TR-10 March 1981

The Portable Standard LISP Users Manual
Part 1: Language Specification

by
The Utah Symbolic Computation Group

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Version 3.2: 16 March 1984

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 the DEC-20, VAX, and MC68000.

Work supported in part by the Hewlett Packard Company, the International Business Machines Corporation and the National Science Foundation under Grant Numbers MCS80-07034 and MCS82-04247.

PREFACE

This Portable LISP implementation would not have been started without the effort and inspiration of the original STANDARD LISP reporters (A. C. Hearn, J. Marti, M. L. Griss and C. Griss) and the many people who gave freely of their advice (often unsolicited!). We especially appreciate the comments of A. Norman, M. Rothstein, H. Stoyan and T. Ager.

It would not have been completed without the efforts of the many people who have worked arduously on SYSLISP and PSL at various levels: Eric Benson, Will Galway, Ellen Gibson, Martin Griss, Bob Kessler, Steve Lowder, Chip Maguire, Beryl Morrison, Don Morrison, Bobbie Othmer, Bob Pendleton, John Peterson, and John W. Peterson.

We are also grateful for the many comments and significant contributions by the LISP users at the Hewlett-Packard Computer Research Center in Palo Alto.

This document has been worked on by most members of the current Utah Symbolic Computation Group. The primary editorial function has been in the hands of B. Morrison, M. L. Griss, B. Othmer, and W. Galway; major sections have been contributed by E. Benson, W. Galway, and D. Morrison. There have also been significant contributions to the manual from Hewlett-Packard.

We have reorganized the manual for this version, following the Common Lisp idea of having four parts for language definition, utilities, system-dependent information, and implementation information. Most of this reorganization was done at Hewlett-Packard.

This is a preliminary version of the manual, and so may suffer from a number of errors and omissions. Please let us know of problems you may detect.

We have also made some stylistic decisions regarding font to indicate semantic classification and case to make symbols more readable. Based on feedback from users of the earlier 3.0 PSL release and manual, we have decided to use LISP syntax as the primary description language; where appropriate RLISP syntax also appears. We would appreciate comments on these and other decisions.

Report bugs, errors and mis-features by sending MAIL to PSL-BUGS@Utah-20.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1. Opening Remarks

This document describes PSL (PORTABLE STANDARD LISP[1]), a portable, "modern" LISP developed at the University of Utah for a variety of machines. PSL is upward-compatible with STANDARD LISP [Marti 79]. 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 (such as existing PASCAL or ADA applications).

* Effectively supporting the REDUCE algebra system on a number of machines, and providing algebra modules extracted from (or modeled upon) REDUCE to be included in applications such as CAI and CAGD.

* Providing a uniform, modern LISP programming environment on all of the machines that we use (DEC-20, VAX, and 68000 based personal machines)--of the power of FRANZ LISP, UCI LISP or MACLISP.

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

[1]"LSP" backwards!

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 very helpful. A selection of LISP primers is listed in the bibliography in Chapter 16; see for example [Allen 79, Charniak 80, Weissman 67, Winston 81].

The PSL documentation is divided into four parts following the Common LISP practice. Part 1, the "white pages" (this document), is a language specification. Part 2, the "yellow pages", is a program library document. Part 3, the "red pages", is implementation-dependent documentation. Part 4, the "blue pages", is an implementation guide.

1.2.1. Typographic Conventions within the Manual

A large proportion of this manual is devoted to descriptions of the functions that make up PSL. Each function is provided with a prototypical header line. Each argument is given a name and followed by its allowed type. If an argument type is not commonly used, it may be a specific set enclosed in brackets {...}. For example, this header shows that PutD (which defines other functions) takes three arguments:

(PutD FNAME:id TYPE:ftype BODY:{lambda, code-pointer}): FNAME:id expr

1. FNAME, which is an id (identifier).

2. TYPE, which is the "function type" of the function being defined.

3. BODY, which is a lambda expression or a code-pointer.

and returns FNAME, the name of the function being defined. Some functions are compiled open; these 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. For example:

(And [U:form]): extra-boolean

And is a function which accepts zero or more arguments each of which may be any form.

In some cases, LISP or RLISP code is given in the function documentation as the function's definition. As far as possible, the code is extracted from the the current PSL sources (perhaps converted from one syntax to the other); however, this code is not always necessarily used in PSL, and may be given only to clarify the semantics of the function. Please check carefully if you depend on the exact definition.

Some features of PSL are anticipated but not yet fully implemented. When these are documented in this manual they are indicated with the words: **[not implemented yet]**.

1.2.2. The Organization of the Manual

This manual is arranged in separate chapters, which are meant to be self-contained units. Each begins with a small table of contents serving as a summary of constructs and as an aid in skimming. Here is a brief overview of the following chapters:

Chapter 2 describes the data types used in PSL. It includes functions useful for testing equality and for changing data types, and predicates useful with data types.

The next seven chapters describe in detail the basic functions provided by PSL.

Chapters 3, 4, 5, and 6 describe functions for manipulating the basic data structures of LISP: numbers, ids, lists, and strings and vectors. As virtually every LISP program uses integers, identifiers, and lists extensively, these three chapters (3, 4 and 5) should be included in an overview. As vectors and strings are used less extensively, Chapter 6 may be skipped on a first reading.

Chapter 7 and, to some extent, Chapter 2 describe the basic functions used to drive a computation. The reader wanting an overview of PSL should certainly read these two.

Chapter 8 describes functions useful in function definition and the idea of variable binding. The novice LISP user should definitely read this information before proceeding to the rest of the manual.

Chapter 9 describes functions associated with the interpreter. It includes functions having to do with evaluation (`Eval` and `Apply`.)

Chapter 10 describes the I/O facilities. Most LISP programs do not require sophisticated

I/O, so this may be skimmed on a first reading. The section dealing with input deals extensively with customizing the scanner and reader, which is only of interest to the sophisticated user.

Chapter 11 presents information about the user interface for PSL. It includes some generally useful information on running the system.

Chapter 12 discusses error handling. Much of the information is of interest primarily to the sophisticated user. However, LISP provides a convenient interactive facility for correcting certain errors which may be of interest to all, so a first reading should include parts of this chapter.

Chapter 13 discusses some tools for debugging and statistics gathering based on the concept of embedding function definitions.

Chapter 14 describes some miscellaneous useful facilities.

Chapter 15 describes functions associated with the compiler.

Chapter 16 contains the bibliography.

Chapter 17 is an alphabetical index of concepts. Chapter 18 is an alphabetical index of all functions defined in the manual. Chapter 19 contains an alphabetical index of all global variables and switches defined in the manual.

CHAPTER 2
DATA TYPES

2.1. Data Types and Structures Supported in PSL

2.1.1. Data Types

Data objects in PSL are tagged with their type. This means that the type declarations required in many programming languages are not needed. Some functions are "generic" in that the result they return depends on the types of the arguments. A tagged PSL object is called an item, and has a tag field (9 bits on the DEC-20, 5 bits on the VAX), an info field (18 bits on the DEC-20, 27 bits on the VAX), and possibly some bits for garbage collection. The info field is either immediate data or an index or address into some other structure (such as the heap or id space). For the purposes of input and output of items, an appropriate notation is used (see Chapter 10 for full details on syntax, restrictions, etc.). More explicit implementation details can be found in Part 4 of the manual.

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, BIG, is loaded (LOAD BIG). The notation for integers is a sequence of digits in an appropriate radix (radix 10 is the default, which can be overridden by a radix 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. (19 bit [-2 ^ 18 ... 2 ^ 18 - 1] on the DEC-20, 28 bit on the VAX.)

fixnum A full-word signed integer, allocated in the heap. (36 bit on the DEC-20, fitting into a register; 32 bit on the VAX.)

[??? Do we need fixnums, and if yes how large ???]

bignum A signed integer of arbitrary precision, allocated as a vector of integers. Bignums are currently not installed by default; to use them load the module BIG.

float A floating point number, allocated in the heap. The precision of floats is determined solely by the implementation, and is 72-bit double precision on the DEC-20, 64-bit on the VAX. The notation for a float is a sequence of digits with the addition of a single floating point (.) and 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 starting with a letter. 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 10 for more information on ids and their syntax. NIL and T are treated as special ids in PSL.

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 10).

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 6.3. 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""". (Input strings may cross the end-of-line boundary, but a warning is given.) See

!*EOLInStringOK in chapter 10.

word-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. [??? The word-vector could be used to implement machine-code blocks on some machines. ???]
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.
Halfword-Vector	A vector of machine-sized halfwords.
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 (#<Code a nnnn> on the DEC-20 and VAX, where a is the number of arguments and nnnn is the entry point).
env-pointer	A data type used to support a funarg capability. **[not implemented yet]**

2.1.2. 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 Chapter 4.6).
character	Integers in the range of 0 to 127 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 9).
extra-boolean	Any value in the system. Anything that is not NIL has the boolean interpretation T.
ftype	The class of definable function types. The set of ids {expr, fexpr, macro, nexpr}. 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 I/O channel.
number	The class of {integer, float}.
x-vector	Any kind of vector; i.e., a string, vector, word-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. (e.g., Go)

2.1.3. 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 and third examples.

```
dot-notation                    list-notation
(a . (b . (c . NIL)))           (a b c)
(a . (b . c))                   (a b . c)
(a . ((b . c) . (d . NIL)))     (a (b . c) d)
```

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. (See Chapter 9 for more details.)

lambda A lambda expression must have the form (in list-notation): (lambda parameters . body). "Parameters" is an id-list of formal parameters for "body", which is a form to be evaluated (note the implicit ProgN). The semantics of the evaluation are defined by the Eval function (see Chapter 9).

function A lambda, or a code-pointer. A function is always evaluated as Eval, Spread.

2.2. 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.

2.2.1. 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.

[??? Should numbers of different type be EqN? e.g., 0 vs. 0.0 ???]

(Equal U:any V:any): boolean expr

Returns T if U and V are the same. Pairs are compared recursively to the bottom levels of their trees. Vectors must have identical dimensions and Equal values in all positions. Strings must have identical characters, i.e. all characters must be of the same case. Code-pointers must have Eq values. Other atoms must be EqN equal. A usually valid heuristic is that if two objects look the same if printed with the function Print, they are Equal. If one argument is known to be an atom, Equal is open-compiled as Eq.

```
        For example, if
            (Setq X '(A B C)) and (Setq Y X) have been executed, then
            (EQ X Y) is T
            (EQ X '(A B C)) is NIL
            (EQUAL X '(A B C)) is T
            (EQ 1 1) is T
            (EQ 1.0 1.0) is NIL
            (EQN 1.0 1.0) is T
            (EQN 1 1.0) is NIL
            (EQUAL 0 0.0) is 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. For case-INsensitive equality one must load the STRINGS module (see Section 6.7). 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.

2.2.2. 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.

[??? Should Eval U Eq U if U is a constant? ???]

(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.

2.2.3. 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. NIL may also be represented as '(). The Boolean functions **And**, **Or**, and **Not** can be applied to any LISP type, and are not bitwise functions. **And** and **Or** are frequently used in LISP as control structures as well as Boolean connectives (see Section 7.1). For example, the following two constructs will give the same result:

```
(COND ((AND A B C) D))

(AND A B C D)
```

Since there is no specific Boolean type in LISP and since every LISP expression has a value which may be used freely in conditionals, there is no hard and fast 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.

(**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.

2.3. 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

Gets an id on the id-hash-table. The argument may be an id. Intern searches the id-hash-table (or current id-hash-table if the package system is loaded) for an id with the same print name as U and returns the id on the id-hash-table if a match is found. (See Chapter 4 for a discussion of the id-hash-table. Any properties and GLOBAL values associated with the uninterned U are lost. If U does not match any entry, a new one is created and returned. The argument may also be a string in which case an identifier in the id-hash-table is looked up, created if necessary, and returned. Note carefully: The id returned from Interning a string has exactly the same print name as the string. Most identifiers have **uppercase** print names (even if you type in lower case!), but interning "abc" yields an id with a lower case print name.

```
(EQ (INTERN "abc") 'abc) = NIL
```

[??? Rewrite for package system; include search path, global, local, intern, etc. ???]

The maximum number of characters in any token is 5000.

(NewId S:string): id expr

Allocates a new uninterned id, and sets its print-name to the string S. The string is not copied.

```
(Setq New (NewId "NEWONE")) returns  NEWONE
```

Note that if one refers directly to the id NEWONE, it will become interned and a new position in the id space will be allocated to it. One has to refer to the new id indirectly through the id New.

(Int2Id I:integer): id   expr

Converts an integer to an id; this refers to the I'th id in the id space. Since 0 ... 127 correspond to ASCII characters, Int2Id with an argument in this range converts an ASCII code to the corresponding single character id.

```
(Int2Id 250)  returns QUOTIENT
```

(Id2Int D:id): integer   expr

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

```
(Id2Int 'String) returns 182
```

(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. See CopyString in Chapter 6.

[??? Should it be a copy? ???]

```
(Id2String 'String)  returns "STRING"
```

(String2List S:string): inum-list   expr

Creates a list of Length (Add1 (Size S)), converting the ASCII characters into small integers.

[??? What of 0/1 base for length vs length -1. What of the NUL char added ???]

```
(String2List "STRING")  returns (83 84 82 73 78 71)
```

(List2String L:inum-list): string   expr

Allocates a string of the same Size as L, and converts inums to characters according to their ASCII code. The inums must be in the range 0 ... 127.

[??? Check if 0 ... 127, and signal error ???]

```
(List2String '(83 84 82 73 78 71))  returns "STRING"
```

(String [I:inum]): string <u>nexpr</u>

Creates and returns a <u>string</u> containing all the <u>inums</u> given.

```
(String 83 84 82 73 78 71)  returns "STRING"
```

(Vector [U:any]): vector <u>nexpr</u>

Creates and returns a <u>vector</u> containing all the <u>U</u>s given.

```
(Setq X (Vector 83 84 82 73 78 71))  returns
 [83 84 82 73 78 71]
```

(Vector2String V:vector): string <u>expr</u>

Pack the small <u>integers</u> in the <u>vector</u> into a <u>string</u> of the same Size, using the <u>integers</u> as ASCII values.

[??? check for integer in range 0 ... 127 ???]

```
(Vector2String X)  where X is defined as above returns
        "STRING"
```

(String2Vector S:string): vector <u>expr</u>

Unpack the <u>string</u> into a <u>vector</u> of the same Size. The elements of the <u>vector</u> are small integers, representing the ASCII values of the characters in <u>S</u>.

```
(String2Vector "VECTOR") returns [V E C T O R]
```

(Vector2List V:vector): list <u>expr</u>

Create a <u>list</u> of the same Size as <u>V</u> (i.e. of Length Upbv(<u>V</u>)+1), copying the elements in order 0, 1, ..., Upbv(<u>V</u>).

```
(Vector2List [L I S T])  returns (L I S T)
```

(List2Vector L:list): vector <u>expr</u>

Copy the elements of the <u>list</u> into a <u>vector</u> of the same Size.

```
(List2Vector '(V E C T O R)) returns [V E C T O R]
```

(Int2Sys I:integer): UntaggedSystemWord expr

Converts an integer to an untagged system dependent word.

```
(Int2Sys 250)  returns 250
```

(Sys2Int W:UntaggedSystemWord): Inum or FixNum expr

If the untagged system dependent word will fit into an inum, it will be converted into an inum, otherwise it will be converted into a fixnum. depending on the size.

```
(Sys2Int (GetMem XX)) returns tagged item at memory location XX
```

(Lisp2Char X:{Integer or ID or String}): CharacterNumber expr

If argument is an integer in the 0 to 127 range, then the integer is returned, otherwise if the argument is an identifier then the character value of the first character is returned, otherwise if the argument is a string, the character value of the first character is returned, otherwise a Non Character Error is signalled.

```
(Lisp2Char 32) returns 32
(Lisp2Char 'AA) returns 65
(Lisp2Char "hello") returns 104
(Lisp2Char 500) produces the error:
***** An attempt was made to do a 'LISP2CHAR' on '400',
      which is not a character.
```

(Int2Code I:Integer): CodePointer expr

Converts the argument integer into a code-pointer.

```
(Int2Code 3456) returns #<Code 6600>
```

CHAPTER 3
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. See Section 3.6 for documentation for these functions.

3.1. Big Integers

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

Note that fixnums which are present before loading BIG can cause problems, because loading BIG restricts the legal range of fixnums.

3.2. 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.

[??? Note that unless big is loaded, a float with value larger than 2**35-1 on the DEC-20 is converted into something strange but without any error message. Note how truncation works on negative numbers (always towards zero). ???]

```
(Fix 2.1)  % returns 2

(Fix -2.1) %  returns -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. Float of a float returns the number unchanged. If U is too large to represent in float, an error occurs:

***** Argument to FLOAT is too large

[??? Only if big is loaded can one make an integer of value greater than 2**35-1, so without big you won't get this error message. The largest representable float is (2**62-1)*(2**65) on the DEC-20. ???]

3.3. 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

Part of the USEFUL package (LOAD USEFUL). 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> (Load Useful)
NIL
2 lisp> (Setq Y '(1 5 7))
(1 5 7)
3 lisp> (Decr (Car Y))
0
4 lisp> Y
(0 5 7)
5 lisp> (Decr (Cadr Y) 3 4)
-2
6 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 (LOAD USEFUL). 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

If both U and V are integers the result is the integer remainder of U divided by V. The sign of the result is the same as the sign of the dividend (U). If U and V are not both integers, the result is currently undefined. An error occurs if V is zero:

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

Note that the Remainder function differs from the Mod function in that Remainder returns a negative number when U is negative and V is positive.

(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.

3.4. Functions for Numeric Comparison

The following functions compare the values of their arguments. For functions testing equality (or non-equality) see Section 2.2.1.

(Geq U:any V:any): boolean expr

Returns T if U >= V, otherwise returns NIL. In RLISP, the symbol ">=" can be used.

(GreaterP U:number V:number): boolean expr

Returns T if U is strictly greater than V, otherwise returns NIL. In RLISP, the symbol ">" can be used.

(Leq U:number V:number): boolean <u>expr</u>

Returns T if <u>U</u> <= <u>V</u>, otherwise returns NIL. In RLISP, the symbol "<=" can be used.

(LessP U:number V:number): boolean <u>expr</u>

Returns T if <u>U</u> is strictly less than <u>V</u>, otherwise returns NIL. In RLISP, the symbol "<" can be used.

(Max [U:number]): number <u>macro</u>

Returns the largest of the values in <u>U</u> (numeric maximum). If two or more values are the same, the first is returned.

(Max2 U:number V:number): number <u>expr</u>

Returns the larger of <u>U</u> and <u>V</u>. If <u>U</u> and <u>V</u> are of the same value <u>U</u> is returned (<u>U</u> and <u>V</u> might be of different types).

(Min [U:number]): number <u>macro</u>

Returns the smallest (numeric minimum) of the values in <u>U</u>. If two or more values are the same, the first of these is returned.

(Min2 U:number V:number): number <u>expr</u>

Returns the smaller of its arguments. If <u>U</u> and <u>V</u> are the same value, <u>U</u> is returned (<u>U</u> and <u>V</u> might be of different types).

(MinusP U:any): boolean <u>expr</u>

Returns T if <u>U</u> is a <u>number</u> and less than 0. If <u>U</u> is not a <u>number</u> or is a positive <u>number</u>, NIL is returned.

(OneP U:any): boolean <u>expr</u>

Returns T if <u>U</u> is a <u>number</u> 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.

3.5. 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 according to the following table.

U	0	0	1	1
V	0	1	0	1
Returned Value	0	0	0	1

(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 according to the following table.

U	0	0	1	1
V	0	1	0	1
Returned Value	0	1	1	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.

[??? need to clarify a bit more ???]

(LXOr U:integer V:integer): integer expr

Bitwise or logical exclusive Or. Each bit of the result is independently determined from the corresponding bits of the operands according to the following table.

