putting the tape into the drive,

```
tar -xf /dev/rfd0
```

You first need to install a syscall into the AIX system. **In order to do this, YOU NEED SUPER USER PRIVILEGE.** To install the syscall, do

```
su
cd ./dist/kernel/ibmrs/syscall
./install
./installsysc
exit
```

The script install compiles and installs the required system call. The script installsysc in the same directory installs the system call only. With any reboot of the system it is mandatory to run installsysc in the directory ./dist/kernel/ibmrs/syscall. It is suggested that this is included into the AIX boot files. The sources of the syscall can be found in ./dist/kernel/ibmrs/syscall too.

26.1.3 Reading Diskettes for LINUX 386

The PSL file tree for LINUX is compressed on to 3 diskettes for distribution. The 3 diskettes require about 4 megabytes of disk space to dump. However, you must then rebuild the binary files for your machine before you can do anything. Instructions on doing this are given below. If the files for your machine are built, the file tree then requires about 30 megabytes during the build (but less when complete).

To read the diskettes, make a directory with a name like .../psl, cd to that directory, and do, after putting DISK # 1 into the floppy disk reader,

```
tar -xf /dev/fd0H1440
install_psl
```

The script install_psl prompts for the other disks and will uncompress the files.
If there is an incompatibility between the delivered exeutable and your Linux version, or when you upgrade your Linux version and PSL becomes inoperable, you must recompile the executable *bpsl*:

```
cd $pxk/linux
cclnk
```

After this step you should rebuild the image files using the scripts in `$psl/distrib`.

26.1.4 Customizing Makefiles and scripts

In the next step the system must get knowledge about the PSL root directory and the machine architecture. The correct name for the architecture in the PSL file tree can be found in the table below.

Machine	Operating system	PSL MACHINE name	
CDC 4xxx	EP/IX	mips_cdc	++
Convex Cxxxx		convex	++
Convex SPP		convex_spp	
Cray 1, X-MP	UNICOS	crayxmp	
Cray Y-MP	UNICOS	crayymp	
Cray C90	UNICOS	crayymp	
Cray T3D/E	UNICOS	cray_t3d	
DECStation	ULTRIX	Mips_dec	
DEC Alpha	OSF/1 (DEC Unix)	Alpha	
DEC Alpha	Linux	Alpha_Linux	
DEC VAX	ULTRIX	vax	
DG AViiON	DG-UX	88k-aviion	++
HP9000/300 400	HP-UX	bobcat	++
HP9000/700	HP-UX	Snake	
HP9000/800	HP-UX	spectrum	++
IBM RS/6000	Aix	Ibmrs	
IBM SP2	(the RS/6000 version happens to work)		
Intel 386	Linux	Linux	

Intel 386 ELF	Linux (ELF)	Linux_elf	
Intel x86	Solaris 2.x	Solarisx86	
SGI Iris/Indigo	Irix	Mips_iris	
SGI Iris/Indigo	Irix64	Mips_iris_64	
Sun 3	SunOS	sun	++
Sparc	SunOS (Solaris1.x)	Sun4	
Sparc	Solaris 2.x	Solaris	
UltraSparc	Solaris 2.x	Ultrasparc	

++ This version may be not supported in future version.

The script dist/distrib/newroot.csh will modify various Makefiles and scripts, especially dist/psl-names. It must be started with the PSL root directory and MACHINE name as parameter, e.g. for a Sparc under SUNOS:

```
dist/distrib/newroot.csh ‘pwd‘ Sun4
```

After that the file dist/psl-names contains the settings for the PSL specific environment variables such as $psys. It must be read in by the PSL user to insure correct operation,e.g. for a Sparc machine under SunOS:

```
setenv MACHINE Sun4
source <PSL root directory>/dist/psl-names
```

This installation is designed for usage with multiple machine types, it saves disk space by sharing the PSL code (.sl files). The variable **MACHINE** is used to specify the machine type. If you install a single version of PSL it may be useful to replace **MACHINE** by the correct value in the file $< PSLrootdirectory >$/dist/psl-names. In this case the user does not need to specify the MACHINE environment variable.
To test the installation, you can try the following commands which are supposed to run without error message:

```
$psys/psl
(load inum)
(quit)
```

If you change files, e.g. when you receive a bug fix, please put the file into the correct place , cd to the directory $psl and simply say `make`. This will compile the file (and maybe some files which depend on this), and produce new binaries.