U	0	0	1	1
V	0	1	0	1
Returned Value	0	1	1	0

(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.

3.6. 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, but this implementation tends to support fewer features. The examples used here should be taken with a grain of salt, since the precision of the results will depend on the machine being used, and may change in later implementations of the module.

(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.[1]

(TransferSign S:number Val:number): number expr

Transfers the sign of S to VAL by returning abs(VAL) if S >= 0, and -abs(VAL) otherwise. (The same as FORTRANs sign function.)

(Mod M:integer N:integer): integer expr

Returns M modulo N. Unlike the remainder function, it returns a positive number in the range 0..N-1 when N is positive, even if M is negative.

```
1 lisp> (mod -7 5)
3
2 lisp> (remainder -7 5)
-2
```

[??? Allow to "number" arguments instead of just "integers"? ???]

(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
```

[1]The behavior of Round is ambiguous when its argument ends in ".5"--needs more work.

(RadiansToDMS X:number): list   _expr_

Given an angle _X_ in radians, returns a list of three _integers_ giving the angle in

```
(Degrees   Minutes   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 <u>expr</u>

Returns the cosine of <u>X</u>, an angle in radians.

(CosD X:number): number <u>expr</u>

Returns the cosine of <u>X</u>, an angle in degrees.

(Tan X:number): number <u>expr</u>

Returns the tangent of <u>X</u>, an angle in radians.

(TanD X:number): number <u>expr</u>

Returns the tangent of <u>X</u>, an angle in degrees.

(Cot X:number): number <u>expr</u>

Returns the cotangent of <u>X</u>, an angle in radians.

(CotD X:number): number <u>expr</u>

Returns the cotangent of <u>X</u>, an angle in degrees.

(Sec X:number): number <u>expr</u>

Returns the secant of <u>X</u>, an angle in radians.

secant(X) = 1/cos(X)

(SecD X:number): number <u>expr</u>

Returns the secant of <u>X</u>, an angle in degrees.

(Csc X:number): number <u>expr</u>

Returns the cosecant of <u>X</u>, an angle in radians.

secant(X) = 1/sin(X)

(CscD X:number): number <u>expr</u>

Returns the cosecant of <u>X</u>, an angle in degrees.

(Asin X:number): number *expr*

Returns the arc sine, as an angle in radians, of X.

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.

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.

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.

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.

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.

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, i.e. e^X.

(Log X:number): number expr

Returns the natural (base e) logarithm of X.

log(exp(X)) = 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..N-1.

The random number generator uses a linear congruential method. To get a reproducible sequence of random numbers you should assign one (or some other small number) to the FLUID variable RandomSeed.

RandomSeed [Initially: set from time] *global*

(Factorial N:integer): integer *expr*

Returns the factorial of N.

factorial(0) = 1

factorial(N) = N*factorial(N-1)

CHAPTER 4
IDENTIFIERS

4.1. Introduction

In PSL variables are called identifiers or ids. An identifier is implemented as a tagged data object (described in Chapter 2) containing a pointer or offset into a four item structure - the id space. One item in this structure is called the print name, which is the external representation of the id.

The interpreter uses an id hash table to get from the print name of an identifier to its entry in the id space. The id space and the id hash table are described below.

4.2. Fields of Ids

An id is an item with an info field; the info field is an offset into a special id space consisting of structures of four fields. The fields (items) are:

print-name	The print name points at a string of characters which is the external representation of the identifier. The syntax for identifiers is described in Section 10.4 on reading functions.
value-cell	The value of the identifier or a pointer to the value in the heap is stored in this field. If no value exists, this cell contains an unbound identifier indicator. These cells can be accessed by functions defined in this chapter.
function-cell	An id may have a function or macro associated with it. Access is by

means of the PutD, GetD, and RemD functions defined in Section 8.2.2.

package-cell PSL permits the use of a multiple package facility (multiple id hash table). The package cell refers to the appropriate id hash table.

4.3. Identifiers and the Id hash table

The method used by PSL to retrieve information about an identifier makes use of the id hash table (corresponding to the Oblist, or Object list, in some versions of LISP). A hash function is applied to the identifier name giving a position in the id hash table. The contents of the hash table at that point contain an offset into the id space. For a new identifier, the next free position in the id space is found and a pointer to it is placed in the hash table entry.

The process of putting an id into the hash table is called interning. This is done automatically by the LISP reader, so any id typed in at the terminal is interned. Interning can also be done by the programmer using the function Intern to convert a string to an id. An id may have an entry in the id space without being interned. In fact it is possible to have several ids with the same print name, one interned and the others not.

Note that when one starts PSL, the id space already contains approximately 2000 ids. These include all of the ASCII characters, the functions and globals described in this manual, plus system functions and globals. If a user uses any of these names for his own functions or globals, there can be a conflict. A warning message appears if a user tries to redefine a system function.

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

If the user answers "Y", his definition replaces the current definition. (See Chapter 8 for a description of the switch !*USERMODE which controls the printing of this message.)

Information on converting ids to other types can be found in Chapter 10 and Section 2.3.

4.3.1. Identifier Functions

The following functions deal with identifiers and the id hash table.

(GenSym): id   expr

Creates an identifier which is not interned on the id hash table and consequently not Eq to anything else. The id is derived from a string of the form "G0000", which is incremented upon each call to GenSym.

[??? Is this interned or recorded on the NIL package ???]

[??? Can we change the GenSym string ???]

(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): U:id   expr

If U is present on the current package search path it is removed. This does not affect U having properties, flags, functions and the like. U is returned.

(InternP U:{id,string}): boolean   expr

Returns T if U is interned in the current search path.

(MapObl FNAME:function): Undefined   expr

MapObl applies function FNAME to each id interned in the current hash table.

4.3.2. Find

These functions take a string or id as an argument, and scan the id hash table to collect a list of ids with prefix or suffix matching the argument. This is a loadable option (LOAD FIND).

(FindPrefix KEY:{id, string}): id-list expr

Scans current id hash table for all ids whose prefix matches KEY. Returns all the identifiers found as an alphabetically sorted list.

(FindSuffix KEY:{id, string}): id-list expr

Scans current id hash table for all ids whose suffix matches KEY. Returns all the identifiers found as an alphabetically sorted list.

```
   (Setq X (FindPrefix '!*)  % Finds all identifiers starting with *

   (Setq Y (FindSuffix "STRING")) % Finds all identifiers ending with STRING
```

4.4. Property List Functions

The property cell of an identifier points to a "property list". The list is used to quickly associate an id name with a set of entities; those entities are called "flags" if their use gives the id a boolean value, and "properties" if the id is to have an arbitrary attribute (an indicator with a property).

(Put U:id IND:id PROP:any): any expr

The indicator IND with the property PROP is placed on the property list of the id U. If the action of Put occurs, the value of PROP is returned. If either of U and IND are not ids the type mismatch error occurs and no property is placed.

```
(Put 'Jim 'Height 68)
```

The above returns 68 and places (Height . 68) on the property list of the id Jim.

(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. (In older LISPs, Get could access functions.) Get returns NIL if U is not an id.

```
(Get 'Jim 'Height) returns 68
```

(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 Put function. The value of DefList is a list of the first elements of each two-element list. Like Put, DefList may not be used to define functions.

```
(DE DEFLIST (U IND)
     (COND ((NULL U) NIL)
           (T (CONS(PROGN(PUT (CAAR U) IND (CADAR U))
                         (CAAR U))
                   (DEFLIST (CDR U) IND)))))
```

(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*

Remove property IND from all ids in U.

4.4.1. Functions for Flagging Ids

In some LISPs, flags and indicators may clash. In PSL, flags are ids and properties are pairs on the prop-list, so no clash occurs.

(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 the elements of U must be identifiers or the type mismatch error occurs. After Flagging, the id V appears on the property list of each id X in U. However, flags cannot be accessed, placed on, or removed from property lists using normal property list functions Get, Put, and RemProp. Note that if an error occurs during execution of Flag, then some of the ids on 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

Puts flag V on the property list of id U.

(RemFlag1 U:id V:any): Undefined	expr

Removes the flag V from the property list of id U.

[??? Make Flag1 and RemFlag1 return single value. ???]

4.4.2. 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.

4.5. Value Cell Functions

The contents of the value cell are usually accessed by Eval (Chapter 9) or ValueCell (below) and changed by SetQ or sometimes Set.

(SetQ VARIABLE:id VALUE:any): any	fexpr

The value of the current binding of VARIABLE is replaced by the value of VALUE.

```
(SETQ X 1)
```

is equivalent to

```
(SET 'X 1)
```

SetQ now conforms to the Common LISP standard, allowing sequential assignment:

```
(SETQ A 1 B 2)
   ==> (SETQ A 1)
       (SETQ B 2)
```

(Set EXP:id VALUE:any): any <u>expr</u>

<u>EXP</u> must be an <u>identifier</u> or a type mismatch error occurs. The effect of Set is replacement of the item bound to the identifier by <u>VALUE</u>. If the identifier is not a LOCAL variable or has not been declared GLOBAL, it is automatically declared FLUID with the resulting warning message:

*** EXP declared FLUID

<u>EXP</u> must not evaluate to T or NIL or an error occurs:

***** Cannot change T or NIL

(DeSetQ U:any V:any): V:any <u>macro</u>

This is a function in the USEFUL package. DeSetQ is a destructuring SetQ. That is, the first argument is a piece of <u>list</u> structure whose <u>atoms</u> are all <u>id</u>s. Each is SetQ'd to the corresponding part of the second argument. For instance

```
(DeSetQ (a (b) . c) '((1) (2) (3) 4))
```

SetQ's a to (1), b to 2, and c to ((3) 4).

(PSetQ [VARIABLE:id VALUE:any]): Undefined <u>macro</u>

Part of the USEFUL package (LOAD USEFUL).

```
(PSETQ VAR1 VAL1 VAR2 VAL2 ...  VARn VALn)
```

SetQ's the VAR's to the corresponding VAL's. The VAL's are all evaluated before any assignments are made. That is, this is a parallel SetQ.

(SetF [LHS:form RHS:any]): RHS:any macro

There are two versions of SetF. SetF is redefined on loading USEFUL. The description below is for the resident SetF. SetF provides a method for assigning values to expressions more general than simple ids. For example:

```
(SETF (CAR X) 2)
    ==> CAR X := 2;
```

is equivalent to

```
(RPLACA X 2)
```

In general, SetF has the form

```
(SetF LHS RHS)
```

in which LHS is the "left hand side" to be assigned to and RHS is evaluated to the value to be assigned. LHS can be one of the following:

id	SetQ is used to assign a value to the id.
(Eval expression)	Set is used instead of SetQ. In effect, the "Eval" cancels out the "Quote" which would normally be used.
(Value expression)	Is treated the same as Eval.
(Car pair)	RplacA is used to store into the Car "field".
(Cdr pair)	RplacD is used to store into the Cdr "field".
(GetV vector)	PutV is used to store into the appropriate location.
(Indx "indexable object")	SetIndx is used to store into the object.
(Sub vector)	SetSub is used to store into the appropriate subrange of the vector.

Note that if the LHS is (Car pair) or (Cdr pair), SetF returns the modified pair instead of the RHS, because SetF uses RplacA and RplacD in these cases.

Loading USEFUL brings in declarations to SetF about Caar, Cadr, ... Cddddr. This is rather handy with constructor/selector macros. For instance, if FOO is a selector which maps to Cadadr,

```
(SETF (FOO X) Y)
```

works; that is, it maps to something which does a

```
(RPLACA (CDADR X) Y)
```

and then returns X.

(PSetF [LHS:form RHS:any]): Undefined macro

Part of the USEFUL package (LOAD USEFUL). PSetF does a SetF in parallel: i.e., it evaluates all the right hand sides (RHS) before assigning any to the left hand sides (LHS).

(MakeUnBound U:id): Undefined expr

Make U an unbound id by storing a "magic" number in the value cell.

(ValueCell U:id): any expr

Safe access to the value cell of an id. If U is not an id a type mismatch error is signalled; if U is an unbound id, an unbound id error is signalled. Otherwise the current value of U is returned. [See also the Value and LispVar functions, described in [], for more direct access].

(UnBoundP U:id): boolean expr

Tests whether U has no value.

[??? Define and describe General Property LISTs or hash-tables. See Hcons. ???]

4.6. System Global Variables, Switches and Other "Hooks"

4.6.1. Introduction

A number of global variables provide global control of the LISP system, or implement values which are constant throughout execution. Certain options are controlled by switches, with T or NIL properties (e.g., ECHOing as a file is read in); others require a value, such as an integer for the current output base. PSL has the convention (following the REDUCE/RLISP convention) of using a "!*" in the name of the variable: **!***xxxxx for GLOBAL variables expecting a T/NIL value (called "switches"), and xxxxx**!*** for other GLOBALs. Chapter 19 is an index of switches and global variables used in PSL.

[??? These should all be FLUIDs, so that ANY one of these variables may be rebound, as appropriate ???]

4.6.2. Setting Switches

Strictly speaking, `xxxx` is a switch and **`!*xxxx`** is a corresponding global variable that assumes the T/NIL value; both are loosely referred to as switches elsewhere in the manual.

The On and Off functions are used to change the values of the variables 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]): None <u>macro</u>

For each <u>U</u>, the associated **!*U** variable is set to T. If a "(T (action-for-ON))" clause is found by (GET <u>U</u> 'SIMPFG), the "action" is EVAL'ed.

(Off [U:id]): None <u>macro</u>

For each <u>U</u>, the associated **!*U** variable is set to NIL. If a "(NIL (action-for-OFF)" clause is found by (GET <u>U</u> 'SIMPFG), the "action" is EVAL'ed.

```
(On Comp Ord Usermode)
```

will set !*Comp, !*Ord, and !*Usermode to T.

Note that

```
(Get 'Cref 'Simpfg)
```

returns

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

```
((T (Crefon)) (Nil (Crefoff)))
```

Setting CREF on will result in !*CREF being set to T and the function Crefon being evaluated.

4.6.3. Special Global Variables

NIL [Initially: NIL] global

NIL is a special GLOBAL variable. It is protected from being modified by Set or SetQ.

T [Initially: T] global

T is a special GLOBAL variable. It is protected from being modified by Set or SetQ.

4.6.4. Special Put Indicators

Some actions search the property list of relevant ids for these indicators:

'HELPFUNCTION	An id, a function to be executed to give help about the topic; ideally for a complex topic, a clever function is used.
'HELPSTRING	A help string, kept in core for important or short topics.
'HELPFILE	The most common case, the name of a file to print; later we hope to load this file into an EMODE buffer for perusal in a window.
'SWITCHINFO	A string describing the purpose of the SWITCH, see ShowSwitches below.
'GLOBALINFO	A string describing the purpose of the GLOBAL, see ShowGlobals below.
'BREAKFUNCTION	Associates a function to be run with an Id typed at Break Loop, see Chapter 12.
'TYPE	PSL uses the property TYPE to indicate whether a function is a fexpr, macro, or nexpr; if no property is present, expr is assumed.
'VARTYPE	PSL uses the property VARTYPE to indicate whether an identifier is of type GLOBAL or FLUID.
'TRACE	Used by the debug facility to record information about the debug

facilities being used and the original function definition.

'!*LAMBDALINK The interpreter also looks under '!*LAMBDALINK for a Lambda expression, if a procedure is not compiled.

The compiler and loader use the following indicators: MC, CONST, EXTVAR, MEMMOD, NOSIDEEFFECT, REG, TERMINAL, TRANSFER, VAR, ANYREG, CFNTYPE, DESTROYS, DOFN, EMITFN, EXITING, FLIPTST, GROUPOPS, MATCHFN, NEGJMP, ONE, PATTERN, SUBSTFN, ZERO. This are described in more detail with the documentation of compiler and loader implementation.

4.6.5. Special Flag Indicators

'EVAL If the id is flagged EVAL, the RLISP top-loop evaluates and outputs any expression (id ...) in On Defn (!*DEFN := T) mode.

'IGNORE If the id is flagged IGNORE, the RLISP top-loop evaluates but does NOT output any expression (id ...) in On Defn (!*DEFN := T) mode.

'LOSE If an id has the 'LOSE flag, it will not be defined by PutD when it is read in.

'USER 'USER is put on all functions defined when in !*USERMODE, to distinguish them from "system" functions. See Chapter 8.

See also the functions LoadTime and CompileTime in Chapter 15.

[??? Mention Parser properties ???]

4.6.6. Displaying Information About Globals

The Help function has two options, (HELP SWITCHES) and (HELP GLOBALS), which should display the current state of a variety of switches and globals respectively. These calls have the same effect as using the functions below, using an initial table of Switches and Globals.

The function (ShowSwitches switch-list) may be used to print names, current settings and purpose of some switches. Use NIL as the switch-list to get information on ALL switches of interest; ShowSwitches in this case does a MapObl (Section 4.3.1) looking for 'SwitchInfo property.

Similarly, (ShowGlobals global-list) may be used to print names, values and purposes of important GLOBALs. Again, NIL used as the global-list causes ShowGlobals to do a MapObl looking for a 'GlobalInfo property; the result is some information about all globals of interest.

CHAPTER 5
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.

```
        -----------------
        || Car   | Cdr  ||
        -----------------
```

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

```
                -----------------
                ||   /  |  \    ||
                ----/-------\----
                   /         \
       -----------------   -----------------
       ||  A   |   B  ||   ||  C   |    D ||
       -----------------   -----------------
```

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. Note also that in RLISP a dot may be used as the infix operator for the function Cons, as in the expression x := 'y . 'z;, or as part of the notation for pairs, as in the expression x := '(y . z).

An important special case occurs frequently enough that it has a special notation. This is a list of items, terminated by convention with the id NIL. The dot and surrounding parentheses are omitted, as well as the trailing NIL. Thus

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

can be represented in list-notation as

```
(A B C)
```

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. In RLISP syntax the dot, ".", is an infix operator meaning Cons. Thus (A . (B . fn C) . D) is equivalent to Cons (A, Cons (Cons (B, fn C), D)).

(Car U:pair): any — open-compiled, expr

The left part of U is returned. A type mismatch error occurs if U is not a pair, except when U is NIL. Then NIL is returned. (Car (Cons a b)) ==> a.

(Cdr U:pair): any — open-compiled, expr

The right part of U is returned. A type mismatch error occurs if U is not a pair, except when U is NIL. Then NIL is returned. (Cdr (Cons a b)) ==> b.

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. They may return any type and are generally open-compiled. An example of their use is that Cddar p is equivalent to Cdr Cdr Car p. As with Car and Cdr, a type mismatch error occurs if the argument does not possess the specified component.

As an alternative to employing chains of CxxxxR to obscure depths, particularly in extracting elements of a list, consider the use of the functions First, Second, Third, Fourth, or Nth (Section 5.3.1), or possibly even the Defstruct package (See Part 2 of the manual).

(NCons U:any): pair　　　　　　　　　　　　　　　　　　　　expr

Equivalent to Cons (U, NIL).

(XCons U:any V:any): pair　　　　　　　　　　　　　　　　　　expr

Equivalent to Cons (V, U).

(Copy X:any): any　　　　　　　　　　　　　　　　　　　　　　expr

Copies all pairs in X, but does not make copies of atoms (including vectors and strings). For example, if A is

```
([2 5] "ATOM")
```

and B is the result of (Copy A), then

```
     (Eq A B) is NIL
but  (Eq (Car A) (Car B)) is T
and  (Eq (Cadr A) (Cadr B)) is T
```

See TotalCopy in Section 6.5. Note that Copy is recursive and will not terminate if its argument is a circular list.

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 the object containing the pair. They are most frequently used for various "efficient" functions (e.g. the non-copying ReverseIP and NConc functions, and destructive DeleteIP) and to build structures that have deliberately shared sub-structure. They are also capable of creating circular structures, which create havoc with normal printing and list traversal functions. Be **careful** using them.

(RplacA U:pair V:any): pair   open-compiled, expr

The Car of the pair U is replaced by V, and the modified U is returned. (If U is (a . b) then (V .b) is returned). A type mismatch error occurs if U is not a pair.

(RplacD U:pair V:any): pair   open-compiled, expr

The Cdr of the pair U is replaced by V, and the modified U is returned. (If U is (a . b) then (a . V) is returned). A type mismatch error occurs if U is not a pair.

(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 A is returned.

[??? Should we add some more functions here someday? Probably the RLISP guys that do arbitrary depth member type stuff. ???]

5.3. Functions for Manipulating Lists

The following functions are meant for the special pairs which are lists, as described in Section 5.1. Note that the functions described in Chapter 6 can also be used on lists.

[??? Make some mention of mapping with FOR...COLLECT and such like. ???]

5.3.1. Selecting List Elements

(First L:pair): any <u>macro</u>

A synonym for Car <u>L</u>.

(Second L:pair): any <u>macro</u>

A synonym for Cadr <u>L</u>.

(Third L:pair): any <u>macro</u>

A synonym for Caddr <u>L</u>.

(Fourth L:pair): any <u>macro</u>

A synonym for Cadddr <u>L</u>.

(Rest L:pair): any <u>macro</u>

A synonym for Cdr <u>L</u>.

(LastPair L:pair): any <u>expr</u>

Last <u>pair</u> of a <u>list</u>. It is often useful to think of this as a pointer to the last element for use with destructive functions such as RplacA. Note that if <u>L</u> is atomic a type mismatch error occurs.

```
(De LastPair (L)
     (Cond ((Null (Rest L)) L)
          (T (LastPair (Rest L)))))
```

(LastCar L:any): any <u>expr</u>

Returns the last element of the <u>list</u> <u>L</u>. A type mismatch error results if <u>L</u> is not a list. Equivalent to First LastPair <u>L</u>.

(Nth L:pair N:integer): any <u>expr</u>

Returns the Nth element of the <u>list</u> <u>L</u>. If <u>L</u> is atomic or contains fewer than <u>N</u> elements, an out of range error occurs. Equivalent to (First (PNth L N)).

(PNth L:list N:integer): any <u>expr</u>

Returns <u>list</u> starting with the Nth element of a <u>list</u> <u>L</u>. Note that it is often useful to view this as a pointer to the Nth element of <u>L</u> for use with destructive functions such as RplacA. If <u>L</u> is atomic or contains fewer than <u>N</u> elements, an out of range error occurs.

```
(De PNth (L N)
        (Cond ((Leq N 1) L)
              (T (PNth (Cdr L) (Sub1 N)))))
```

5.3.2. Membership and Length of Lists

(Member A:any L:list): extra-boolean <u>expr</u>

Returns NIL if <u>A</u> is not Equal to some top level element of <u>list</u> <u>L</u>; otherwise it returns the remainder of <u>L</u> whose first element is <u>A</u>.

```
(De Member (A L)
        (Cond((Null L) Nil)
             ((Equal A (First L)) L)
             (T (Member A (Rest L)))))
```

(MemQ A:any B:list): extra-boolean <u>expr</u>

Same as Member, but an Eq check is used for comparison.

```
(De Memq (A L)
        (Cond((Null L) Nil)
             ((Eq A (First L)) L)
             (T (Memq A (Rest L)))))
```

(Length X:any): integer <u>expr</u>

The top level length of the <u>list</u> <u>X</u> is returned.

```
(De Length (X)
        (Cond((Atom X) 0)
             (T (Plus (Length (Rest X)) 1))))
```

5.3.3. Constructing, Appending, and Concatenating Lists

(List [U:any]): list fexpr

Construct a list of the evaluated arguments. A list of the evaluation of each element of U is returned.

(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 ((Null U) V)
                 (T (Cons (Car U) (Append (Cdr U) V)))))
```

(NConc U:list V:list): list expr

Destructive version of Append. Concatenates V to U without copying U. The last Cdr of U is modified to point to V. See the warning on page 5.3 about the use of destructive functions.

```
   (De Nconc (U V)
            (Cond ((Null U) V)
                  (T (Rplacd (Lastcdr U V)))))
```

(AConc U:list V:any): list expr

Destructively adds element V to the tail of list U.

(LConc PTR:list ELEM:list): list expr

Effectively NConc, but avoids scanning from the front to the end of PTR for the RPLACD(PTR, ELEM) by maintaining a pointer to end of the list PTR. PTR is (list . LastPair list). Returns updated PTR. PTR should be initialized to NIL . NIL before calling the first time. Used to build lists from left to right.

(TConc PTR:list ELEM:any): list expr

Effectively AConc, but avoids scanning from the front to the end of PTR for the RPLACD(PTR, List(ELEM)) by maintaining a pointer to end of the list PTR. PTR is (list . LastPair list). Returns updated PTR. PTR should be initialized to NIL . NIL before calling the first time. Used to build lists from left to right.

5.3.4. Lists as Sets

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

(Adjoin ELEMENT:any SET:list): list — expr

Add ELEMENT to SET if it is not already on the top level. Equal is used to test for equality.

(AdjoinQ ELEMENT:any SET:list): list — expr

Adjoin using Eq for the test whether ELEMENT is already in SET.

(Union X:list Y:list): list — expr

Set union.

(UnionQ X:list Y:list): list — expr

Eq version of Union.

(InterSection U:list V:list): list — expr

Set intersection.

(InterSectionQ U:list V:list): list — expr

Eq version of InterSection.

(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.

5.3.5. Deleting Elements of Lists

Note that functions with names of the form xxxIP indicate that xxx is done InPlace.

(Delete U:any V:list): list <u>expr</u>

Returns <u>V</u> with the first top level occurrence of <u>U</u> removed from it. That portion of <u>V</u> before the first occurrence of <u>U</u> is copied.

```
(De Delete (U V)
        (Cond((Null V) Nil)
             ((Equal (First V) U) (Rest V))
             (T (Cons (First V) (Delete U (Rest V))))))
```

(Del F:function U:any V:list): list <u>expr</u>

Generalized Delete function with <u>F</u> as the comparison function.

(DeletIP U:any V:list): list <u>expr</u>

Destructive Delete; modifies <u>V</u> using RplacD. Do not depend on <u>V</u> itself correctly referring to <u>list</u>.

(DelQ U:any V:list): list <u>expr</u>

Delete <u>U</u> from <u>V</u>, using Eq for comparison.

(DelQIP U:any V:list): list <u>expr</u>

Destructive version of DelQ; see DeletIP.

(DelAsc U:any V:a-list): a-list <u>expr</u>

Remove first (<u>U</u> . xxx) from <u>V</u>.

(DelAscIP U:any V:a-list): a-list <u>expr</u>

Destructive DelAsc.

(DelatQ U:any V:a-list): a-list <u>expr</u>

Delete first (<u>U</u> . xxx) from <u>V</u>, using Eq to check equality with <u>U</u>.

(DelatQIP U:any V:a-list): a-list expr

Destructive DelatQ.

5.3.6. List Reversal

(Reverse U:list): list expr

Returns a copy of the top level of U in reverse order.

```
(De Reverse (U)
        (Prog (W)
          (While U
            (ProgN
              (Setq W (Cons (Car U) W))
              (Setq U (Cdr U))))
          (Return W)))
```

(ReversIP U:list): list expr

Destructive Reverse.

5.3.7. Functions for Sorting

The Gsort module (LOAD GSORT) provides functions for sorting lists and vectors. Some of the functions take a comparison function as an argument. The comparison function takes two arguments and returns NIL if they are out of order, i.e. if the second argument should come before the first in the sorted result. Lambda expressions are acceptable as comparison functions.

(Gsort TABLE:list leq-fn:{id,function}): list expr

Returns a sorted list or vector. 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 the original table then X will appear before Y in the final table unless X and Y are out of order. (An unstable sort, on the other hand, might swap X and Y even if they're in order. This could happen when X and Y have the same "key field", so either one could come first without making a difference to the comparison function.)

(`GmergeSort table:list leq-fn:{id,function}`): list <u>expr</u>

The same as Gsort, but destructively modifies the <u>TABLE</u> argument. GmergeSort has the advantage of being 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 <u>expr</u>

Returns a table of <u>id</u>s sorted into alphabetical order. The original table is unchanged. Case is not significant in determining the alphabetical order. The table may contain <u>string</u>s as well as <u>id</u>s.

The following example illustrates the use of Gsort.

```
1 lisp> (load gsort)
NIL
2 lisp> (setq X '(3 8 -7 2 1 5))
(3 8 -7 2 1 5)
3 lisp>   % Sort from smallest to largest.
3 lisp> (Gsort X 'leq)
(-7 1 2 3 5 8)
4 lisp>   % Sort from largest to smallest.
4 lisp> (GmergeSort X 'geq)
(8 5 3 2 1 -7)
5 lisp>   % Note that X was "destroyed" by GmergeSort.
5 lisp> X
(3 2 1 -7)
6 lisp>
6 lisp>   % Here's IdSort, taking a vector as its argument.
6 lisp> (IdSort '[the quick brown fox jumped over the lazy dog])
[BROWN DOG FOX JUMPED LAZY OVER QUICK THE THE]
7 lisp>
7 lisp>   % Some examples of user defined comparison functions...
7 lisp> (setq X '(("Joe" . 20000) ("Moe" . 21000) ("Larry" . 7000)))
(("Joe" . 20000) ("Moe" . 21000) ("Larry" . 7000))
8 lisp>
8 lisp>   % First, sort the list alphabetically according to name,
8 lisp>   % using a lambda expression as the comparison function.
8 lisp> (Gsort X
8 lisp>     '(lambda (X Y) (string-not-greaterp (car X) (car Y))))
(("Joe" . 20000) ("Larry" . 7000) ("Moe" . 21000))
9 lisp>
9 lisp>   % Now, define a comparison function that compares cdrs of
9 lisp>   % pairs, and returns T if the first is less than or equal
9 lisp>   % to the second.
9 lisp> (de cdr_leq (pair1 pair2)
9 lisp>   (leq (cdr pair1) (cdr pair2)))
CDR_LEQ
10 lisp>
10 lisp>   % Use the cdr_leq function to sort X.
10 lisp> (Gsort X 'cdr_leq)
(("Larry" . 7000) ("Joe" . 20000) ("Moe" . 21000))
```

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, else NIL is returned. Assoc might not detect a poorly formed a-list so an invalid construction may be detected by Car or Cdr.

```
   (De Assoc (U V)
          (Cond ((Null V) Nil)
                ((Atom (Car V))
                 (Error 000 (List V "is a poorly formed alist")))
                ((Equal U (Caar V)) (Car V))
                (T (Assoc U (Cdr V)))))
```

(Atsoc R1:any R2:any): any expr

Scan R2 for pair with Car Eq R1. Eq version of Assoc.

(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.

```
   (De SAssoc (U V FN)
          (Cond ((Null V) (FN))
                ((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 U V)(Cons (Cons (Car U)(Car V))
                               (Pair (Cdr U)(Cdr V))))
               ((Or U V)(Error 000 "Different length lists in PAIR"))
               (T Nil)))
```

5.5. Substitutions

(Subst U:any V:any W:any): any <u>expr</u>

Returns the result of substituting <u>U</u> for all occurrences of <u>V</u> in <u>W</u>. Copies all of <u>W</u> which is not replaced by <u>U</u>. The test used is Equal.

```
(De Subst (U V W)
        (Cond ((Null W) Nil)
              ((Equal V W) U)
              ((Atom W) W)
              (T (Cons (Subst U V (Car W))(Subst U V (Cdr W))))))
```

(SubstIP U:any V:any W:any): any <u>expr</u>

Destructive Subst.

(SubLis X:a-list Y:any): any <u>expr</u>

This performs a series of Substs in parallel. The value returned is the result of substituting the Cdr of each element of the <u>a-list</u> <u>X</u> for every occurrence of the Car part of that element in <u>Y</u>.

```
(De SubLis (X Y)
  (Cond
    ((Null X) Y)
    (T
      (Prog (U)
        (Setq U (Assoc Y X))
        (Return
          (Cond
            (U (Cdr U))
            ((Atom Y) Y)
            (T (Cons (SubLis X (Car Y)) (SubLis X (Cdr Y)))))))))
```

(SublA U:a-list V:any): any <u>expr</u>

Eq version of SubLis; replaces atoms only.

CHAPTER 6
STRINGS AND VECTORS

6.1. Vector-Like Objects

In this chapter, LISP strings, vectors, word-vectors, halfword-vectors, and byte-vectors are described. Each may have several elements, accessed by an integer index. For convenience, members of this set are referred to as x-vectors. X-vector functions also apply to lists. Currently, the index for x-vectors ranges from 0 to an upper limit, called the Size or UpB (upper bound). Thus an x-vector X has 1 + Size(X) elements. Strings index from 0 because they are considered to be packed vectors of bytes. Bytes are 7 bits on the DEC-20 and 8 bits on the VAX.

[??? Note that with new integer tagging, strings are "packed" words, which are special cases of vectors. Should we add byte-vectors too, so that strings are different print mode of byte vector ???]

[??? Size should probably be replaced by UPLIM or UPB. ???]

6.2. Strings

A string is currently thought of as a Byte vector, or a packed integer vector, with elements that are ASCII characters. A string has a header containing its length and perhaps a tag. The next M words contain the 0...Size characters, packed as appropriate, terminated with at least 1 NULL. On the DEC-20, this means that strings have an ASCIZ string starting in the second word. (ASCIZ strings are NULL terminated.)

(MkString UPLIM:integer INITVAL:integer): string expr

Returns a string of characters all initialized to INITVAL, with upper bound UPLIM. So, the returned string contains a total of UPLIM + 1 characters.

(String [ARGS:integer]): string nexpr

Create string of elements from a list of ARGS.

[??? Should we check each arg in 0...127. What about 128 - 255 with 8 bit vectors? ???]

(String 65 66 67) returns "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 heap space.

[??? Should we add GetS, PutS, UpbS, etc ???]

When processing strings it is frequently necessary to be able to specify a particular character. In PSL a character is just its ASCII code representation, but it is difficult to remember the code, and the use of codes does not add to the readability of programs.

(Char U:id): integer macro

The Char macro returns the ASCII code corresponding to its single character-id argument. CHAR also can handle alias's for special characters, remove QUOTE marks that may be needed to pass special characters through the parser, and can accept prefixes to compute lower case, <Ctrl> characters, and <Meta> characters. For example:

```
(Char A)  returns 65
(Char !a) returns 97
(Char (lower a)) returns 97
(Char (control a)) returns 1
(Char (meta (control a))) returns 129, but
(Char (control (meta a))) returns 1
```

"Control" forces the character code into the range 0-31. "Meta" turns on the "meta bit". "Lower" is only well-defined for alphabetic characters. To get lower-case a one may precede the a by "!". See also the sharp-sign macros in Chapter 10.

The following Aliases are defined by PUTing the association under the indicator 'CharConst:

```
DefList('((NULL 8#0)
          (BELL 8#7)
          (BACKSPACE 8#10)
          (TAB 8#11)
          (LF 8#12)
          (EOL 8#12)
          (FF 8#14)
          (CR 8#15)
          (EOF 26)
          (ESC 27)
          (ESCAPE 27)
          (BLANK 32)
          (SPACE 32)
          (RUB 8#177)
          (RUBOUT 8#177)
          (DEL 8#177)
          (DELETE 8#177)), 'CharConst);
```

Users can add new "modifiers" such as META or CONTROL: just hang the appropriate function (from integers to integers) off the char-prefix-function property of the modifier.

6.3. Vectors

A vector is a structured entity in which random item 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 suggested input/output "vector notation" is defined (see Chapter 10).

(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

A similar effect may be obtained in RLISP by using V[INDEX];.

(MkVect UPLIM:integer): vector expr

Defines and allocates space for a vector with UPLIM + 1 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 < -1 or if there is not enough space for a vector of this size:

***** A vector of size UPLIM cannot be allocated

(Make!-Vector UPLIM:integer INITVAL:any): vector expr

Like MkVect but each element is initialized to INITVAL.

(PutV V:vector INDEX:integer VALUE:any): any expr

Stores VALUE in the vector V at position INDEX. VALUE is returned. The type mismatch error may occur. If INDEX does not lie in 0...UPBV(V), an error occurs:

***** INDEX subscript is out of range

:=VALUE;. It is important to use square brackets, i.e. "[]".]

(UpbV U:any): {NIL, integer} expr

Returns the upper limit of U if U is a vector, or NIL if it is not.

(Vector [ARGS:any]): vector nexpr

Create vector of elements from list of ARGS. The vector has N elements, i.e. Size = N - 1, in which N is the number of ARGS.

(CopyVectorToFrom NEW:vector OLD:vector): NEW:vector expr

Move elements, don't recurse.

[???Check size compatibility?]

(CopyVector V:vector): vector expr

Copy to new vector in heap.

The following functions can be used after the FAST!-VECTOR module has been loaded (LOAD FAST!-VECTOR).

(IGetV V:vector INDEX:integer): any — open-compiled, expr

Used the same way as GetV.

(IPutV V:vector INDEX:integer VALUE:any): any — open-compiled, expr

Fast version of PutV.

(ISizeV U:any): {NIL,integer} — open-compiled, expr

Fast version of UpbV.

(ISizeS X:x-vector): integer — open-compiled, expr

Fast version of Size.

(IGetS X:x-vector I:integer): any — open-compiled, expr

Fast version of Indx.

(IPutS X:x-vector I:integer A:any): any — open-compiled, expr

Fast version of SetIndx.

6.4. 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 UPLIM + 1 elements, each initialized to INITVAL.

(Make!-Halfwords UPLIM:integer INITVAL:integer): Halfword-Vector — expr

Defines and allocates space for a Halfword-vector with UPLIM + 1 elements, each initialized to INITVAL.

(Make!-Bytes UPLIM:integer INITVAL:integer): Byte-vector expr

Defines and allocates space for a Byte-Vector with UPLIM + 1 elements, each initialized to INITVAL.

[??? Should we convert elements to true integers when accessing ???]

[??? Should we add GetW, PutW, UpbW, etc ???]

6.5. General X-Vector Operations

(Size X:x-vector): integer expr

Size (upper bound) of x-vector.

(Indx X:x-vector I:integer): any expr

Access the I'th element of an x-vector.

[??? Rename to GetIndex, or some such ???]

Generates a range error if I is outside the range 0 ... Size(X):

***** Index is out of range

(SetIndx X:x-vector I:integer A:any): any expr

Store an appropriate value, A, as the I'th element of an x-vector. Generates a range error if I is outside the range 0...Size(X):

***** Index is out of range

(Sub X:x-vector I1:integer S:integer): x-vector expr

Extract a subrange of an x-vector, starting at I1, 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 of Size HI-LO-1, 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. For example,

```
(Setq A '[0 1 2 3 4 5 6])
(SubSeq A 4 6)
```

returns [4 5].

(SetSubSeq X:x-vector LO:integer HI:integer Y:x-vector): Y:x-vector expr

Y must be of Size HI-LO-1; it must also be of the same type of x-vector as X. Elements LO through HI-1 in X are replaced by elements 0 through Size(Y) of Y. Y is returned and X is changed destructively. If A is "0123456" and B is "abcd", then

```
(SetSubSeq A 3 7 B)
```

returns "abcd". A is "012abcd" and B is unchanged.

(Concat X:x-vector Y:x-vector): x-vector expr

Concatenate two x-vectors. Currently they must be of same type.

[??? Should we do conversion to common type ???]

(TotalCopy S:any): any expr

Returns a unique copy of 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.

6.6. Arrays

Arrays do not exist in PSL as distinct data-types; rather an array macro package is anticipated for declaring and managing multi-dimensional arrays of items, characters and words, by mapping them onto one dimensional vectors.

[??? What operations, how to map, and what sort of checking ???]

6.7. Common LISP String Functions

A Common LISP compatible package of string and character functions has been implemented in PSL, obtained by LOADing the STRINGS module. The following functions are defined from Chapters 13 and 14 of the Common LISP manual [Steele 81]. Char and String are not defined because of PSL functions with the same name.

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.

Recall that in PSL a character is represented as its code, a number in the range 0...127. For an argument to the following character functions give the code or use the **Char** function or the sharp-sign macros in Chapter 10.

(Standard!-CharP C:character): boolean <u>expr</u>

Returns T if the argument is a "standard character", that is, one of the ninety-five ASCII printing characters or <return>.