26.1.5 Printing Documentation

The documentation for PSL (User's Manual and some more documents) can be found in the directories dist/doc and dist/lpt and dist/manual (in LaTeX or plain ASCII print format)

26.2 New unexec procedure, Image model

PSL's former unexec model (Savesystem) was to produce executables in the sense of the operating system (mostly a.out files). This has led into maintainance problems with various new file formats (e.g. ELF, COFF) and dynamic loading. Influenced by the model developed for the PSL on Personal Computers, for all new PSL versions for new architectures the so called **image model** is used. This simply means that the command `savesystem` does not write an executable file but it puts all the LISP data into a file such that the information can be reloaded afterwards. The reload is done by the program `$pxk/bpsl` when the -f parameter is used. The dumpfiles names get the suffix `.img`, e.g. the command

```
(savesystem "Banner" "dir/file" initform)
```

produces a file `dir/file.img` which can be restarted by issueing the command

```
$pxk/bpsl -f dir/file.img
```

The LISP data written to the file are: `SYMVAL, SYMPRP, SYMFNC, SYMNAM and SYMGET, hashtable` and the allocated parts of `bps and heap.` This reduces the amount of disk storage used drastically.
To insure proper reloading, the program bpsl can be used for reloading of images only if these were written from its own core image. A check for this is provided using a datetag scheme which must be the same in bpsl and image file. The error message **Cant start image with this bpsl** means that the check was not successful and you have to produce a new image file. This normally happens when the file bpsl is rebuild.
In very rare cases after dramatic changes in the environment (generally speaking), you may get the message **Cant relocate the image**. In this case, please try to rebuild the images. If this fails (it should not), please try

```
$pxk/bpsl -f dir/file.img -g 1000000
```

If this works properly, you may try to reduce the g value. In any case please send a note the the author of this document.

26.3 Dynamic configuration of Heap Size and Binding Stack

The PSL workspaces `Heap and Bindingstack` can be enlarged at run time. This is often necessary for programs that allocate a lot dynamic storage (heap) or use variable binding intensively.
If heap space gets low, the garbage collector tries to increase heap size automatically. This is a reaction to an emergency situation though and it is tried after many garbage collections. Moreover, depending on the history of the PSL session, the system may find itself unable to allocate the requested memory causing an LISP error condition. A manual increase of heap size is much better for overall performance.
The Binding stack is not enlarged automatically, because in case of on infinite loop the PSL memory would blow up and it would take a very long time until the user would see a problem (if at all).
Two functions have been added to PSL for handling heap and binding stack size:

`(set-heap-size N:Integer or NIL): Integer or NIL` *expr*

> If called with NIL it returns the number of LISP items in heap. If called with a number (which is counted in items and should be bigger that the current size in items) PSL tries to allocate the memory. If successful, the argument is returned, if not, NIL is returned.

`(set-bndstk-size N:Integer or NIL): Integer or NIL` *expr*

> If called with NIL it returns the number of LISP items in bndstk. If called with a number (which is counted in items and should be bigger that the current size in items) PSL tries to allocate the memory. If successful, the argument is returned, if not, NIL is returned.

26.4 Size of Address Space

The tradition PSL model is `high tagged`, i.e. a tag which describes the data type is stored in the high bits of a LISP item. In most cases the number of bits used for tagging is 5, which limits the address space to bitsperword - 5 bits. This has not been a problem for a long time for the widely used 32 bit microprocessor architectures, since the swap space available was much less that this number. E.g on a Sparc system you can use about 100 MB LISP workspace, about 7000000 LISP items.
For the time being the 64 bit microprocessors do not suffer from this limitation. A `low tagged` version of PSL is under development already since 1990.