```
(Standard-CharP (Char A)) returns T
(Standard-CharP (Char !^A)) returns Nil
```

(GraphicP C:character): boolean <u>expr</u>

Returns T if <u>C</u> is a printable character and **Nil** if it is a non-printable (formatting or control) character. The space character is assumed to be graphic.

(String!-CharP C:character): boolean <u>expr</u>

Returns T if <u>C</u> 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 <u>expr</u>

Returns T if <u>C</u> is an alphabetic character.

(UpperCaseP C:character): boolean <u>expr</u>

Returns T if <u>C</u> 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).

(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 symbol is returned. Otherwise an error occurs.

(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 C:character): integer expr

Converts character to its code if C is a one-digit number. If C is larger than one digit, Nil is returned. If C is not numeric, an error message is caused.

`(Char!-Int C:character)`: integer <u>expr</u>

Converts character to integer. This is the identity operation in PSL.

`(Int!-Char I:integer)`: character <u>expr</u>

Converts integer to character. This is the identity operation in PSL.

The string functions follow.

`(RplaChar S:string I:integer C:character)`: character <u>expr</u>

Store a character <u>C</u> in a string <u>S</u> at position <u>I</u>.

`(String!= S1:string S2:string)`: boolean <u>expr</u>

Compares two strings <u>S1</u> and <u>S2</u>, case sensitive. (Substring options not implemented).

`(String!-Equal S1:string S2:string)`: boolean <u>expr</u>

Compare two strings <u>S1</u> and <u>S2</u>, ignoring case, bits and font.

The following string comparison functions are <u>extra-boolean</u>. If the comparison results in a value of T, the first position of inequality in the strings is returned.

`(String!< S1:string S2:string)`: extra-boolean <u>expr</u>

Lexicographic comparison of strings. Case sensitive.

`(String!> S1:string S2:string)`: extra-boolean <u>expr</u>

Lexicographic comparison of strings. Case sensitive.

`(String!<!= S1:string S2:string)`: extra-boolean <u>expr</u>

Lexicographic comparison of strings. Case sensitive.

`(String!>!= S1:string S2:string)`: extra-boolean <u>expr</u>

Lexicographic comparison of strings. Case sensitive.

(String!<!> S1:string S2:string): extra-boolean expr

Lexicographic comparison of strings. Case sensitive.

(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.

(String!-Repeat S:string I:integer): string expr

Appends copy of S to itself total of I-1 times.

(Make!-String I:integer C:character): string expr

Constructs a string with I characters all initialized to C.

(String!-Trim BAG:{list, string} S:string): string expr

Remove leading and trailing characters in BAG from a string S.

```
(String-Trim "ABC" "AABAXYZCB") returns "XYZ"
(String-Trim (List (Char A) (Char B) (Char C))
                                     "AABAXYZCB")
 returns "XYZ"
(String-Trim '(65 66 67) "ABCBAVXZCC") returns "VXZ"
```

(String!-Left!-Trim BAG:{list, string} S:string): string <u>expr</u>

Remove leading characters from string.

(String!-Right!-Trim BAG:{list, string} S:string): string <u>expr</u>

Remove trailing characters from string.

(String!-UpCase S:string): string <u>expr</u>

Copy and raise all alphabetic characters in string.

(NString!-UpCase S:string): string <u>expr</u>

Destructively raise all alphabetic characters in string.

(String!-DownCase S:string): string <u>expr</u>

Copy and lower all alphabetic characters in string.

(NString!-DownCase S:string): string <u>expr</u>

Destructively lower all alphabetic characters in string.

(String!-Capitalize S:string): string <u>expr</u>

Copy and raise first letter of all words in string; other letters in lower case.

(NString!-Capitalize S:string): string <u>expr</u>

Destructively raise first letter of all words; other letters in lower case.

(String!-to!-List S:string): list <u>expr</u>

Unpack string characters into a list.

(String!-to!-Vector S:string): vector <u>expr</u>

Unpack string characters into a vector.

(SubString S:string LO:integer HI:integer): string <u>expr</u>

Same as SubSeq, but the first argument must be a <u>string</u>. Returns a substring of <u>S</u> of Size <u>HI</u> - <u>LO</u> - 1, beginning with the element with index <u>LO</u>.

(String!-Length S:string): integer <u>expr</u>

Last index of a string, plus one.

CHAPTER 7
FLOW OF CONTROL

7.1. Conditionals

7.1.1. Conds and Ifs

(Cond [U:form-list]): any <u>open-compiled, fexpr</u>

The LISP function Cond corresponds to the If statement of most programming languages.

The arguments to Cond have the form:

```
(COND (predicate action action ...)
      (predicate action action ...)
      ...
      (predicate action action ...) )
```

The predicates are evaluated in the order of their appearance until a non-NIL value is encountered. The corresponding actions are evaluated and the value of the last becomes the value of the Cond. If there are no corresponding actions, the value of the predicate is returned.

The actions may also contain the special functions Go, Return, Exit, and Next, subject to the constraints on placement of these functions given in Section 7.2. In these cases, Cond does not have a defined value, but rather an effect. If no predicate is non-NIL, the value of Cond is NIL.

The following Macros are defined in the USEFUL module for convenience.

(If E:form S0:form [S:form]): any macro

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.

```
(IF E S0 S1...Sn)
```

The then-clause S0 is evaluated if and only if the test E is non-NIL, otherwise the else-clauses Si are evaluated, and the last returned. The else clauses are optionally present.

Related macros for common COND forms are WHEN and UNLESS.

(When E:form [S:form]): any macro

```
(WHEN E S1 S2 ... Sn)
```

evaluates the Si and returns the value of Sn if and only if the test E is non-NIL. Otherwise When returns NIL.

(Unless E:form [U:form]): any macro

```
(UNLESS E S1 S2 ... Sn)
```

Evaluates the Si if and only if the test E is NIL. It is equivalent to

```
(WHEN (NOT E) S1 S2 ... Sn)
```

While And and Or are primarily of interest as Boolean connectives, they are often used in LISP as conditionals. For example,

```
(AND (FOO) (BAR) (BAZ))
```

has the same result as

```
(COND ((FOO) (COND ((BAR) (BAZ)))))
```

See Section 2.2.3.

7.1.2. Case and Selectq Statements

PSL provides a numeric case statement, that is compiled quite efficiently; some effort is made to examine special cases (compact vs. non-compact sets of cases, short vs. long sets of cases, etc.). It has mostly been used in SYSLISP mode, but can also be used from LISP mode provided that case-tags are numeric. There is also an FEXPR, Case, for the interpreter.

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

I is meant to evaluate to an integer, and is used as a selector amongst the various Us. Each case-clause has the form (case-expr form) where case-expr has the form:

```
NIL               -> default case
(I1 I2 ... In)    -> where each Ik is an integer or
(RANGE low high)
```

For example:

```
(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 form I, the "key". Each selectq-clause has the form (key-part action action ...). Each key-part is a list of keys, or 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, which would be ambiguous.

After I is evaluated, it is compared against the members of each of the key-part lists in turn. If the key is Eq to any member of a key list, then each of the forms in that selectq-clause are evaluated, and 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. Clearly a T or OTHERWISE clause should be the last of the clauses. If no clause is satisfied Selectq returns NIL.

For example:

```
(SELECTQ (CAR W)
         ((NIL) NIL)
         (END (PRINT 'DONE) 'END)
         ((0 1 2 3 4 5 6 7 8 9) 'DIGIT)
         (OTHERWISE 'OTHER))
```

[??? Perhaps we should move SELECTQ (and define a SELECT) from the COMMON module to the basic system ???]

7.2. Sequencing Evaluation

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

(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 B (the second argument).

[??? Redefine prog2 to take N arguments, return second. ???]

(Prog1 [U:form]): any — macro

Prog1 is a function defined in the USEFUL package. Prog1 evaluates its arguments in order, as ProgN does, 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, 8.3). The Prog's variables are allocated space if the Prog form is applied, and are deallocated if 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. identifiers appearing in the top level of the PROGRAM are labels which can be referred to by Go. The value returned by the **Prog** function is determined by a **Return** function or **NIL** if the **Prog** "falls through".

There are restrictions as to where a number of control functions, such as **Go** and **Return**, may be placed. This is so that they may have only locally determinable effects. Unlike most LISPs, which make this restriction only in compiled code, PSL enforces this restriction uniformly in both compiled and interpreted code. Not only does this help keep the semantics of compiled and interpreted code the same, but we believe it leads to more readable programs. For cases in which a non-local exit is truly required, there are the functions **Catch** and **Throw**, described in Section 7.4.

The functions so restricted are **Go**, **Return**, **Exit**, and **Next**. They must be placed at top-level within the surrounding control structure to which they refer (e.g., the **Prog** which **Return** causes to be terminated), or nested within only selected functions. The functions in which they may be nested (to arbitrary depth) are:
* **ProgN** (compound statement)
* actions of **Conds** (if then else)
* actions in **Cases**

(**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. A **Go** may appear only in the following situations:

a. At the top level of a **Prog** referring to a LABEL that also appears at the top level of the same **Prog**.
b. As the action of a **Cond** item

    i. appearing on the top level of a **Prog**.
    ii. which appears as the action of a **Cond** item to any level.

c. As the last statement of a **ProgN**

    i. which appears at the top level of a **Prog** or in a **ProgN** appearing in the action of a **Cond** to any level subject to the restrictions of b.i, or b.ii.
    ii. within a **ProgN** or as the action of a **Cond** in a **ProgN** to any level subject to the restrictions of b.i, b.ii, and c.i.

If LABEL does not appear at the top level of the **Prog** in which the **Go** appears, an error occurs:

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

If the Go has been placed in a position not defined by rules a-c, another error is detected:

***** Illegal use of GO To LABEL

(Return U:form): None Returned <u>open-compiled, expr</u>

Within a Prog, Return terminates the evaluation of a Prog and returns <u>U</u> as the value of the Prog. The restrictions on the placement of Return are exactly those of Go. Improper placement of Return results in the error:

***** Illegal use of RETURN

7.3. Iteration

(While E:form [S:form]): NIL <u>macro</u>

This is the most commonly used construct for indefinite iteration in LISP. <u>E</u> is evaluated; if non-NIL, the <u>S</u>'s are evaluated from left to right and then the process is repeated. If <u>E</u> evaluates to NIL the While returns NIL. Exit may be used to terminate the While from within the body and to return a value. Next may be used to terminate the current iteration.

(Repeat [S:form E:form]): NIL <u>macro</u>

The <u>S</u>'s are evaluated left to right, and then <u>E</u> is evaluated. This is repeated until the value of <u>E</u> is non-NIL, at which point Repeat returns NIL. Next and Exit may be used in the <u>S</u>'s branch to the next iteration of a Repeat or to terminate one and possibly return a value. Go, and Return may appear in the <u>S</u>'s.

(Next): None Returned <u>open-compiled, restricted, macro</u>

This terminates the current iteration of the most closely surrounding While or Repeat, and causes the next to commence. See the note in Section 7.2 about the lexical restrictions on placement of this construct, which is essentially a GO to a special label placed at the front of a loop construct.

(Exit [U:form]): None Returned <u>open-compiled,restricted, macro</u>

> The <u>U</u>'s are evaluated left to right, the most closely surrounding While or Repeat is terminated, and the value of the last <u>U</u> is returned. With no arguments, NIL is returned. See the note in Section 7.2 about the lexical restrictions on placement of this construct, which is essentially a Return.

While and Repeat each macro expand into a Prog; Next and Exit are macro expanded into a Go and a Return respectively to this Prog. Thus using a Next or an Exit within a Prog within a While or Repeat will result only in an exit of the internal Prog.

7.3.1. For

A simple For construct is available in the basic PSL system; an extended form can be obtained by loading USEFUL. It is planned to make the extended form the version available in the basic system, combining all the features of FOR and ForEach. The basic PSL For provides only the (FROM ..) iterator, and (DO ...) action clause, and uses the ForEach construct for some of the (IN ...) and (ON ...) iterators. Most users should use the full For construct.

(For [S:form]): any <u>macro</u>

> The arguments to For are clauses; each clause is itself a list of a keyword and one or more arguments. The clauses may introduce local variables, specify return values and when the iteration should cease, have side-effects, and so on. Before going further, it is probably best to give some examples.

```
(FOR (FROM I 1 10 2) (DO (PRINT I)))
        Prints the numbers 1 3 5 7 9

(FOR (IN U '(A B C)) (DO (PRINT U)))
        Prints the letters A B C

(FOR (ON U '(A B C)) (DO (PRINT U)))
        Prints the lists (A B C) (B C) and (C)

Finally, the function
(DE ZIP (X Y)
  (FOR (IN U X) (IN V Y)
        (COLLECT (LIST U V))))
```

produces a list of 2 element lists, each consisting of the the corresponding elements of the three lists X, Y and Z. For example,

```
(ZIP '(1 2 3 4) '(A B C) )
```

produces

```
((1 a)(2 b)(3 c))
```

The iteration terminates as soon as one of the (IN ..) clauses is exhausted.

Note that the (IN ...), (ON ...) and (FROM ...) clauses introduce local variables U, V or I, that are referred to in the action clause.

All the possible clauses are described below. The first few introduce iteration variables. Most of these also give some means of indicating when iteration should cease. For example, if a list being mapped over by an In clause is exhausted, iteration must cease. If several such clauses are given in a For expression, iteration ceases when one of the clauses indicates it should, whether or not the other clauses indicate that it should cease.

(IN V1 V2) assigns the variable V1 successive elements of the list V2.

This may take an additional, optional argument: a function to be applied to the extracted element or sublist before it is assigned to the variable. The following returns the sum of the lengths of all the elements of L.

[??? Rather a kludge -- not sure why this is here. Perhaps it should come out again. ???]

```
(DE LENGTHS (L)
  (FOR (IN N L LENGTH)
(COLLECT (LIST N N)))

is the same as

(DE LENGTHS (L)
  (FOR (IN N L)
     (COLLECT
      (LIST (LENGTH N) (LENGTH N))))
)
```

but only calls LENGTH once. Using the (WITH ..) form to introduce a local LN may be clearer.

```
For example,
(SUMLENGTHS
 '((1 2 3 4 5)(a b c)(x y)))
is
((5 5) (3 3) (2 2))
```

(ON V1 V2) assigns the variable V1 successive Cdrs of the list V2.

(FROM VAR INIT FINAL STEP)
is a numeric iteration clause. The variable is first assigned INIT, and then incremented by step until it is larger than FINAL. INIT, FINAL, and STEP are optional. INIT and STEP both default to 1, and if FINAL is omitted the iteration continues until stopped by some other means. To specify a STEP with INIT or FINAL omitted, or a FINAL with INIT omitted, place NIL (the constant -- it cannot be an expression) in the appropriate slot to be omitted. FINAL and STEP are only evaluated once.

(FOR VAR INIT NEXT)
assigns the variable INIT first, and subsequently the value of the expression NEXT. INIT and NEXT may be omitted. Note that this is identical to the behavior of iterators in a Do.

(WITH V1 V2 ... Vn)
introduces N locals, initialized to NIL. In addition, each Vi may also be of the form (VAR INIT), in which case it is initialized to INIT.

(DO S1 S2 ... Sn)
causes the Si's to be evaluated at each iteration.

There are two clauses which allow arbitrary code to be executed before the first iteration, and after the last.

(INITIALLY S1 S2 ... Sn)
causes the Si's to be evaluated in the new environment (i.e. with the iteration variables bound to their initial values) before the first iteration.

(FINALLY S1 S2 ... Sn)
causes the Si's to be evaluated just before the function returns.

The next few clauses build up return types. Except for the RETURNS/RETURNING clause, they may each take an additional argument which specifies that instead of returning the appropriate value, it is accumulated in the specified variable. For example, an unzipper might be defined as

```
  (DE UNZIP (L)
    (FOR (IN U L) (WITH X Y)
      (COLLECT (FIRST U) X)
      (COLLECT (SECOND U) Y)
      (RETURNS (LIST X Y))))
```

This is essentially the opposite of Zip. Given a list of 2 element lists, it unzips them into 2 lists, and returns a list of those 2 lists. For example, (unzip '((1 a)(2 b)(3 c))) returns is ((1 2 3)(a b c)).

(RETURNS EXP)
: causes the given expression to be the value of the For. Returning is synonymous with returns. It may be given additional arguments, in which case they are evaluated in order and the value of the last is returned (implicit ProgN).

(COLLECT EXP)
: causes the successive values of the expression to be collected into a list. Each value is Appended to the end of the list.

(ADJOIN EXP), (ADJOINQ EXP)
: are similar to COLLECT, but a value is added to the result only if it is not already in the list. ADJOIN tests with EQUAL, ADJOINQ tests with EQ.

(CONC EXP)
: causes the successive values to be NConc'd together.

(JOIN EXP) causes them to be appended.

(UNION EXP), (UNIONQ EXP)
: are similar to JOIN, but only add an element to the list if it is not already there. UNION tests with EQUAL, UNIONQ tests with EQ.

(INTERSECTION EXP), (INTERSECTIONQ EXP)
: compute the set of elements that are in all the sets iterated over. With INTERSECTION, elements are the same if EQUAL,

with INTERSECTIONQ they are the same if EQ.

(COUNT EXP)
returns the number of times EXP was non-NIL.

(SUM EXP), (PRODUCT EXP), (MAXIMIZE EXP), and (MINIMIZE EXP)
do the obvious. Synonyms are summing, maximizing, and minimizing.

(MAXIMAL EXP1 EXP2), (MINIMAL EXP1 EXP2)
are more general than maximize and minimize. MAXIMAL determines the greatest value for EXP2 over the iteration, returning the value of EXP1 rather than the value of EXP2. As a particular case it is possible to return the value of an iteration variable for which some function attains a maximum (or minimum) value, e.g. (MAXIMAL x (f x)). As with other kinds of clauses, the user may "accumulate" the value of EXP1 into a variable by supplying a third expression which is the name of a variable.

(ALWAYS EXP)
returns T if EXP is non-NIL on each iteration. If EXP is ever NIL, the loop terminates immediately, no epilogue code, such as that introduced by finally is run, and NIL is returned.

(NEVER EXP)
is equivalent to (ALWAYS (NOT EXP)).

(WHILE EXP) and (UNTIL EXP)
Explicit tests for the end of the loop may be given using (WHILE EXP). The loop terminates if EXP becomes NIL at the beginning of an iteration. (UNTIL EXP) is equivalent to (WHILE (NOT EXP)). Both **While** and **Until** may be given additional arguments; (WHILE E1 E2 ... En) is equivalent to (WHILE (AND E1 E2 ... En)) and (UNTIL E1 E2 ... En) is equivalent to (UNTIL (OR E1 E2 ... En)).

(WHEN EXP)
causes a jump to the next iteration if EXP is NIL.

(UNLESS EXP)
is equivalent to (WHEN (NOT EXP)).

For is a general iteration construct similar in many ways to the LISP Machine and MACLISP Loop construct, and the earlier Interlisp CLISP iteration construct. **For**, however, is considerably simpler, far more "lispy", and somewhat less powerful.