26.5 Arbitrary Precision Integer Support

The efficient support of Arbitrary Precision Integers ("Bignums") is crucial for a computer algebra system. This version of PSL tries to exploit the given machine hardware as far as possible, e.g. using double size multiplication and division in hardware where available. Because of the different processor architectures the optimal internal representation of bignums may vary between workstations even with same number of bits per word.
The PSL manual mentions the modules "big" and "nbig" which are written in the true spirit of portable software but these are unfortunately not very efficient. We recommend to use the load module **zbig** instead. In all versions this load module contains the bignum version which we have found to be optimal.

26.6 Monitoring of Performance

26.6.1 SPY (Unix only)

The Unix profil system call provides a facility to get information about the cpu consumption in a LISP program. A simple interface was build to be able to turn profiling on and off and to map the results on to the LISP address space. For details, please refer to the source code in `$pxu/spy.sl`.

`(spywhole N:Integer): void` *expr*

Turns on profiling for the whole bps space. N must be a power of 2, defining the grain for the profil call. Typical values are 4 and 8. Returns `nil`.

`(spyprint): void` *expr*

Turns off the profiler. Prints profiler information after mapping it to the LISP address space, i.e the names of LISP functions are printed together with their relative cpu consumption. The variable &spyminimum& defines the cutoff for spyprint, i.e. functions with less than &spyminimum& ticks will not be printed. Returns `nil`.

`(spyon fwa lwa tslice bucketsize power): void` *expr*

This is a complete interface for the profil call. Allocates Warray space for the Unix profiler and turns on profiling. fwa and lwa are the addresses of the first and last word of code memory which should be inspected, tslice is the timeslice and not suported on most Unix systems. It is a parameter for compatibility reasons. Bucketsize must be a power of 2 and power is the base 2 logarithm of bucketsize. Typical values are 4 or 8 for bucketsize and 2 or 3 for power resp. Returns `nil`.

`(spyoff): void` *expr*

Turns off profiling using the Unix profil system call. Returns `nil`.

Example:

```
(load spy zbig)

NIL

(spywhole 4)

NIL

(null (setq aa(expt 3000 10000)))

NIL

(spyprint)  % This was done on a SUN4 PSL

214       39.5%     BPLUSA2
204       37.7%     INITCODE
67        12.3%     WTIMESDOUBLE
15        2.7%      COPYFROMONESTATICHEAP
12        2.2%      XXCOPYFROMSTACK
12        2.2%      SET_HEAP_SIZE
NIL
```

It is easy to see that the command `(null (setq aa(expt 3000 10000)))` leads to a lot of bignum computations. The spyprint output proves that. Please note that `INITCODE` sums up all the functions that belong to PSL's C runtime support. In this case the high percentage is due to the fact that the (old) SPARC implementation performs integer multiply and divide in software i.e. in non LISP code.

26.6.2 Qualified timing

Qualified counting is used to measure the cpu time spend between call and return from a function, not necessarily spend within the body of the function. It gives an overview about the costs that a call of particular function causes. The sum shown as result may be bigger that 100 percent, if a function and its callee are both measured or if gc time is accumulated.

`(qualtime S:list): void` *macro*

Starts qualified timing for the list of functions contained in list. Returns `nil`.

`(print!-qualtime):` `void` *expr*

Prints the accumulated timings and number of calls for all functions. Resets counters. Returns `nil`.

Example:

```
(load qualified-timing zbig)
(off usermode)

(qualtime gtpos times2)

 (null (setq aa(expt 3000 10000))) % this will be measured

(print-qualtime)
 *********** Qualified Timing **************
 **** Overall Cpu time :  5083  *****

 *** TIMES2                    * calls : 18        * time : 5797  * % 114
 *** GTPOS                     * calls : 5560      * time : 935   * % 18
```

26.6.3 Qualified counting

For our own applications it had turned out to be useful to know the callers of a function especially when the function is called very, very often, e.g. basic routines like generic arithmetic. To provide this a module qualified counting was generated using a similar syntax like qualified timing.

`(qualcount S:list):` `void` *macro*

Starts qualified counting for the list of functions contained in list. Returns `nil`.

`(print!-qualcount):` `void` *expr*

Prints the accumulated call counting and the callers for all functions. Resets counters. Printing will be suppressed if the number of calls is less than bordervalue (default = 20) Returns `nil`.

`(reset!-qualcount):` `void` *expr*

Resets the accumulated call counting and the callers for all functions. Returns `nil`.

Example:

```
 (load qualified-counting)
 (off usermode)

 (qualcount faslin intern)
*** Function 'G0003' has been redefined
*** Function 'FASLIN' has been redefined
*** Function 'G0025' has been redefined
*** Function 'INTERN' has been redefined

 (load nbig30) % this will be counted

(print-qualcount)
************* calls for function FASLIN *************

************* calls for function INTERN *************

number of calls : 306 from READ-ID-TABLE
```

26.7 Compiler Modifications

In order to optimize the code generation for the RISC processors which (in most cases) have a concept of delayed branches a special optimization phase for the PSL compiler was build to fill the delay slots. This new compiler phase is called **lapopt**. It collects several optimizations which work on the generated stream of instructions (the lap code). This optimization phase is enabled by default, it is controled by the switch **lapopt**, and it can be turned off and on by

```
(on lapopt)    and  (off lapopt)
```

A sample of optimized code can be found in the following section on disassemble.

26.8 Disassembler

For most implementations of PSL, a disassembler for compiled functions is available which gives insight into the 'real' code. It is a tool for code optimization only and not meant to give insight for the 'normal' LISP user. Example:

```
(load disassemble)
NIL
(disassemble 'get)
( ** Function  GET  at 16#36110)

00036110    9DE3BFA0    save       r14,-x'60,r14
00036114    3B3E0000    sethi      -x'20000,r29
00036118    21360002    sethi      -x'9FFFE,r16            Alloc 2
0003611C    A2100002    or         r0,r2,r17               (frame 1)
00036120    9930601B    srl        r1,x'1B,r12
00036124    80A3201E    subcc      r12,x'1E,r0
00036128    32800007    bne,a      L0003
0003612C    82100006    or         r0,r6,r1
00036130    9930A01B    srl        r2,x'1B,r12
00036134    80A3201E    subcc      r12,x'1E,r0
00036138    22800005    be,a       L0004
0003613C    88100006    or         r0,r6,r4
00036140    82100006    or         r0,r6,r1
                    L0003:
00036144    81C7E008    jmpl       r31+8,r0
00036148    81E82000    restore    r0,0,r0

              ... etc ...
```

26.9 More unsupported software

A few parts of the PSL system or applications are no longer useful or cant be supported.

26.9.1 Oload not supported

We discontinue the support of OLOAD, which allows on some versions to link in external routines into PSL at run time using the system loader. As alternative we recommend the usage of shared memory (see below). An example how to use an shared memory based oload substitute is under development.

26.9.2 Portable Common Lisp Subset (PCLS) not supported

If you want Common LISP , please use Common LISP.

26.10 Shared Memory Interface (Unix only)

The PSL shared memory interface provides all function for operating with shared memory regions and semaphores (See e.g. Unix man pages for shmop,shmget,shmctl,semop,se etc.) The definitions of these man pages are used in the paragraph. Using the memory address map mechanism described below,it is easy to write one's own shared memory application.
In the rest of this paragraph we describe a simple model implementation of a 'pipe' using shared memory and a semaphore. This code is contained in the file $pu/shmem.sl.

`(shm!-open S:pair M:Mode): any` *expr*

> If S = 0 a new shared memory area is allocated. Otherwise S is expected to be a dotted pair of shmid and semid of an existing shared memory. Legal modes are **input_create, output_create, input and output**. A list consisting of channelnumber shmid and semid is returned.

`(independentdetachshm C:channel): any` *expr*

> Detaches the shared memory region used by C, closes the channel.

`(readfromshm C:channel): any` *expr*

> Waits until the shared memory region is readable, reads an expression and resets the mode to writable. Returns the expression.

`(writetoshm C:channel E:expression): list` *expr*

> Waits until the shared memory region is writeable, prints the expression using prin2. Returns the value of prin2.