All variable binding/updating still precedes any tests or other code. Also note that all When or Unless clauses apply to all action clauses, not just subsequent ones. This fixed order of evaluation makes For less powerful than Loop, but also keeps it considerably simpler. The basic order of evaluation is

a. bind variables to initial values (computed in the outer environment)

b. execute prologue (i.e., Initially clauses)

c. while none of the termination conditions are satisfied:

    i. check conditionalization clauses (When and Unless), and start next iteration if all are not satisfied.

    ii. perform body, collecting into variables as necessary

    iii. next iteration

d. (after a termination condition is satisfied) execute the epilogue (i.e., Finally clauses)

For does all variable binding/updating in parallel. There is a similar macro, For*, which does it sequentially.

(For!* [S:form]): any macro

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:

```
(Prog (I)
  (Setq I 0)
  (Return
    (Mapcar L
      (Function (Lambda (X)
                  (Progn
                    (Setq I (Plus I 1))
                    (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 (CARs) 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)
```

(Map X:list FN:function): NIL expr

Applies FN to successive Cdr segments of X. NIL is returned. This is equivalent to:

```
   (FOREACH u ON x DO (FN u))
```

(MapC X:list FN:function): NIL expr

FN is applied to successive Car segments of list X. NIL is returned. This is equivalent to:

```
   (FOREACH u IN x DO (FN u))
```

(MapCan X:list FN:function): list expr

A concatenated list of FN applied to successive Car elements of X is returned. This is equivalent to:

```
   (FOREACH u IN x CONC (FN u))
```

(MapCar X:list FN:function): list   expr

Returned is a constructed list, the elements of which are FN applied to each Car of list X. This is equivalent to:

```
(FOREACH u IN x COLLECT (FN u))
```

(MapCon X:list FN:function): list   expr

Returned is a concatenated list of FN applied to successive Cdr segments of X. This is equivalent to:

```
(FOREACH u ON x CONC (FN u))
```

(MapList X:list FN:function): list   expr

Returns a constructed list, the elements of which are FN applied to successive Cdr segments of X. This is equivalent to:

```
(FOREACH u ON x COLLECT (FN u))
```

7.3.3. Do

The MACLISP style Do and Let are now partially implemented in the USEFUL module.

(Do A:list B:list [S:form]): any   macro

The Do macro is a general iteration construct similar to that of LISPM and friends. However, it does differ in some details; in particular it is not compatible with the "old style Do" of MACLISP, nor does it support the "no end test means once only" convention. Do has the form

```
(DO (I1 I2 ... In)
    (TEST R1 R2 ... Rk)
    S1
    S2
    ...
    Sm)
```

in which there may be zero or more I's, R's, and S's. In general the I's have the form

```
(var init step)
```

On entry to the Do form, all the inits are evaluated, then the variables are bound to their respective inits. The test is evaluated, and if non-NIL the form evaluates the R's and returns the value of the last one. If none are supplied it returns NIL. If the test evaluates to NIL the S's are evaluated, the variables are assigned the values of their respective steps in parallel, and the test evaluated again. This iteration continues until test evaluates to a non-NIL value. Note that the inits are evaluated in the surrounding environment, while the steps are evaluated in the new environment. The body of the Do (the S's) is a Prog, and may contain labels and Go's, though use of this is discouraged. This may be changed at a later date. Return used within a Do returns immediately without evaluating the test or exit forms (R's).

There are alternative forms for the I's: If the step is omitted, the variable's value is left unchanged. If both the init and step are omitted or if the I is an <u>id</u>, it is initialized to NIL and left unchanged. This is particularly useful for introducing dummy variables which are SetQ'd inside the body.

(Do!* A:list B:list [C:form]): any <u>macro</u>

Do!* is like Do, except the variable bindings and updatings are done sequentially instead of in parallel.

(Do-Loop A:list B:list C:list [S:form]): any <u>macro</u>

Do-Loop is like Do, except that it takes an additional argument, a prologue. The general form is

```
(DO-LOOP (I1 I2 ... In)
    (P1 P2 ... Pj)
    (TEST R1 R2 ... Rk)
    S1
    S2
    ...
    Sm)
```

This is executed just like the corresponding Do, except that after the bindings are established and initial values assigned, but before the test is first executed the P's are evaluated, in order. Note that the P's are all evaluated exactly once (assuming that none of the P's errs out, or otherwise throws to a surrounding context).

(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.

(Let A:list [B:form]): any macro

Let is a macro giving a more perspicuous form for writing lambda expressions. The basic form is

```
(LET ((V1 I1) (V2 I2) ...(Vn In)) S1 S2 ...  Sn)
```

The I's are evaluated (in an unspecified order), and then the V's are bound to these values, the S's evaluated, and the value of the last is returned. Note that the I's are evaluated in the outer environment before the V's are bound.

Note: the id LET conflicts with a similar construct in RLISP and REDUCE

(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.

7.4. Non-Local Exits

One occasionally wishes to discontinue a computation in which the lexical restrictions on placement of Return are too restrictive. The non-local exit constructs Catch and Throw exist for these cases. 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. Under some conditions, these functions are invaluable. Under others, they can wreak havoc.

(Catch TAG:id [FORM:form]): any Open-Compiled, fexpr

Catch evaluates the TAG and then calls Eval on the FORMs in a protected environment. If during this evaluation (Throw TAG VAL) occurs, Catch immediately returns VAL. If no Throw occurs, the value of the last FORM is returned. Note that in general only Throws with the same TAG are caught. Throws whose TAG is not Eq to that of Catch are passed on out to surrounding Catches. A TAG of NIL, however, is special. (Catch NIL FORM) catches any Throw. The tag of NIL serves to match any tag specified to Throw.

ThrowSignal!* [Initially: NIL] global

ThrowTag!* [Initially: NIL] global

The FLUID variables ThrowSignal!* and ThrowTag!* may be interrogated to find out if the most recently evaluated Catch was Thrown to, and what tag was passed to the Throw. ThrowSignal!* is Set to NIL upon normal exit from a Catch, and to T upon normal exit from Throw. ThrowTag!* is Set to the first argument passed to the Throw. (Mark a place to Throw to, Eval FORM.)

(Throw TAG:id VAL:any): None Returned expr

This passes control to the closest surrounding Catch with an Eq or null TAG. If there is no such surrounding Catch it is an error in the context of the Throw. That is, control is not Thrown to the top level before the call on Error. (Non-local Goto.)

Some examples:

```
With
(DE DOIT (x)
 (COND ((EQN x 1) 100)
       (T (THROW 'FOO 200))))

(CATCH 'FOO (DOIT 1) (PRINT "NOPE") 0)
        will continue and execute the PRINT statement
        and return 0
while

(CATCH 'FOO (DOIT 2) (PRINT "NOPE") 0)

will of course THROW, returning 200 and not executing
the last forms.
```

A common problem people encounter is how to pass arguments and/or computed functions or tags into CATCH for protected evaluation. The following examples should illustrate. Note that TAG is quoted, since it is evaluated before use in CATCH and THROW.

```
(DE PASS-ARGS(X1 X2)
   (CATCH 'FOO (FEE (PLUS2 X1 X2) (DIFFERENCE X1 X2))))
```

This is simple, because CATCH compiles open. No FLUID declarations or LIST building is needed, as in previous versions of PSL. An explicit Apply must be used for a function argument; usually, the APPLY will compile open, with no overhead:

```
(DE PASS-FN(X1 FN)
   (CATCH 'FOO (APPLY FN (LIST X1))))
```

The following MACROs are provided to aid in the use of Catch and Throw with a NIL tag, by examining the ThrowSignal!* and ThrowTag!*:

(Catch!-All FN:function [FORM:form]): any macro

This issues a (Catch NIL ...); if a Throw was actually done, the function FN is applied to the two arguments ThrowTag!* and the value returned by the throw. Thus FN is applied only if a Throw was executed.

(Unwind!-All FN:function [FORM:form]): any macro

This issues a (Catch NIL ...). The function FN is always called, and applied to the two arguments ThrowTag!* and the value returned by the throw. If no Throw was done then FN is called on NIL and the value returned.

(Unwind!-Protect F:form [C:form]): any macro

The idea is to execute the "protected" form, F, and then run some "clean-up" forms C even if a Throw (or Error) occurred during the evaluation of F. This issues a (Catch NIL ...), the cleanup forms are then run, and finally either the value is returned if no Throw occurred, or the Throw is "re-thrown" to the same tag.

A common example is to ensure a file be closed after processing, even if an error or throw occurred:

```
(SETQ chan (OPEN file ....))
(UNWIND-PROTECT (process-file)
                (CLOSE chan))
```

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 12).

!$UNWIND!-PROTECT!$
A special TAG placed to ensure that ALL throws pause at the UNWIND-PROTECT "mark".

!$PROG!$ Used to communicate between interpreted PROGs, GOs and RETURNs.

CHAPTER 8
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. A warning message is given for compatibility. The first major section in this chapter describes how to define new functions; the second describes the binding of variables in PSL.

8.2. 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 of 15 arguments. 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. Future versions of PSL will check this consistency.

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 9 provide precise details.

8.2.1. 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 (#<Code:a nnnn>, where a is the number of arguments of the function, and nnnn is the function's entry point, (given in octal on the DEC-20 and VAX). A code-pointer may not be created by Compress. (See Chapter 10 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 (on the DEC-20 and VAX, compiled code is not relocated, so code-pointers do not change). A code-pointer may be stored using PutD, Put, SetQ and the like or by being bound to a variable. It may be checked for equivalence by Eq. The value may be checked for being a code-pointer by the CodeP function.

8.2.2. 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 code-pointer or is compiled (i.e., !*COMP=T as the function was defined), a special instruction to jump to the start of the code is placed in the function cell. If it is a lambda, the lambda expression is saved on the property list under the indicator !*LAMBDALINK and a call to an interpreter function (LambdaLink) is placed in the function cell.

The TYPE is recorded on the property list of FNAME if it is not an expr.

[??? We need to add code to check that the the arglist has no more than 15 arguments for exprs, 1 argument for fexprs and macros, and ??? for nexprs. Declaration mechanisms to avoid overhead also need to be available. (In fact are available for the compiler, although still poorly documented.) When should we expand macros? ???]

After using PutD on FNAME, GetD returns a pair of the the FNAME's (TYPE . BODY).

The GlobalP predicate returns T if queried with the defined function's name. If the function FNAME has already been declared as a GLOBAL or FLUID variable the warning:

*** FNAME is a non-local variable

occurs, but the function is defined. If function FNAME is already defined, a warning message appears:

*** Function FNAME has been redefined

Note: All function types may be compiled.

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 if !*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 10. 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 defined in its arguments 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, particularly FLUID type access.

(GetD U:any): {NIL, pair} expr

If U is not the name of a defined function, NIL is returned. If U is a defined function then the pair ({expr, fexpr, macro, nexpr} . {code-pointer, lambda}) is returned.

(CopyD NEW:id OLD:id): NEW:id expr

The function body and type for NEW become the same as 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. The function type attribute of U is removed from the property list of U.

8.2.3. Function Definition in LISP Syntax

The functions De, Df, Dn, Dm, and Ds are most commonly used in the LISP syntax form of PSL. They are difficult to use from RLISP as there is not a convenient way to represent the argument list. The functions are compiled if the compiler is loaded and the GLOBAL !*COMP is 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[1] is used as the body of the function.

Previous definitions of the function are lost. The name of the defined function, FNAME, is returned.

(Df FNAME:id PARAM:id-list FN:any): id macro

Defines the function named FNAME, of type fexpr. 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.

(Dn FNAME:id PARAM:id-list FN:any): id macro

Defines the function named FNAME, of type nexpr. 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.

[1]Or the compiled code pointer for the lambda expression if the compiler is on.

Previous definitions of the function are lost. The name of the defined function, FNAME, is returned.

(Dm MNAME:id PARAM:id-list FN:any): id macro

Defines the function named FNAME, of type macro. 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.

(Ds SNAME:id PARAM:id-list FN:any): id macro

Defines the smacro SNAME. Smacros are actually a syntactic notation for a special class of macros, those that essentially treat the macro's argument as a list of arguments to be substituted into the body of the expression and then expanded in line, rather than using the computational power of the macro to customize code. Thus they are a special case of defmacro. See also the BackQuote facility.

For example:

```
To make a substitution macro for
FIRST ->CAR we could say

(DM FIRST(X)
    (LIST 'CAR (CADR X)))

Instead the following is clearer

(DS FIRST(X)
     (CAR X))
```

The following (and other) macro utilities are in the file PU:USEFUL.SL; use (LOAD USEFUL) to access.[2]

[2]Useful was written by D. Morrison.

(DefMacro A:id B:form [C:form]): id macro

DefMacro is a useful tool for defining macros. A DefMacro form looks like

```
(DEFMACRO <NAME> <PATTERN> <S1> <S2> ... <Sn>)
```

The <PATTERN> is an S-expression made of pairs and ids. It is matched against the arguments of the macro much like the first argument to DeSetQ. All of the non-NIL ids in <pattern> are local variables which may be used freely in the body (the <Si>). If the macro is called the <Si> are evaluated as in a ProgN with the local variables in <pattern> appropriately bound, and the value of <Sn> is returned. DefMacro is often used with BackQuote.

(DefLambda): macro

DefLambda defines a macro much like a substitution macro (smacro) except that it is a lambda expression. Thus, modulo redefinability, it has the same semantics as the equivalent expr. It is mostly intended as an easy way to open compile things. For example, we would not normally want to define a substitution macro for a constructor (NEW-FOO X) which maps into (CONS X X), in case X is expensive to compute or, far worse, has side effects. (DEFLAMBDA NEW-FOO (X) (CONS X X)) defines it as a macro which maps (NEW-FOO (SETQ BAR (BAZ))) to ((LAMBDA (X) (CONS X X)) (SETQ BAR (BAZ))).

8.2.4. BackQuote

Note that the special symbols described below only work in LISP syntax, not RLISP. In RLISP you may simply use the functions BackQuote, UnQuote, and UnQuoteL. Load USEFUL to get the BackQuote function.

The backquote symbol "`" (accent grave) is a Read macro which introduces a quoted expression which may contain the unquote symbols comma "," and comma-atsign ",@". An appropriate form consisting of the unquoted expression calls to the function Cons and quoted expressions are produced so that the resulting expression looks like the quoted one except that the values of the unquoted expressions are substituted in the appropriate place. ",@" splices in the value of the subsequent expression (i.e. strips off the outer layer of parentheses). Thus

```
'(a (b ,x) c d ,@x e f)
```

is equivalent to

```
(cons 'a (cons (list 'b x) (append '(c d) (append x '(e f)))))
```

In particular, if x is bound to (1 2 3) this evaluates to

```
(a (b (1 2 3)) c d 1 2 3 e f)
```

(BackQuote A:form): form   macro

Function name for back quote ' (accent grave).

(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.

8.2.5. 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 macros, and only those specified are expanded. MacroExpand is in the USEFUL package.

8.2.6. Low Level Function Definition Primitives

The following functions are used especially by PutD and GetD, defined above in Section 8.2.2, and by Eval and Apply, defined in Chapter 9.

(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: Undefined functions actually call a special function, UndefinedFunction, that invokes Error. FUnBoundP defines "unbound" to mean "calls UndefinedFunction".

(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 an error function, UndefinedFunction, in the function cell of U.

(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 codepointer C. This integer, or NIL is returned.

[??? Should be extended for nexprs and declared exprs. ???]

8.2.7. Function Type Predicates

See Section 8.2 for a discussion of the function types available in PSL.

(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.

8.3. 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 LOCAL variables in Prog's. 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 GLOBAL variables may never be rebound. Access is to the value bound to the variable. The scope of a GLOBAL variable is universal. Variables declared GLOBAL may not appear as parameters in lambda expressions or as Prog form variables. A variable must be declared GLOBAL prior to its use as a GLOBAL variable since the default type for undeclared variables is FLUID. Note that the interpreter does not stop one from rebinding a global variable. The compiler will issue a warning in this situation.

8.3.1. Binding Type Declaration

(Fluid IDLIST:id-list): NIL expr

The ids in IDLIST are declared as FLUID type variables (ids not previously declared are initialized to NIL). Variables in IDLIST already declared FLUID are ignored. Changing a variable's type from GLOBAL to FLUID is not permissible and results in the error:

***** 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 already declared GLOBAL are ignored. Changing a variable's type from FLUID to GLOBAL is not permissible and results in the error:

***** ID cannot be changed to GLOBAL

(UnFluid IDLIST:id-list): NIL expr

The variables in IDLIST which have been declared as FLUID variables are no longer considered as FLUID variables. Others are ignored. This affects only compiled functions, as free variables in interpreted functions are automatically considered FLUID (see [Griss 81]).

8.3.2. Binding Type Predicates

(FluidP U:any): boolean expr

If U is FLUID (by declaration only), T is returned; otherwise, NIL is returned.

(GlobalP U:any): boolean expr

If U has been declared GLOBAL or is the name of a defined function, T is returned; else NIL is returned.

(UnBoundP U:id): boolean expr

Tests whether U has no value.

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.

[??? Are these correct when Environments are managed correctly ???]

(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)

CHAPTER 9
THE INTERPRETER

9.1. Evaluator Functions Eval and Apply

The PSL evaluator uses an identifier's function cell (SYMFNC(id#) which is directly accessible from kernel functions only) to access the address of the code for executing the identifier's function definition, as described in chapter 8. The function cell contains either the entry address of a compiled function, or the address of a support routine that either signals an undefined function or calls the lambda interpreter. The PSL model of a function call is to place the arguments (after treatment appropriate to function type) in "registers", and then to jump to or call the code in the function cell.

Expressions which can be legally evaluated are called forms. They are restricted S-expressions:

```
form ::=  id
        | constant
        | (id form ... form)     % Normal function call
        | (special . any)        % Special cases: COND, PROG, etc.
                                 % usually fexprs or macros.
```

The definitions of Eval and Apply may clarify which expressions are forms.

In Eval, Apply, and the support functions below, ContinuableError is used to indicate malformed lambda expressions, undefined functions or mismatched argument numbers; the user is permitted to correct the offending expression or to define a missing function inside a Break loop.

The functions Eval and Apply are central to the PSL interpreter. Since their efficiency is important, some of the support functions they use are hand-coded in LAP. The functions LambdaApply, LambdaEvalApply, CodeApply, CodeEvalApply, and IDApply1 are support functions for Eval and Apply. CodeApply and CodeEvalApply are coded in LAP.

IDApply1 is handled by the compiler.

(Eval U:form): any   expr

The value of the form U is computed. The following is an approximation of the real code, leaving out some implementation details.

```
(DE EVAL (U)
  (PROG (FN)
    (COND
      ((IDP U) (RETURN (VALUECELL U))))
    % ValueCell  returns the contents of Value Cell if ID is
    % BoundP, else signals unbound error.
    (COND ((NOT (PAIRP U)) (RETURN U)))

    % This is a "constant" which EVAL's to itself
    (COND
      ((EQCAR (CAR U) 'LAMBDA)
        (RETURN
          (LAMBDAEVALAPPLY (CAR U) (CDR U)))))

    % LambdaEvalApply applies the lambda- expression Car U to the
    % list containing the evaluation of each argument in Cdr U.
    (COND
      ((CODEP (CAR U))
        (RETURN (CODEEVALAPPLY (CAR U) (CDR U)))))

    % CodeEvalApply applies the function with code-pointer Car U
    % to the list containing the evaluation of each argument in
    % Cdr U.
    (COND
      ((NOT (IDP (CAR U)))
        (RETURN
          % permit user to correct U, and reevaluate.
          (CONTINUABLEERROR 1101
            "Ill-formed expression in EVAL" U))))

    (SETQ FN (GETD (CAR U)))
    (COND
      ((NULL FN)
        % user might define missing function and retry
        (RETURN
           (CONTINUABLEERROR 1001 "Undefined function EVAL" U))))

    (COND
      ((EQ (CAR FN) 'EXPR)
        (RETURN
          (COND
            ((CODEP (CDR FN))
              % CodeEvalApply applies the function with
              % codepointer Cdr FN to the list containing the
              % evaluation of each argument in Cdr U.
              (CODEEVALAPPLY (CDR FN) (CDR U)))
```

```
                (T
                  (LAMBDAEVALAPPLY
                    (CDR FN) (CDR U))))))

     % LambdaEvalApply applies the lambda-expression Cdr FN to the
     % list containing the evaluation of each argument in Cdr U.
     (COND
       ((EQ (CAR FN) 'FEXPR)
         % IDApply1 applies the fexpr Car U to the list of
         % unevaluated arguments.
         (RETURN (IDAPPLY1 (CDR U) (CAR U))))

       ((EQ (CAR FN) 'MACRO)
         % IDApply1 first expands the macro call U and then
         % evaluates the result.
         (RETURN (EVAL (IDAPPLY1 U (CAR U)))))

       ((EQ (CAR FN) 'NEXPR)
         % IDApply1 applies the nexpr Car U to the list obtained
         % by evaluating the arguments in Cdr U.
         (RETURN (IDAPPLY1 (EVLIS (CDR U)) (CAR U)))))))
```

(Apply FN:{id,function} ARGS:form-list): any   expr

Apply allows one to make an indirect function call. It returns the value of FN with actual parameters ARGS. The actual parameters in ARGS are already in the form required for binding to the formal parameters of FN. PSL permits the application of macros, nexprs and fexprs; the effect is the same as (Apply (Cdr (GetD FN)) ARGS); i.e. no fix-up is done to quote arguments, etc. as in some LISPs. A call to Apply using List on the second argument [e.g. (Apply F (List X Y))] is compiled so that the list is not actually constructed.