The following C program works together with the PSL part such that it prints the messages read from shared memory when they are ready for printing. It must be started with two paramemters, namely the shmid and the semid to synchronize with the PSL part.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

struct sembuf ;

struct sembuf sembu;
```

```
struct sembuf *sembuptr;

main (argc,argv)
     int argc;
     char *argv[];

{ int sema , shmemid;
  char * shmaddress;

  sembuptr = &sembu;

  sscanf(argv[1],"%d",&shmemid);
  sscanf(argv[2],"%d",&sema);

  /* open shared memory */

 printf("the data is : %d %d\n",shmemid,sema);
  shmaddress = shmat(shmemid,0,0);

  while (1)
  { waitsema(sema) ; /* wait for a 0 */
    printf("the message is : %s \n",shmaddress +4);
    setsema (sema,2) ; /* ok, eaten */
  }
}

setsema (sema,val)
int sema,val;
{ semctl(sema,0,SETVAL,val); }

waitsema (sema)

int sema;
{
  sembu.sem_num =0; sembu.sem_op = 0; sembu.sem_flg =0;

  semop (sema, sembuptr , 1);
}
```

26.11 Socket interface (Unix only)

to be added....

26.12 Pipe Interface (Unix only)

The Pipe interface establishes a pipe as a PSL channel from which (or to which) normal PSL read/write operations can be used. Pipes are specially useful to supply data to other processes, e.g. REDUCE uses it to send data to a plot program.

`(pipe!-open S:string M:Mode): integer` *expr*

> Starts a process and executes the command in ¡string¿. The mode must be either Input or Output. Returns a channel.

Pipes are closed with a normal close call. This will kill the process that has been started. If this process should survive, you have to use abandonpipe.

`(abandonpipe channel): void` *expr*

> Closes channel without killing the process started by pipe!-open. Returns `nil`.

Example:

```
(load pipes)
(setq chn (pipe!-open "uname"))
(channelread chn)
(close chn)
```

26.13 Mapping of LISP Addresses to C addresses

This paragraph applies for some architectures only, e.g. the `IBM/RS, HP PA-Risc and MIPS` where the system architecture uses 'high' bits in the addresses which conflict with the PSL tagging mechanism.(See the paragraph on the Size of the Address space above). The consequence of this is that using the inf operation for calculating the C equivalent from a LISP address does not work here. The easiest way (without explaining the reason) of mapping for the architectures is:

IBM/RS	(mkstr item)
HP PA-Risc	(mkvec item)
Mips	(wshift (inf item) 2)
Linux_elf	(mkfixn item)

On Convex systems the 'high' bit is the leftmost one. Therefore the tag bits start at bit 1 and the inf range is limited by 26 bits. The inf operation works as expected.

Index

<, 18
<<, 20
<=, 18
>, 18
>=, 18
>>, 20
∗, 19
|, 20
∼, 19
∼=, 19
∧, 20
*backtrace (switch), 176
*break (switch), 177
*comp (switch), 110
*compressing, 149
*compressing (switch), 152
*crefsummary (switch), 213
*echo (switch), 155
*emsgp (fluid), 176
*eolinstringok, 6
*eolinstringok (switch), 151
*fast-evectors (switch), 201
*fast-integers, 17
*fast-integers (switch), 202
*fast-strings (switch), 201
*fast-vectors (switch), 201
*gc (switch), 210
*in, 137
*nolinke (switch), 203
*ord (switch), 203
*out, 137
*output (switch), 169
*pcmac (switch), 203
*pgwd (switch), 203
*plap (switch), 203
*printloadnames, 154
*printloadnames (switch), 155
*pwrds (switch), 203
*r2i (switch), 202
*raise (switch), 151
*redefmsg, 3, 4
*redefmsg (switch), 109
*time (switch), 169
*usermode, 3, 48
*usermode (switch), 109
*verboseload, 154
*verboseload (switch), 155
+, 19
–, 19
/, 19
//, 19
/=, 18
=, 18
=>, 237, 239
&, 20

abandonpipe, 254
abs, 20
aconc, 56
acos, 29
acosd, 29
acot, 29
acotd, 30
acsc, 30
acscd, 30
add1, 20
adjoin, 57
adjoinq, 57
allocate-string, 70
alphanumericp, 68
alphap, 68

and, 2, 12
ans, 170
append, 56
apply, 132
asec, 30
asecd, 30
asin, 28
asind, 28
ass, 63
assoc, 62
atan, 29
atan2, 29
atan2d, 29
atand, 29
atom, 11
atsoc, 63

backquote, 114
binaryclose, 164
binaryopenappend, 164
binaryopenread, 164
binaryopenupdate, 164
binaryopenwrite, 164
binarypositionfile, 165
binaryread, 164
binaryreadblock, 165
binarywrite, 165
binarywriteblock, 165
bldmsg, 158
bothcasep, 68
bothtimes, 201
br, 184
break, 183
breakin* (global), 158, 179
breaklevel* (global), 177
breakout* (global), 158, 179

c, 150
car, 50
case, 84
catch, 102
catch-all, 104
cd, 253
cdr, 51
ceiling, 25
channeleject, 145
channellinelength, 146
channellposn, 146
channelposn, 145
channelprin1, 141
channelprin2, 141
channelprin2t, 144
channelprinc, 144
channelprint, 141
channelprintf, 143
channelread, 147
channelreadch, 149
channelreadchar, 148
channelreadline, 150
channelreadtoken, 149
channelreadtokenwithhooks, 150
channelspaces, 142
channeltab, 142
channelterpri, 142
channelunreadchar, 149
channelwritechar, 144
char, 65
char<, 69
char>, 69
char-bits, 69
char-code, 69
char-downcase, 70
char-equal, 68
char-font, 69
char-greaterp, 69
char-int, 70
char-lessp, 69
char-upcase, 70
char=, 68
character, 69
close, 153
code-address-to-symbol, 189
code-char, 69
code-number-of-arguments, 117
codeapply, 133

codeevalapply, 134
codep, 11
commentoutcode, 201
compd, 193
compile, 194
compile-file, 192
compile-file-aux, 192
compiletime, 201
compress, 159
concat, 252
cond, 81
cons, 50
const, 199
constant?, 198
constantp, 11
conterror, 174
continuableerror, 174
copy, 51
copyd, 110
copyscantable, 163
copystring, 71
copystringtofrom, 71
copyvector, 248
copyvectortofrom, 248
cos, 27
cosd, 27
cot, 28
cotd, 28
csc, 28
cscd, 28
currentreadmacroindicator* (global), 152
currentscantable* (global), 152

date, 209
date-and-time, 209
de, 111
declare-flavor, 239
decr, 20
defconst, 199
defflavor, 232
define-constant, 198
deflambda, 113
deflist, 38
defmacro, 112
deformat, 225
degreestodms, 27
degreestoradians, 26
del, 59
delasc, 60
delascip, 60
delatq, 60
delatqip, 60
delete, 59
deletip, 60
delq, 60
delqip, 60
desetq, 41
df, 111
difference, 21
digit-char, 70
digitp, 68
divide, 21
dm, 111
dmstodegrees, 27
dmstoradians, 27
dn, 111
do, 100
do*, 101
do-loop, 102
do-loop*, 102
ds, 112
dskin, 156
dumplisp, 208

eject, 145
emsg* (global), 176
eolinstringok, 6
eq, 9
eqcar, 11
eqn, 9
eqstr, 11
equal, 10
error, 173
errorform* (global), 177
errorprintf, 145

errorset, 176
errout* (global), 158
errprin, 144
errset, 175
ev-send, 238
eval, 48, 130
evlis, 133
evprogn, 134
exit, 89
exitlisp, 207
exitwithstatus, 207
exp, 30
expand, 135
explode, 159
explode2, 159
exprp, 117
expt, 21

factorial, 31
faslin, 157
fatalerror, 175
fboundp, 116
fcodep, 116
fexprp, 117
filep, 152
findprefix, 37
findsuffix, 37
first, 53
fix, 17
fixp, 11
flag, 40
flag1, 40
flagp, 40
flambdalinkp, 116
flatsize, 158
flatsize2, 159
flavor-defined?, 239
float, 17
floatp, 11
floor, 25
fluid, 125
fluidp, 126
for, 90
for*, 97
foreach, 98
fourth, 53
free-bps, 209
funboundp, 116
funcall, 133
function, 135
function-basic-definition, 121
function-lambda-list, 121

garbage collection, 210
gcknt* (global), 210
gctime* (global), 210
gensym, 35
geq, 23
get, 38
getd, 110
getfcodepointer, 117
gethostid, 258
getpid, 258
getv, 250
getwords, 190
global, 125
globalp, 126
gmergesort, 62
go, 88
graphicp, 67
greaterp, 23
gsort, 61

hist, 169
historycount* (global), 170
historylist* (global), 170

id2int, 14
id2string, 14
idapply, 133
idp, 11
idsort, 62
if, 82, 83
if_system, 204
ignore, 48
ignoredinbacktrace* (global), 178
implode, 159

imports, 154
in* (global), 157
incr, 21
independentdetachshm, 256
indx, 251
init file, 208
init-file-string, 208
initforms* (global), 169
inp, 170
input–case, 34
instantiate-flavor, 236
int-char, 70
int2id, 14
intern, 13
interngensym):, 35
internp, 36
interpreterfunctions* (global), 178
intersection, 58
intersectionq, 58

kernel-fns (global), 218
known-free-space, 209

land, 24
lapin, 156
lastcar, 54
lastpair, 54
lbind1, 126
lconc, 56
length, 55
leq, 23
lessp, 23
let, 85
let*, 85
lexpr-send, 237
lexpr-send-1, 238
lexpr-send-1-if-handles, 238
lexpr-send-if-handles, 238
linelength, 146
lispbanner* (global), 208
lispscantable* (global), 162
list, 55
list2set, 58
list2setq, 59
list2string, 15
list2vector, 16
lnot, 24
load, 153
loaddirectories* (global), 154
loadextensions* (global), 155
loadtime, 201
log, 30
log10, 31
log2, 30
lor, 24
lose, 3, 48
lowercasep, 68
lposn, 145
lshift, 25
lxor, 25

macroexpand, 116
macrop, 118
main, 171
make-bytes, 250
make-instance, 235
make-string, 70
make-vector, 247
make-words, 250
makefcode, 117
makeflambdalink, 117
makefunbound, 117
makeunbound, 45
map, 99
mapc, 99
mapcan, 99
mapcar, 99
mapcon, 100
maplist, 100
mapobl, 37
max, 23
max2, 23
maxbreaklevel* (global), 177
member, 55
memq, 55
min, 23

min2, 24
minus, 21
minusp, 24
mkquote, 134
mkstring, 71
mkvect, 247
mkwords, 190
mod, 26

nconc, 56
ncons, 51
ne, 10
neq, 10
newid, 14, 36
nexprp, 117
next, 89
nil (global), 47
nolist* (global), 213
noncharactererror, 181
noniderror, 180
nonintegererror, 181
nonlisterror, 180
nonnumbererror, 181
nonpairerror, 180
nonpositiveintegererror, 181
nonsequenceerror, 181
nonstringerror, 181
nonvectorerror, 181
nonwordserror, 180, 181
not, 12
nstring-capitalize, 75
nstring-downcase, 75
nstring-upcase, 75
nth, 54
null, 11
numberp, 11

object-fns (global), 219
object-get-handler, 240
object-type, 239
objects-version):, 241
off, 46
on, 46
onep, 24
open, 152