The following is an approximation of the real code, leaving out implementation details.

```
(DE APPLY (FN ARGS)
  (PROG (DEFN)
    (COND
      ((CODEP FN)
        % Spread the ARGS into the registers and transfer to the
        % entry point of the function.
        (RETURN (CODEAPPLY FN ARGS)))

      ((EQCAR FN 'LAMBDA)
        % Bind the actual parameters in ARGS to the formal
        % parameters of the lambda expression If the two lists
        % are not of equal length then signal
        % (CONTINUABLEERROR 1204
        %         "Number of parameters do not match"
        %         (CONS FN ARGS))

        (RETURN (LAMBDAAPPLY FN ARGS)))

      ((NOT (IDP FN))
        (RETURN (CONTINUABLEERROR 1104
                "Ill-formed function in APPLY"
                (CONS FN ARG))))

      ((NULL (SETQ DEFN (GETD FN)))
        (RETURN (CONTINUABLEERROR 1004
                "Undefined function in Apply"
                (CONS FN ARGS))))

      (T
        % Do EXPR's, NEXPR's, FEXPR's and MACRO's alike, as
        % EXPR's
        (RETURN (APPLY (CDR DEFN) ARGS))))))
```

[??? Instead, could check for specific function types in Apply ???]

9.2. Support Functions for Eval and Apply

(EvLis U:any-list): any-list expr

EvLis returns a list of the evaluation of each element of U.

(LambdaApply FN:lambda, U:any-list): any expr

Checks that FN is a legal lambda, binds the formals of the lambda using LBind1 to the arguments in U, and then uses EvProgN to evaluate the forms in the lambda body. Finally the formals are unbound, using UnBindN, and the result returned.

(LambdaEvalApply FN:lambda, U:form-list): any expr

Essentially LambdaApply(FN,EvLis(U)), though done more efficiently.

(CodeApply FN:code-pointer, U:any-list): any expr

Efficiently spreads the arguments in U into the "registers", and then transfers to the starting address referred to by FN

(CodeEvalApply FN:code-pointer, U:any-list): any expr

Essentially CodeApply(FN,EvLis(U)), though more efficient.

The following entry points are used to get efficient calls on named functions, and are open compiled.

(IdApply0 FN:id): any expr

(IdApply1 A1:form, FN:id): any expr

(IdApply2 A1:form, A2:form, FN:id): any expr

(IdApply3 A1:form, A2:form, A3:form, FN:id): any expr

(IdApply4 A1:form, A2:form, A3:form, A4:form, FN:id): any expr

(EvProgN U:form-list): any expr

Evaluates each form in U in turn, returning the value of the last. Used for various implied ProgNs.

9.3. Special Evaluator Functions, Quote, and Function

(Quote U:any): any fexpr

Returns U. Thus the argument is not evaluated by Eval.

(MkQuote U:any): list expr

(MkQuote U) returns (List 'QUOTE U)

(Function FN:function): function fexpr

The function FN is to be passed to another function. If FN is to have side effects its free variables must be FLUID or GLOBAL. Function is like Quote but its argument may be affected by compilation.

[??? Add FQUOTE, and make FUNCTION become CLOSURE ???]

See also the discussion of Closure and related functions in Section 8.4.

9.4. Support Functions for Macro Evaluation

(Expand L:list, FN:function): list expr

FN is a defined function of two arguments to be used in the expansion of a macro. Expand returns a list in the form:

(FN L[0] (FN L[1] ... (FN L[n-1] L[n]) ...))

"n" is the number of elements in L, L[i] is the i'th element of L.

```
(DE EXPAND (L FN)
   (COND ((NULL (CDR L)) (CAR L))
         (T (LIST FN (CAR L) (EXPAND (CDR L) FN)))))
```

[??? Add RobustExpand (sure!) (document) ???]

[??? Add an Evalhook and Apply hook for CMU toplevel (document) ???]

CHAPTER 10
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:

* Ids begin with a letter or any character preceded by an escape character. They may contain letters, digits and escaped characters. Ids may also start with a digit, if the first non-digit following is a plus sign, minus sign, or letter other than "b" or "e". This is to allow identifiers such as "1+" which occur in some LISPs. Finally, a string of characters bounded by the IDSURROUND character is treated as an id.

  If !*Raise is non-NIL, unescaped lower case letters are folded to upper case. The maximum size of an id (or any other token) is currently 5000 characters.

  Note: Using lower case letters in identifiers may cause portability problems. Lower case letters are automatically converted to upper case if the !*RAISE switch is T. This case conversion is done only for id input, not for single character or string input.

  [??? Can we retain input Case, but Compare RAISEd ???]

  The following examples show identifiers in a form accepted by the LISP scan table. Note that most characters are treated as letters by the LISP scan table, while they are treated as delimiters by the RLISP scan table.

- ThisIsALongIdentifier

- THISISALONGIDENTIFIER

- ThisIsALongIdentifierAndDifferentFromTheOther

- this_is_a_long_identifier_with_underscores

- this!_is!_a!_long!_identifier!_with!_underscores

- an-identifier-with-dashes

- *RAISE

- !2222

* 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 #.[1] 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)

  [??? Should we permit trailing . in integers for compatibility with some LISPs and require digits on each side of . for floats ???]

[1]Octal numbers can also be written as a string of digits followed by the letter "B". This "feature" may be removed in the future.

* 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 octal-address>.

* Anything else is printed as #<Unknown:nnnn>, where nnnn is the octal 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 LISP printing function. For well-formed, non-circular (non-self-referential) 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.

(Print U:any): U:any expr

(ChannelPrint CHAN:io-channel U:any): U:any expr

Display U using ChannelPrin1 and terminate line using ChannelTerpri.

10.3.2. Whitespace Printing Functions

(TerPri): NIL expr

(ChannelTerPri CHAN:io-channel): NIL expr

Terminate OUTPUT line on channel CHAN, and reset the POSN counter to 0.

(Spaces N:integer): NIL expr

(ChannelSpaces CHAN:io-channel N:integer): NIL <u>expr</u>

ChannelPrin2 <u>N</u> spaces. Will continue across multiple lines if <u>N</u> is greater than the number of positions in the output buffer. (See POSN and LINELENGTH)

(Tab N:integer): NIL <u>expr</u>

(ChannelTab CHAN:io-channel N:integer): NIL <u>expr</u>

Move to position <u>N</u> on channel <u>CHAN</u>, emitting spaces as needed. Calls ChannelTerPri if past column <u>N</u>.

10.3.3. Formatted Printing

(PrintF FORMAT:string [ARGS:any]): NIL <u>expr</u>

(ChannelPrintF CHAN:io-channel FORMAT:string [ARGS:any]): NIL <u>expr</u>

ChannelPrintF is a simple routine for formatted printing, similar to the function with the same name in the C language[22]. <u>FORMAT</u> is either a LISP or SYSLISP <u>string</u>, which is printed on the output channel <u>CHAN</u>. However, if a % is encountered in the <u>string</u>, the character following it is a formatting directive, used to interpret and print the other arguments to ChannelPrintF in order. The following format characters are currently supported:

* For LISP tagged items, use:

%p	print the next argument as a LISP item, using Prin1
%w	print the next argument as a LISP item, using Prin2
%r	print the next argument as a LISP item, using ErrPrin (Ordinarily Prin2 "'"; Prin1 Arg; Prin2 "'")
%l	same as %w, except lists are printed without top level parens; NIL is printed as a blank

* Control formats:

%b	take next argument as an integer and print that many blanks
%f	"fresh-line", print an end-of-line character if not at the beginning of the output line (does not use a matching

	argument)
%n	print end-of-line character (does not use a matching argument)
%t	take the next argument as an integer, and ChannelTab to that position

* The following are NOT for use from ordinary LISP programs. For SYSLISP arguments **only**, use:

%d	print the next argument as a decimal integer
%o	print the next argument as an octal integer
%x	print the next argument as a hexadecimal integer
%c	print the next argument as a single character
%s	print the next argument as a string

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 the FORMAT expects and the number given. If the number given is less than the number in the FORMAT string, then 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.

`(RPrint U:form): NIL` expr

Print in RLISP format. Autoloading.

`(PrettyPrint U:form): U` expr

Prettyprints U. Autoloading. This is a rather powerful utility, unfortunately not properly documented.

10.3.4. The Fundamental Printing Function

`(WriteChar CH:character): character` expr

(ChannelWriteChar CHANNEL:io-channel CH:character): character expr

Write one character to CHANNEL. All output is defined in terms of this function. If CH is equal to char EOL (ASCII LF, 8#12) the line counter POSN associated with CHANNEL is set to zero. Otherwise, it is increased by one. The writing function associated with CHANNEL is called with CHANNEL and CH as its arguments.

```
(de WRITECHAR (CH)
    (CHANNELWRITECHAR OUT!* CH))
```

10.3.5. Additional Printing Functions

(Prin2L L:any): L expr

Prin2, except that a list is printed without the top level parens.

(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

Same function as Prin2.

(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 instead of using the currently selected output channel, ERROUT!* is used. Also, an end-of-line character is always printed after the message, and an end-of-line character is printed before the message if the line position of ERROUT!* is greater than zero.

(Eject): NIL expr

Skip to top of next output page on current output channel.

(ChannelEject CHAN:io-channel): NIL expr

Skip to top of next output page on channel CHAN.

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, e.g. 8#20 is"20" in base 8, or 16.

(Posn): integer expr

(ChannelPosn CHAN:io-channel): integer expr

Returns number of characters output on this line (i.e. POSN counts since last Terpri) on this channel.

(LPosn): integer expr

(ChannelLPosn CHAN:io-channel): integer expr

Returns number of lines output on this page (i.e. LPosn counter since last Eject or formfeed character) on this channel.

(LineLength LEN:{integer, NIL}): integer expr

(ChannelLineLength CHAN:io-channel LEN:{integer, NIL}): integer expr

Set maximum output line length on CHAN if a positive integer, returning previous value. If NIL just return previous value. Controls the insertion of automatic Terpri's. If LEN is 0, an EOL character will not be inserted.

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 debugging or dealing with large or deep objects. These variables affect the functions Prin1, Prin2, PrinC, Print, and PrintF (and the corresponding Channel functions). The documentation of these variables is from the Common Lisp Manual.

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, indicating the maximum level to be printed. 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, pair-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.

ChannelRead uses the ChannelReadToken function, with tokens scanned according to the "Lisp scan table". The user can define similar read functions for use with other scan tables. ChannelRead uses the Read macro mechanism to do S-expression parsing. See Section 10.4.5 for more information on read macros and how to add extensions. The following read macros are defined initially:

(	Starts a scan collecting S-expressions according to list or dot notation until terminated by a). A pair or list is returned.
[	Starts a scan collecting S-expressions according to vector notation until terminated by a]. A vector is returned.
'	Calls Read to get an S-expression, x, and then returns the list (Quote x).
(CHAR EOF)	Generates an error when still inside an S-expression:

***** Unexpected EOF while reading on channel

. Otherwise the value of !$EOF!$ is returned. (Actually the value is EQ to the initial value of !$EOF!$.)

The USEFUL library defines several MACLISP style sharp sign read macros. Note that these only work with the LISP reader, not RLISP. Those currently included are

#'	this is like the quote mark ' but is for FUNCTION instead of QUOTE.
#/	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 the CHAR macro, described in the PSL manual. Not that the argument is raised, if *RAISE is non-nil. For example, #\a = #\A = 65, while #\!a = #\(lower a) = 97.
#.	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)
#+	This reads two expressions, and passes them to the if_system macro. That is, the first should be a system name, and if that is the current system the second argument is returned by the reader. If not, the next expression is returned.
#-	#- is similar, but causes the second arg to be returned only if it is NOT the current system.

10.4.2. Reading Single Characters

(ReadChar): character <u>expr</u>

(ChannelReadChar CHANNEL:io-channel): character <u>expr</u>

Reads one character (an <u>integer</u>) from <u>CHANNEL</u>. All input is defined in terms of this function. If <u>CHANNEL</u> 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 <u>CHANNEL</u>, the buffer is emptied (set to zero) and the value returned. Otherwise, the reading function associated with <u>CHANNEL</u> is called with <u>CHANNEL</u> as argument, and the value it returns is returned by ChannelReadChar.

***** Channel not open

***** Channel open for write only

(ReadCH): id <u>expr</u>

(ChannelReadCH CHAN:io-channel): id <u>expr</u>

Like ChannelReadChar, but returns the <u>id</u> for the character rather than its ASCII code.

(UnReadChar CH:character): Undefined <u>expr</u>

(ChannelUnReadChar CHAN:io-channel CH:character): Undefined <u>expr</u>

The input backup function. <u>CH</u> is deposited in the backup buffer associated with <u>CHAN</u>. This function should be only called after ChannelReadChar is called, 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 LISP token scanner. The value returned is a LISP item corresponding to the next token from the input stream. Ids are interned, unless the FLUID variable !*COMPRESSING is non-NIL. The GLOBAL variable TOKTYPE!* is set to:

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!*
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. (Not called ReadToken for historical reasons.)

[??? Should we bind CurrentScanTable!* for this function too ???]

(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!* to determine the current indicator.

10.4.4. Reading Entire Lines

Two functions exist for reading entire lines.

(ReadLine): string <u>expr</u>

(ChannelReadLine CHAN:io-channel): string <u>expr</u>

ReadLine and ChannelReadLine read everything from the current position of the scanner to the next EOL character.

It is frequently used as in the following example.

```
(de foo ()
   (prog (promptstring*)
         (readline)
         (setq promptstring* "-->")
         (return (readline))))
```

When one runs foo, the value of promptstring* is printed and then one types some characters and ends the line. The string is returned.

```
4 lisp> (foo)
-->abcd
"abcd"
```

10.4.5. Read Macros

A function of two arguments (Channel, Token) can be associated with any DELIMITER or DIPHTHONG token (i.e., those that have TokType!*=3) by putting the name of the function on the appropriate indicator for the ID for the token that is to be a Read macro.

A <u>ReadMacro</u> function is called by ChannelReadTokenWithHooks if the appropriate token with TOKTYPE!*=3 is returned by ChannelReadToken. This function can then take over the reading (or scanning) process, finally returning a token (actually an S-expression) to be returned in place of the token itself.

Example: The quote mark, 'x converting to (Quote x), is done by the following example which makes use of the function PutReadMacro which is defined in Section 10.12.

```
(de DOQUOTE (CHANNEL TOKEN))
   (LIST 'QUOTE  (CHANNELREAD CHANNEL))

(PUTREADMACRO LISPSCANTABLE!* '!' (FUNCTION DOQUOTE))
```

A ReadMacro is installed on the property list of the macro-character as a function under the indicators 'LISPREADMACRO, 'RLISPREADMACRO, etc. A Diphthong is installed on the property list of the first character as (second-character . diphthong) under the indicators 'LISPDIPHTHONG, 'RLISPDIPHTHONG, etc.

10.4.6. Terminal Interaction

(YesP MESSAGE:string): boolean expr

If the user responds Y or Yes, YesP returns T and the calculation continues from that point in the file. If the user responds N or No, YesP returns NIL and control is returned to the terminal, and the user can type in further commands. However, later on he can use the command CONT; and control is then transferred back to the point in the file after the last PAUSE was encountered. If the user responds B, one enters a break loop. After quitting the break loop, one still must respond Y, N, Yes, or No.

(Pause): Nil expr

(Only in RLISP.) The command PAUSE; may be inserted at any point in an input file. If this command is encountered on input, the system prints the message CONT? on the user's terminal and halts by calling YesP.

10.4.7. Input Status and Mode

PROMPTSTRING!* [Initially: "lisp>"] global

Displayed as a prompt when any input is taken from TTY. Thus prompts should 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:] global

This variable is set to LISPSCANTABLE!* by the Read function (the "Lisp syntax" reader). The RLISP reader sets it to RLISPSCANTABLE!* or LISPSCANTABLE!* depending on the syntax it expects.

CURRENTREADMACROINDICATOR!* [Initially:] 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, then call the function ChannelReadTokenWithHooks to perform input with the alternative set of Read macros.

Ordinary Read macros may be added by Puting 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.

```
(defun filter-my-file (x)
  (let ((chan (open "myfile.dat" 'input))
        form)
    (while (neq (setq form (channelread chan))
                $eof$)
           (if (and (pairp form) (eq (car form) x))
               (print form)))
    (close chan)))
```

The same thing with an unwind-protect form to give more assurance that the channel (and the file) will be closed in all cases including errors, is shown below.

```
(defun 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))))
```

The following functions are part of RLISP. Please do not use them in LISP code.

(Out U:string): None Returned macro

Opens file U for output, redirecting standard output. Note that Out takes a string as an argument, while Wrs takes an io-channel.

(EvOut L:string-list): None Returned expr

L is a list containing one file name which must be a string. EvOut is the called by Out after evaluating its argument.

(Shut [L:string]): None Returned macro

Closes the output files in the list L. Note that Shut takes file names as arguments, while Close takes an io-channel. The RLISP IN function maintains a stack of (file-name . io-channel) associations for this purpose. Thus a shut will also correctly select the previous file for further output.

(EvShut L:string-list): none Returned expr

Does the same as Shut but evaluates its arguments.

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 on the VAX or DEC-20); see Section 15.2.2 for information on producing FASL files.

(Load [FILE:{string, id}]): NIL macro

Each FILE is converted into a file name of the form "/u/local/lib/psl/file.b" on the VAX, "pl:file.b" on the DEC-20. An attempt is made to execute the function FaslIn on it. Once loaded, the symbol FILE is added to the GLOBAL variable OPTIONS!*. All loads consult the OPTIONS!* list and do not load files that are already present (See ReLoad below to load functions already loaded). Also, consult the GLOBALS LoadDirectories!* and LoadExtensions!* below for information on where loadable files may be found and how their names are constructed.

(ReLoad [FILE:{string,id}]): NIL macro

Removes the filename from the list Options!* and executes the function Load.

(Imports MODULENAMES:list): NIL expr

Imports is almost identical to load in its behavior, though not in the way it is called. The only behavioral difference is that if imports is invoked as a module is being loaded, the actual loading of the additional modules may be delayed until loading of the current module is complete. This allows the module loader to reclaim some space that would otherwise be wasted, a matter that is specific to the way PSL is currently implemented.