options* (global), 155
or, 13
out* (global), 157
output–case, 34
outputbase* (global), 145

pair, 63
pairp, 12
pbind1, 126
pipe-open, 254
plus, 22
plus2, 22
pnth, 54
pop, 189
posn, 145
pp-file, 221
pp-object, 222
prettyprint, 143
prin1, 141
prin2, 141
prin2l, 144
prin2t, 144
princ, 144
prinlength (global), 146
prinlevel (global), 146
print, 141
printf, 142
printscantable, 163
prog, 86
prog1, 86
prog2, 86
progn, 86
promptstring* (global), 151
prop, 41
psetf, 45
psetq, 42
push, 189
put, 37
putd, 109
putdiphthong, 163
putv, 250

putwords, 190
pwd, 253

quit, 207
quote, 134
quotient, 22

radianstodegrees, 26
radianstodms, 26
random, 31
randomseed (global), 31
rangeerror, 180
ratom, 150
rds, 157
read, 147
read-init-file, 208
readch, 148
readchar, 148
readfromshm, 256
readline, 150
recip, 22
reclaim, 209
record-macro-usage, 218
record-usage, 218
redefmsg, 3
redo, 170
reload, 154
remainder, 22
remd, 110
remflag, 40
remflag1, 40
remob, 36
remove-wrapper, 120
remprop, 38
rempropl, 38
repeat, 88
reset, 208
rest, 54
return, 88
reverse, 61
reversip, 61
robustexpand, 135
round, 25
rplaca, 52
rplacd, 52
rplachar, 76
rplacw, 53

sassoc, 63
savesystem, 207
scanalyze, 216
scanalyze-file, 215
scanalyze-form, 215
sec, 28
secd, 28
second, 53
selectq, 84
send, 237
send-if-handles, 237
set, 41
setf, 42
setindx, 251
setprop, 41
setq, 41
setsub, 251
setsubseq, 252
setup-some-xref-actions, 218
shm-open, 256
sin, 27
sind, 27
size, 251
socketflushbuffer, 255
socketopen, 255
spaces, 142
specialclosefunction* (global), 161
specialrdsaction* (global), 158
specialreadfunction* (global), 161
specialwritefunction* (global), 161
specialwrsaction* (global), 158
sqrt, 30
standard-charp, 67
stderror, 180
stdin* (global), 157
stdout* (global), 158
string, 15, 71
string<, 73

string<=, 73
string<>, 73
string>, 73
string>=, 73
string-capitalize, 75
string-charp, 68
string-concat, 74
string-downcase, 75
string-empty?, 72
string-equal, 73
string-fetch, 72
string-greaterp, 73
string-left-trim, 75
string-length, 72, 76
string-lessp, 73
string-not-equal, 74
string-not-greaterp, 73
string-not-lessp, 74
string-read, 78
string-repeat, 74
string-right-trim, 75
string-search, 77
string-search-equal, 77
string-search-from, 77
string-search-from-equal, 77
string-store, 72
string-to-list, 75
string-to-vector, 75
string-trim, 74
string-upcase, 75
string-upper-bound, 72
string2list, 14
string2vector, 15
string=, 72
stringgensym, 36
stringp, 12
sub, 251
sub1, 22
subla, 64
sublis, 64
subseq, 251
subst, 64
substip, 64
substring, 74
substring-equal, 76
substring=, 76
system, 253

t (global), 47
tab, 142
tan, 28
tand, 28
tconc, 57
template, 225
terpri, 142
third, 53
throw, 103
time, 209
times, 22
times2, 23
toktype*, 149
toploop, 168
toploopeval* (global), 169
toplooplevel* (global), 169
toploopname* (global), 169
toploopread* (global), 169
totalcopy, 252
tr, 184
trace, 183
transfersign, 26
trst, 184
truncatevector, 190
truncatewords, 190
twritetoshm, 256
typeerror, 180

unbindn, 126
unboundp, 45, 126
unbr, 185
undeclare-flavor, 239
unfluid, 126
union, 57
unionq, 58
unless, 82
unquote, 115

unquotel, 115
unreadchar, 149
untr, 184
unwind-all, 104
unwind-protect, 104
upbv, 250
upbw, 190
uppercasep, 68
usagetypeerror, 180
useful-fns (global), 219
user, 48
user-homedir-string, 208
usermode, 3

valuecell, 45
vector, 15, 248
vector-empty?, 249
vector-fetch, 2, 249
vector-size, 249
vector-store, 249
vector-upper-bound, 249
vector2list, 16
vector2string, 15
vectorp, 12

when, 82
while, 88
with-input-from-string, 78
with-output-to-string, 79
wrap, 120
wrapped?, 120
wrapper-of-type?, 120
wrapper-standard-order (global), 121
wrapper-types, 121
writechar, 144
wrs, 157

xcons, 51
xref-assert, 217
xref-assert-list, 217

yesp, 151

zerop, 24