!*VerboseLoad [Initially: NIL] switch

If T, turns on !*RedefMsg during Loads so that every function redefined during a Load is announced. Also messages are given when a request is made to Load a file that is already loaded, and a message is printed for each file that is actually loaded.

!*PrintLoadNames [Initially: NIL] switch

If T, turns on printing of the message announcing each file loaded.

LoadDirectories!* [Initially: A list of strings] global

Contains a list of strings to append to the front of file names given in Load commands. This list may be one of the following, if your system is an Apollo, Dec-20, or Vax:

```
("" "/utah/psl/lap/")
("" "pl:")
("" "/usr/local/src/cmd/psl/dist/lap/")
```

load tries each directory on this list in turn as it searches for a specific file to load in as the requested module.

LoadExtensions!* [Initially: An a-list] global

Contains an a-list of (str . fn) in which the str is an extension to append to the end of the filename and fn is a function to apply. The a-list contains

```
((".b" . FaslIn)(".lap" . LapIn)(".sl" . LapIN))
```

At present the file extensions in this list are searched in order within each of the directories of LOADDIRECTORIES!*.

The following are some scenarios on the use of Load and Imports. Suppose that Module B:

a. requires modules C and D during its execution, but can be loaded after B. Then place (LOAD C D) in the file. When B is loaded interpretively, the load will be executed immediately. When B is compiled, the LOAD of C and D will be deferred until B is "LOADED". If (IMPORTS '(C D)) had been used, nothing would happen when B is interpreted, but when B is compiled, the same deferred load will occur.

b. needs module A to be loaded before it is loaded. It is then necessary to create a .LAP file to load in each of the modules:

```
AB.LAP consists of:
(LOAD A)
(LOAD B)
```

The user may then load in AB and will get A followed by B.

c. needs module A during its compilation. Place a (Compiletime (Load A)) somewhere in module B.

d. needs module A during its compilation and during execution. Place a (Bothtimes (Load A)) somewhere in module B.

10.7. Reading File into PSL

The following procedures are used to read complete files into PSL, by first calling Open, and then looping until end of file. The effect is similar to what would happen if the file were typed into PSL. Recall that file names are strings, and therefore one needs string-quotes (") around file names. File names may be given using full system dependent file name conventions, including directories and sub-directories, "links" and "logical-device-names", as appropriate on the specific system.

!*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 F:string): None Returned expr

Enters a Read-Eval-Print loop on the contents of the file F. DskIn expects LISP syntax in the file F. Use the following format: (Dskln "File").

(LapIn U:string): None Returned expr

Reads a single LISP file as "quietly" as possible, i.e., it does not echo or return values. Note that LapIn can be used only for LISP files. By convention, files with the extension ".LAP" are intended to be read by LapIn. These files are typically used to load modules made up of several binary (also known as FASL) files. The use of the Load function is normally preferable to using LapIn. For information about fast loading of files of compiled functions (FASL files) see FASL and the Load and FaslIn functions in Chapter 15.

(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.

10.7.1. RLISP File Reading Functions

The following functions are present in RLISP, they can be used from Bare-PSL by loading RLISP.

(In [L:string]): None Returned macro

Similar to DskIn but expects RLISP syntax in the files it reads unless it can determine that the files are not in RLISP syntax. Also In can take more than one file name as an argument. On most systems the function In expects files with extension .LSP and .SL to be written in LISP syntax, not in RLISP. This is convenient when using both LISP and RLISP files. It is conventional to use the extension .RED (or .R) for RLISP files and use .LSP or .SL only for fully parenthesized LISP files. There are some system programs, such as TAGS on the DEC-20, which expect RLISP files to have the extension .RED.

If it is not desired to have the contents of the file echoed as it is read, either end the In command with a "$" in RLISP, as

```
In "FILE1.RED","FILE2.SL"$
```

or include the statement "Off ECHO;" in your file.

(PathIn FileName-Tail:string): None Returned expr

Allows the use of a directory search path with the Rlisp IN function. It finds a list of search paths in the fluid variable PATHIN!*. These are successively concatenated onto the front of the string argument to PathIn until an existing file is found (using FileP. If one is found, In will be invoked on this file. If not, a continuable error occurs. For example on the VAX,

```
(Setq PathIn!* '( "" "/u/psl/" "/u/smith/"))
(PathIn "foo.red")
```

will attempt to open "foo.red", then "/u/psl/foo.red", and finally "/u/smith/foo.red" until a successful open is achieved.

To use Pathin in Bare-PSL, load PATHIN as well as RLISP.

!*PrintPathin [Initially: NIL] switch

If T, a message is printed for each file that is read by Pathin.

(EvIn L:string-list): None Returned expr

L must be a list of strings that are filenames. EvIn is the function called by In after evaluating its arguments. In is useful only at the top-level, while EvIn can be used inside functions with file names passed as parameters.

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.

HelpIn!* [Initially: NIL] global

The Help mechanism uses this variable's value for input.

HelpOut!* [Initially: NIL] global

This variable's value determines the output channel used by the Help mechanism.

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. If overflow occurs BldMsg returns a string stating that the string could not be constructed.

(FlatSize U:any): integer expr

Character length of Prin1 S-expression.

(FlatSize2 U:any): integer expr

Prin2 version of flatsize.

Note that for many purposes it is easier to use DigitP, AlphaP, etc. for performing the kind of testing that Digit and Liter do.

(Digit U:any): boolean expr

Returns T if U is a digit, otherwise NIL. Effectively this is:

```
(de DIGIT (U)
  (IF (MEMQ U '(!0 !1 !2 !3 !4 !5 !6 !7 !8 !9)) T NIL))
```

(Liter U:any): boolean expr

Returns T if U is a character of the alphabet, NIL otherwise. This is effectively:

```
(de LITER(U)
  (IF (MEMQ U '(A B C D E F G H I J K L M
    N O P Q R S T U V W X Y Z a b c d e f
    g h i j k l m n o p q r s t u v w x y
    z))  T NIL))
```

(**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 **ChannelPrin1**, 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. Example:

* Explode 'FOO; => (F O O)

* Explode '(A . B); => (!(A ! !. ! B !))

[??? add print macros. cf. UCI lisp ???]

(**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

[??? We should replace these globals and SPECIAL option by a (SPECIALOPEN Readfunction writefunction closefunction) call ???]

All input and output functions are implemented in terms of operations on "channels". A channel is just a small integer[2] which has 3 functions and some other information associated with it. The three functions are:

a. 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 T. 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.

b. 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.

c. 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),

[2]The range of channel numbers is from 0 to MaxChannels, where MaxChannels is a system-dependent constant, currently 31, defined in IO-DATA.RED. MaxChannels is a WCONST, and is not available for use at runtime.

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

Note: Please pardon the creaky mechanism used to implement this facility. We expect to improve it.

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

Scan tables have 129 entries, indexed by 0 through 128. 0 through 127 are indexed by ASCII character code to get an integer code determining the treatment of the corresponding character. The last entry is not an integer, but rather an id which specifies a Diphthong Indicator for the token scanner.

[??? A future implementation may replace the FLUID CurrentReadMacroIndicator!* with another entry in the scan table. ???]

The following encoding for characters is used.

0 ... 9	DIGIT: indicates the character is a digit, and gives the corresponding numeric value.
10	LETTER: indicates that the character is a letter.
11	DELIMITER: indicates that the character is a delimiter which is not the starting character of a diphthong.
12	COMMENT: indicates that the character begins a comment terminated by an end of line.
13	DIPHTHONG: indicates that 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, i.e., "<<" or ":=".)
14	IDESCAPE: indicates that the character is an escape character, to cause the following character to be taken as part of an id. (Ordinarily an exclamation point, i.e. "!".)
15	STRINGQUOTE: indicates that the character is a string quote. (Ordinarily a double quote, i.e. '"'.)
16	PACKAGE: indicates that the character is used to introduce explicit package names. (Ordinarily "\".)
17	IGNORE: indicates that the character is to be ignored. (Ordinarily BLANK, TAB, EOL and NULL.)
18	MINUS: indicates that the character is a minus sign.
19	PLUS: indicates that the character is a plus sign.
20	DECIMAL: indicates that the character is a decimal point.
21	IDSURROUND: indicates that the character is to act for identifiers as a string quote acts for strings. Although this is not used in the default scan table, the intended character for this function is a vertical bar, |.)

System builders who wish to define their own parsers can bind an appropriate scan table to **CurrentScanTable!*** and then call **ChannelReadToken** or **ChannelReadTokenWithHooks** for lexical scanning. Utility functions for building scan tables are described in the next section.

LISPSCANTABLE!* [Initially: as shown in following table] global

0	^@	IGNORE	32		IGNORE	64	@	LETTER	96	'	DELIMITER
1	^A	LETTER	33	!	IDESCAPECHAR	65	A	LETTER	97	a	LETTER
2	^B	LETTER	34	"	STRINGQUOTE	66	B	LETTER	98	b	LETTER
3	^C	LETTER	35	#	LETTER	67	C	LETTER	99	c	LETTER
4	^D	LETTER	36	$	LETTER	68	D	LETTER	100	d	LETTER
5	^E	LETTER	37	%	COMMENTCHAR	69	E	LETTER	101	e	LETTER
6	^F	LETTER	38	&	LETTER	70	F	LETTER	102	f	LETTER
7	^G	LETTER	39	'	DELIMITER	71	G	LETTER	103	g	LETTER
8	^H	LETTER	40	(	DELIMITER	72	H	LETTER	104	h	LETTER
9	<tab>	IGNORE	41	)	DELIMITER	73	I	LETTER	105	i	LETTER
10	<lf>	IGNORE	42	*	LETTER	74	J	LETTER	106	j	LETTER
11	^K	LETTER	43	+	PLUSSIGN	75	K	LETTER	107	k	LETTER
12	^L	IGNORE	44	,	DIPHTHONGSTART	76	L	LETTER	108	l	LETTER
13	<cr>	IGNORE	45	-	MINUSSIGN	77	M	LETTER	109	m	LETTER
14	^N	LETTER	46	.	DECIMALPOINT	78	N	LETTER	110	n	LETTER
15	^O	LETTER	47	/	LETTER	79	O	LETTER	111	o	LETTER
16	^P	LETTER	48	0	DIGIT	80	P	LETTER	112	p	LETTER
17	^Q	LETTER	49	1	DIGIT	81	Q	LETTER	113	q	LETTER
18	^R	LETTER	50	2	DIGIT	82	R	LETTER	114	r	LETTER
19	^S	LETTER	51	3	DIGIT	83	S	LETTER	115	s	LETTER
20	^T	LETTER	52	4	DIGIT	84	T	LETTER	116	t	LETTER
21	^U	LETTER	53	5	DIGIT	85	U	LETTER	117	u	LETTER
22	^V	LETTER	54	6	DIGIT	86	V	LETTER	118	v	LETTER
23	^W	LETTER	55	7	DIGIT	87	W	LETTER	119	w	LETTER
24	^X	LETTER	56	8	DIGIT	88	X	LETTER	120	x	LETTER
25	^Y	LETTER	57	9	DIGIT	89	Y	LETTER	121	y	LETTER
26	^Z	DELIMITER	58	:	LETTER	90	Z	LETTER	122	z	LETTER
27	$	LETTER	59	;	LETTER	91	[	DELIMITER	123	{	LETTER
28	^\	LETTER	60	<	LETTER	92	\	PACKAGE	124	|	LETTER
29	^]	LETTER	61	=	LETTER	93	]	DELIMITER	125	}	LETTER
30	^^	LETTER	62	>	LETTER	94	^	LETTER	126	~	LETTER
31	^_	LETTER	63	?	LETTER	95	_	LETTER	127	<rubout>	LETTER

The Diphthong Indicator in the 128th entry is the identifier LISPDIPHTHONG.

RLISPSCANTABLE!* [Initially: as shown in following table] global

0 ^@ IGNORE	32 IGNORE	64 @ DELIMITER	96 ' DELIMITER
1 ^A DELIMITER	33 ! IDESCAPECHAR	65 A LETTER	97 a LETTER
2 ^B DELIMITER	34 " STRINGQUOTE	66 B LETTER	98 b LETTER
3 ^C DELIMITER	35 # DELIMITER	67 C LETTER	99 c LETTER
4 ^D DELIMITER	36 $ DELIMITER	68 D LETTER	100 d LETTER
5 ^E DELIMITER	37 % COMMENTCHAR	69 E LETTER	101 e LETTER
6 ^F DELIMITER	38 & DELIMITER	70 F LETTER	102 f LETTER
7 ^G DELIMITER	39 ' DELIMITER	71 G LETTER	103 g LETTER
8 ^H DELIMITER	40 (DELIMITER	72 H LETTER	104 h LETTER
9 <tab> IGNORE	41) DELIMITER	73 I LETTER	105 i LETTER
10 <lf> IGNORE	42 * DIPHTHONGSTART	74 J LETTER	106 j LETTER
11 ^K DELIMITER	43 + DELIMITER	75 K LETTER	107 k LETTER
12 ^L IGNORE	44 , DELIMITER	76 L LETTER	108 l LETTER
13 <cr> IGNORE	45 - DELIMITER	77 M LETTER	109 m LETTER
14 ^N DELIMITER	46 . DECIMALPOINT	78 N LETTER	110 n LETTER
15 ^O DELIMITER	47 / DELIMITER	79 O LETTER	111 o LETTER
16 ^P DELIMITER	48 0 DIGIT	80 P LETTER	112 p LETTER
17 ^Q DELIMITER	49 1 DIGIT	81 Q LETTER	113 q LETTER
18 ^R DELIMITER	50 2 DIGIT	82 R LETTER	114 r LETTER
19 ^S DELIMITER	51 3 DIGIT	83 S LETTER	115 s LETTER
20 ^T DELIMITER	52 4 DIGIT	84 T LETTER	116 t LETTER
21 ^U DELIMITER	53 5 DIGIT	85 U LETTER	117 u LETTER
22 ^V DELIMITER	54 6 DIGIT	86 V LETTER	118 v LETTER
23 ^W DELIMITER	55 7 DIGIT	87 W LETTER	119 w LETTER
24 ^X DELIMITER	56 8 DIGIT	88 X LETTER	120 x LETTER
25 ^Y DELIMITER	57 9 DIGIT	89 Y LETTER	121 y LETTER
26 ^Z DELIMITER	58 : DIPHTHONGSTART	90 Z LETTER	122 z LETTER
27 $ DELIMITER	59 ; DELIMITER	91 [DELIMITER	123 { DELIMITER
28 ^\ DELIMITER	60 < DIPHTHONGSTART	92 \ PACKAGE	124 | DELIMITER
29 ^] DELIMITER	61 = DELIMITER	93] DELIMITER	125 } DELIMITER
30 ^^ DELIMITER	62 > DIPHTHONGSTART	94 ^ DELIMITER	126 ~ DELIMITER
31 ^_ DELIMITER	63 ? DELIMITER	95 _ LETTER	127 <rubout> DELIMITER

The Diphthong Indicator in the 128th entry is the identifier RLISPDIPHTHONG.

[??? What about the RlispRead scantable ???]

[??? Perhaps describe one basic table, and changes from one to other, since mostly the same ???]

TOKTYPE!* [Initially: 3] global

ChannelReadToken sets TOKTYPE!* to:

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.

In the last case, the value returned is the id whose print name is the same as the delimiter.

10.12. Scan Table Utility Functions

The following functions are provided to manage scan tables, in the READ-UTILS module (use via LOAD READ-UTILS):

(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.

[??? Make smarter, reduce output, use nice names for control characters, ala EMODE. ???]

(CopyScanTable OLDTABLE:{vector, NIL}): vector expr

Copies the existing scantable (or CURRENTSCANTABLE!* if given NIL). Currently GenSym()'s the indicators used for diphthongs.

[??? Change when we use Property Lists in extra slots of the Scan-Table ???]

(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.

(PutReadMacro TABLE:vector ID1:id FNAME:id): NIL expr

Installs FNAME as the name of the Read macro function for the delimiter or diphthong ID1 in the given scan table. **[not implemented yet]**

CHAPTER 11
TOP LEVEL LOOP

11.1. Introduction

In this chapter those functions are presented relating directly to the user interface; for example, the general purpose Top Loop function, the History mechanism, and changing the default Top Level function.

11.2. The General Purpose Top Loop Function

PSL provides a general purpose Top Loop that allows the user to specify his own Read, Eval and Print functions and otherwise obtain a standard set of services, such as Timing, History, Break Loop interface, and interface to the Help system.

TopLoopEval!* [Initially: NIL] global

The Eval used in the current Top Loop.

TopLoopPrint!* [Initially: NIL] global

The Print used in the current Top Loop.

TopLoopRead!* [Initially: NIL] global

The Read used in the current Top Loop.

(TopLoop TOPLOOPREAD!*:function TOPLOOPPRINT!*:function
TOPLOOPEVAL!*:function TOPLOOPNAME!*:id WELCOMEBANNER:string): NIL expr

This function is called to establish a new Top Loop (currently used for Standard LISP, RLISP, and Break). It prints the WELCOMEBANNER and then invokes a "Read-Eval-Print" loop, using the given functions. Note that TOPLOOPREAD!*, etc. are FLUID variables, and so may be examined (and changed) within the executing Top Loop. TopLoop provides a standard

History and timing mechanism, retaining on a list HISTORYLIST!* the input and output as a list of pairs. A prompt is constructed from TOPLOOPNAME!* and is printed out, prefixed by the History count. As a convention, the name is followed by a number of ">"'s, indicating the loop depth.

The initial values ot the following four globals are those that exist in Bare-PSL. They may differ in other PSL executables.

TopLoopName!* [Initially: lisp] global

Short name to put in prompt.

TopLoopLevel!* [Initially: 0] global

Depth of top loop invocations.

!*EMsgP [Initially: T] switch

Whether to print error messages.

InitForms!* [Initially: NIL] global

Forms to be evaluated at startup.

!*TIME [Initially: NIL] switch

If on, causes a step evaluation time to be printed after each command.

(Hist [N:integer]): NIL nexpr

This function does not work with the Top Loop used by PSL:RLISP or by (beginrlisp); it does work with LISP and with RLISP if it is started from LISP using the RLISP function. 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.

[??? Add more info about what a history is. ???]

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

Number of entries read so far.

HistoryList!* [Initially: Nil]   global

List of entries read and evaluated.

TopLoop has been used to define the following StandardLisp and RLISP top loops.

(StandardLisp): NIL   expr

Interpreter LISP syntax top loop, defined as:

```
(De StandardLisp Nil
   (Prog (CurrentReadMacroIndicator!* CurrentScanTable!*)
       (Setq CurrentReadMacroIndicator!* 'LispReadMacro)
       (Setq CurrentScanTable!* LispScanTable!*)
       (Toploop 'Read 'Print 'Eval "LISP"
                                 "PORTABLE STANDARD LISP")))
```

Note that the scan tables are modified.

(RLisp): NIL   expr

Alternative interpreter RLISP syntax top loop, defined as:

[??? xread described in RLISP Section ???]

```
(De RLisp Nil
(Toploop 'XRead 'Print 'Eval "RLISP" "PSL RLISP"))
```

Note that for the moment, the default RLISP loop is not this (though this may be used experimentally); instead a similar (special purpose hand coded) function, BeginRlisp, based on the older Begin1 is used. It is hoped to change the RLISP top-level to use the general purpose capability.

(BeginRLisp): None Returned expr

Starts RLISP from PSL:PSL only if RLISP is loaded. The module RLISP is present if you started in RLISP and then entered PSL.

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 While loop, protected by a Catch. By default, Main calls a StandardLisp top loop, defined using the general TopLoop function, described in the next section. In order to have a saved PSL come up in a different top loop, the function Main should be appropriately redefined by the user (e.g., as is done to create RLISP).

(Main): Undefined expr

Initialization function, called after setting the stack. Should be redefined by the user to change the default TopLoop.

11.4. The Break Loop

The Break Loop uses the top loop mechanism and is described in detail in Chapter 12. For information, look there.

CHAPTER 12
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.

[??? What about errors signalled to the Interrupt Handler ???]

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 LISP.

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.

[??? Describe Error # ranges here, or have in a file on machine ???]

(**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 (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. The 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. Its definition is:

```
(ProgN (ErrorPrintF "***** Fatal error: %s" S)
       (While T (Quit)))
```

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] global

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. Since ErrorSet is somewhat obsolete, the precise behavior controlled by this flag may change.

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 "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. [Not enough errors set this yet]. 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 E, 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. If it does, an escape command will be provided for examining the relevant variables.

The meanings of these commands are:

E	Edit the value of ErrorForm!*. This is the object printed in the "Retry form is ..." message. The function BreakEdit is the associated function called by this command. The Retry command (below) uses the corrected version of ErrorForm!*. The currently available editors are described in Part 2 of the manual.
M	Show the modified ErrorForm!*. 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, assigning a value to a variable, or using the Edit command, above. 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: '(Eval Apply FastApply CodeApply CodeEvalApply Catch ErrorSet EvProgN TopLoop BreakEval BindEval Break Main)] global

A list of function names that will not be printed by the commands I and T given within a Break loop.

InterpreterFunctions!* [Initially: '(Cond Prog And Or ProgN SetQ)] global

A list of function names that will not be printed by the command I given within a Break loop.

The above two globals can be reset in an init file if the programmer desires to do so.

The following is a slightly edited transcript, showing some of the BREAK options:

```
% foo is an undefined function, so the following has two errors
%   in it

1> (Plus2 (foo 1)(foo 2))
***** 'FOO' is an undefined function {1001}
***** Continuable error: retry form is '(FOO 1)'
Break loop
1 lisp break> (plus2 1 1)      % We simply compute a value
2                               % prints as 2
2 lisp break> c                 % continue with this value

% it returns to compute "(foo 2)"

***** 'FOO' is an undefined function {1001}
***** Continuable error: retry form is '(FOO 2)'
Break loop
1 lisp break> 3                 % again compute a value
3
2 lisp break> c                 % and return
5                               % finally complete

% Pretend that we had really meant to call "fee":

2> (de fee (x) (add1 x))
FEE
3> (plus2 (foo 1)(foo 2))              % now the bad expression
***** 'FOO' is an undefined function {1001}
***** Continuable error: retry form is '(FOO 1)'
Break loop
1 lisp break> e                % lets edit it

Type HELP<CR> for a list of commands.

  edit> p                       % print form
(FOO 1)
  edit> (1 fee)                 % replace 1'st by "fee"
  edit> p                       % print again
(FEE 1)
  edit> ok                      % we like it
(FEE 1)
2 lisp break> m                % show modified ErrorForm!*
ErrorForm!* : '(FEE 1)'
NIL
3 lisp break> r                % Retry EVAL ErrorForm!*
***** 'FOO' is an undefined function {1001}
***** Continuable error: retry form is '(FOO 2)'
```

```
Break loop
1 lisp break> (de foo(x) (plus2 x 1))  % define foo
FOO
2 lisp break> r                        % and retry
5
```

12.5. Interrupt Keys

Need to load the module INTERRUPT to enable. This applies only to the DEC20.

<Ctrl-T> indicates routine currently executing, gives the load average, and gives the location counter in octal;

<Ctrl-G> returns you to the Top-Loop;

<Ctrl-B> takes you into a lower-level Break loop.

12.6. 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.

12.7. Some Convenient Error Calls

The following functions may be useful in user packages:

(RangeError Object:any Index:integer Fn:function): None Returned *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))
```

(IndexError Offender:any Fn:function): None Returned *expr*

```
(UsageTypeError Offender Fn "an integer" "an index")
```

(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 Returned   *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")

CHAPTER 13
DEBUGGING TOOLS

13.1. Introduction

This chapter describes the debugging facilities available in PSL. Most of these are made available by loading the module DEBUG. There is also a stepper made available by loading STEP. It is described in Section 13.2. An extensive example showing the use of the facilities in the debugging package can be found in Section 13.13.

13.1.1. Brief Summary of Full Debug Package

The PSL debugging package contains a selection of functions that can be used to aid program development and to investigate faulty programs.[1]

It contains the following facilities.

[1]Much of this chapter was adapted from a paper by Norman and Morrison.

* A trace package. This allows the user to see the arguments passed to and the values returned by selected functions. It is also possible to have traced interpreted functions print all the assignments they make with SetQ (see Section 13.3).

* A break package. This allows the user to wrap a Break around functions.

* A backtrace facility. This allows one to see which of a set of selected functions were active as an error occurred (see Section 13.6).

* Embedded functions make it possible to do everything that the trace package can do, and much more besides (see Section 13.7). This facility is available only in RLISP.

* Some primitive statistics gathering (see Section 13.8).

* Generation of simple stubs. If invoked, procedures defined as stubs simply print their argument and read a value to return (see Section 13.9).

* Some functions for printing useful information, such as property lists, in an intelligible format (see Section 13.10).

* PrintX is a function that can print circular and re-entrant lists and vectors, and so can sometimes allow debugging to proceed even in the face of severe damage caused by the wild use of RplacA and RplacD (see Section 13.11).

13.1.2. Redefining of User Functions by Debug

Many facilities in Debug depend upon redefining user functions so that they may log or print behavior when called. Since several facilities may be active simultaneously for a single user function, Debug redefines a function only the first time a facility is requested. Information about which facility was requested is kept on a property list. If a second facility is requested for a function, that information is added to the property list. When the function is called, the property list is examined to see what activities should occur.

Turning off a specific Debug facility does not cause the function to have its original definition restored. All that happens is that information about the facility is removed from the property list. To restore the original definition of the function use the Restr macro described in Section 13.3.4.

13.1.3. A Few Known Deficiencies

* An attempt to trace certain system functions (e.g. Cons) causes the trace package to overwrite itself. Given the names of functions that cause this sort of trouble it is fairly easy to change the trace package to deal gracefully with them. The global BreakDebugList!* contains the names of functions known to cause trouble. Report any other functions causing trouble to a system expert or send mail to PSL-BUGS.

* The Portable LISP Compiler uses information about registers which certain system functions destroy. Tracing these functions may make the optimizations based thereon invalid. The correct way of handling this problem is currently under consideration. In the mean time you should avoid tracing any functions with the ONEREG or TWOREG flags.

13.2. Step

(Step F:form): any expr

Step is a loadable option (LOAD STEP). It evaluates the form _F_, single-stepping. _F_ is printed, preceded by -> on entry, <-> for macro expansions. After evaluation, _F_ is printed preceded by <- and followed by the result of evaluation. A single character is read at each step to determine the action to be taken:

<Ctrl-N> (Next)
Step to the Next thing. The stepper continues until the next thing to print out, and it accepts another command.

Space
Go to the next thing at this level. In other words, continue to evaluate at this level, but don't step anything at lower levels. This is a good way to skip over parts of the evaluation that don't interest you.

<Ctrl-U> (Up)
Continue evaluating until we go up one level. This is like the space command, only more so; it skips over anything on the current level as well as lower levels.

<Ctrl-X> (eXit)
Exit; finish evaluating without any more stepping.

<Ctrl-G> or <Ctrl-P> (Grind)
Grind (i.e., prettyprint) the current form.

<Ctrl-R> Grind the form in Rlisp syntax.

<Ctrl-E> (Editor)
Invoke the structure editor on the current form.

<Ctrl-B> (Break)
Enter a break loop from which you can examine the values of variables and other aspects of the current environment.

<Ctrl-L> Redisplay the last 10 pending forms.

? Display the help file.

To step through the evaluation of function H on argument X do

```
(Step '(H X))
```

13.3. Tracing Function Execution

13.3.1. Tracing Functions

To see when a function gets called, what arguments it is given and what value it returns, do

```
(TR functionname)
```

or if several functions are of interest,

```
(TR name1 name2 ...)
```

(Tr [FNAME:id]): Undefined macro

If the specified functions are defined (as expr, fexpr, nexpr or macro), Tr modifies the function definition to include print statements. Note that the arguments are not quoted. The following example shows the style of output produced by this sort of tracing:

The input...

```
(DE XCDR (A)
  (CDR A) %A very simple function)
(TR XCDR)
(XCDR '(P Q R))
```

gives output...

```
XCDR entered
   A: (P Q R)
XCDR = (Q R)
```

Interpreted functions can also be traced at a deeper level.

(Trst [FNAME:id]): Undefined macro

```
(TRST name1 name2 ...)
```

causes the body of an interpreted function to be redefined so that all assignments (made with SetQ) in its body are printed. Calling Trst on a function automatically has the effect of doing a Tr on it too, so that it is not possible to have a function subject to Trst but not Tr.

One can use the Trst facility to cause only assignments to variables specified in a function to be printed instead of all of them.

(TrstSome FNAME:id [VARS:id]): Undefined macro

```
(TrstSome fname var1 var2 ...)
```

Give the function name first and then the variables.

Trace output often appears mixed up with output from the program being studied, and to avoid too much confusion Tr arranges to preserve the column in which printing was taking place across any output that it generates. If trace output is produced as part of a line has been printed, the trace data are enclosed in markers '<' and '>', and these symbols are placed on the line so as to mark out the amount of printing that had occurred before trace was entered.

!*PrintNoArgs [Initially: NIL] switch

If !*PrintNoArgs is T, printing of the arguments of traced or broken functions is suppressed.

TracedFns!* [Initially: NIL] global

TracedFns!* contains the names of all functions currently being traced.

13.3.2. Saving Trace Output

The trace facility makes it possible to discover in some detail how a function is used, but in certain cases its direct use results in the generation of vast amounts of (mostly useless) print-out. There are several options. One is to make tracing more selective (see Section 13.3.3). The other, discussed here, is to either print only the most recent information, or dump it all to a file to be perused at leisure.

Debug has a ring buffer in which it saves information to reproduce the most recent information printed by the trace facility (both Tr and Trst). To see the contents of this buffer use Tr without any arguments

```
(TR)
```

(NewTrBuff N:integer): Undefined expr

To set the number of entries retained to n use

```
(NEWTRBUFF n)
```

Initially the number of entries in the ring buffer is 5.

!*TRACE [Initially: T] switch

Enables runtime printing of trace information for functions which have been traced.

Turning off the TRACE switch

```
(OFF TRACE)
```

suppresses the printing of any trace information at run time; it is still saved in the ring buffer. Thus a useful technique for isolating the function in which an error occurs is to trace a large number of candidate functions, do OFF TRACE and after the failure look at the latest trace information by calling Tr with no arguments.

(TrOut [FNAME:id]): Undefined expr

(StdTrace): Undefined expr

Normally trace information is directed to the standard output, rather than the currently selected output. To send it elsewhere use the statement

```
(TROUT filename)
```

The statement

```
(STDTRACE)
```

closes that file and cause future trace output to be sent to the standard output. Note that output saved in the ring buffer is sent to the currently selected output, not that selected by TrOut.

13.3.3. Making Tracing More Selective

One can specify a predicate when tracing a function so that tracing will be enabled only when the predicate is true.

(TrWhen FNAME:id PREDICATE:form): Undefined macro

Trace information for the function FNAME will be printed only if PREDICATE is T. The variables in the predicate must be either globals or parameters of the function.

```
(TrWhen foo (GreaterP x y))
```

(TraceCount N:integer): Undefined expr

The function (TraceCount n) can be used to switch off trace output. If n is a positive number, after a call to (TraceCount n) the next n items of trace output that are generated are not printed. (TraceCount n) with n negative or zero switches all trace output back on. (TraceCount NIL) returns the residual count, i.e., the number of additional trace entries that are suppressed.

To get detailed tracing in the stages of a calculation that lead up to an error, try

```
(TRACECOUNT 1000000) % or some other suitable large number
(TR ...)  % as required
%run the failing problem
(TRACECOUNT NIL)
```

It is now possible to calculate how many trace entries occurred before the error, and so the problem can now be re-run with TraceCount set to some number slightly less than that.

An alternative to the use of TraceCount for getting more selective trace output is TrIn.

(TrIn [FNAME:id]): Undefined macro

To use TrIn, establish tracing for a collection of functions, using Tr in the normal way. Then do TrIn on some small collection of other functions. The effect is just as for Tr, except that trace output is inhibited except if control is dynamically within the TrIn functions. This makes it possible to use Tr on a number of heavily used general purpose functions, and then only see the calls to them that occur within some specific subpart of your entire program.

TRACEMINLEVEL!* [Initially: 0] global

TRACEMAXLEVEL!* [Initially: 1000] global

The global variables TRACEMINLEVEL!* and TRACEMAXLEVEL!* (whose values should be non-negative integers) are the minimum and maximum depths of recursion at which to print trace information. Thus if you only want to see top level calls of a highly recursive function (like a simple-minded version of **Length**) simply do

```
(SETQ TRACEMAXLEVEL!* 1)
```

13.3.4. Turning Off Tracing

If a particular function no longer needs tracing, do

```
(UNTR functionname)
```

or

```
(UNTR name1 name2 ...)
```

(UnTr [FNAME:id]): Undefined macro

This merely suppresses generation of trace output. Other information, such as invocation counts, backtrace information, and the number of arguments is retained.

To completely destroy information about a function use

```
(RESTR name1 name2 ...)
```

(Restr [FNAME:id]): Undefined macro

This returns the functions specified to their original state. If no arguments are given, all functions will be returned to their original state.

To suppress traceset output without suppressing normal trace output use

```
(UNTRST name1 name2 ...)
```

(UnTrst [FNAME:id]): Undefined macro

UnTring a Trsted function also UnTrst's it.

TrIn in Section 13.3.3 is undone by UnTr (but not by UnTrst).

(UnTrAll): Undefined expr

The function UnTrAll untraces all functions currently traced, i.e., all functions on the list TracedFns!*.

13.4. A Break Facility

A break facility exists in Debug that allows one to wrap a Break around a function, causing a Break to occur both before and after execution of the function. Variants on the break function similar to those available for the trace function are available.

(Br [FNAME:id]): Undefined <u>macro</u>

Br causes a Break to be placed around each of the functions listed. A Break occurs both before and after the execution of each broken function. Give a c command to the Break to continue before execution and an r command to continue after execution.

(BrIn [FNAME:id]): Undefined <u>macro</u>

BrIn is used in the same way as TrIn, to cause breaking of a broken function only within the functions specified.

Note that if a function specified by BrIn terminates abnormally, the BrIn facility may not work properly. To fix it, call BrIn with no arguments.

(BrWhen FNAME:id PREDICATE:form): Undefined <u>macro</u>

One can specify a predicate when breaking around a function so that Breaks will be enabled only when the predicate is true. This works exactly as the macro TrWhen.

BrokenFns!* [<u>Initially:</u> NIL] <u>global</u>

BrokenFns!* contains the names of all functions currently broken.

Note that the switch !*PrintNoArgs is also used by the break facility.

(UnBr [FNAME:id]): Undefined <u>macro</u>

UnBr causes breaking to be disabled for the functions specified.

(UnBrAll): Undefined <u>expr</u>

Unbreaks all functions that are currently broken, i.e., all functions on the list BrokenFns!*.

13.5. Enabling Debug Facilities and Automatic Tracing and Breaking

Under the influence of

```
(ON TRACEALL)
```

any functions successfully defined by PutD are traced. Note that if PutD fails (as might happen under the influence of the LOSE flag) no attempt is made to trace the function.

```
(ON BREAKALL)
```

causes any functions successfully defined by PutD to be broken.

To enable those facilities (such as Btr in Section 13.6 and TrCount in Section 13.8) which require redefinition, but without tracing, use

```
(ON INSTALL)
```

Thus, a common scenario might look like

```
(ON INSTALL)
(DSKIN "MYFNS.SL")
(OFF INSTALL)
```

which would enable the backtrace and statistics routines to work with all the functions defined in the MYFNS file.

!*INSTALL [Initially: NIL] switch

Causes DEBUG to know about all functions defined with PutD.

!*TRACEALL [Initially: NIL] switch

Causes all functions defined with PutD to be traced.

!*BreakAll [Initially: NIL] switch

Causes all functions defined with PutD to be broken.

13.6. A Heavy Handed Backtrace Facility

The backtrace facility allows one to see which of a set of selected functions were active as an error occurred. The function **Btr** gives the backtrace information. The information kept is controlled by two switches: !*BTR and !*BTRSAVE.

When backtracing is enabled (BTR is on), a stack is kept of functions entered but not left. This stack records the names of functions and the arguments that they were called with. If a function returns normally the stack is unwound. If however the function fails, the stack is left alone by the normal LISP error recovery processes.

(Btr [FNAME:id]): Undefined macro

When called with no arguments, **Btr** prints the backtrace information available. When called with arguments (which should be function names), the stack is reset to Nil, and the functions named are added to the list of functions Debug knows about.

(ResBtr [FNAME:id]): Undefined expr

ResBtr resets the backtrace stack to Nil.

!*BTR [Initially: T] switch

If !*BTR is T, it enables backtracing of functions which the Debug package has been told about. If it is Nil, backtrace information is not saved.

!*BTRSAVE [Initially: T] switch

Controls the disposition of information about functions which failed within an **ErrorSet**. If it is on, the information is saved separately and printed when the stack is printed. If it is off, the information is thrown away.

13.7. Embedded Functions

Embedding means redefining a function in terms of its old definition, usually with the intent that the new version does some tests or printing, uses the old one, does some more printing and then returns. If ff is a function of two arguments, it can be embedded using a statement of the form:

```
   SYMBOLIC EMB PROCEDURE ff(A1,A2);
     << PRINT A1;
        PRINT A2;
        PRINT ff(A1,A2) >>;
```

The effect of this particular use of embed is broadly similar to a call **Tr** ff, and arranges that whenever ff is called it prints both its arguments and its result. After a function has been embedded, the embedding can be temporarily removed by the use of

```
   UNEMBED ff;
```

and it can be reinstated by

```
   EMBED ff;
```

This facility is available only to RLISP users.

13.8. Counting Function Invocations

!*TRCOUNT [Initially: T] switch

Enables counting invocations of functions known to Debug. If the switch TRCOUNT is ON, the number of times user functions known to Debug are entered is counted. The statement

```
   (ON TRCOUNT)
```

also resets that count to zero. The statement

```
   (OFF TRCOUNT)
```

causes a simple histogram of function invocations to be printed.

If regular tracing (provided by **Tr**) is not desired, but you wish to count the function invocations, use

```
   (TRCNT name1 name2 ...)
```

(TrCnt [FNAME:id]): Undefined macro

See also Section 13.5.

13.9. Stubs

Stubs are useful in top-down program development. If a stub is invoked, it prints its arguments and asks for a value to return.

(Stub [FuncInvoke:form]): macro

> Each FUNCINVOKE must be of the form (id arg1 arg2 ...), where there may be zero arguments. Stub defines an expr for each form with name id and formal arguments arg1, arg2, etc. If executed such a stub prints its arguments and reads a value to return.

The statement

```
(STUB (FOO U V))
```

defines an expr, Foo, of two arguments.

(FStub [FuncInvoke:form]): Nil macro

> FStub does the same as Stub but defines fexprs.

At present the currently (i.e., when the stub is executed) selected input and output are used. This may be changed in the future. Algebraic and possibly macro stubs may be implemented in the future.

13.10. Functions for Printing Useful Information

(PList [X:id]): macro

```
(PLIST id1 id2 ...)
```

> prints the property lists of the specified ids in an easily readable form.

(Ppf [FNAME:id]): macro

```
    (PPF fn1 fn2 ...)
```

prints the definitions and other useful information about the specified functions.

13.11. Printing Circular and Shared Structures

Some LISP programs rely on parts of their data structures being shared, so that an Eq test can be used rather than the more expensive Equal one. Other programs (either deliberately or by accident) construct circular lists through the use of RplacA or RplacD. Such lists can be displayed by use of the function PrintX. This function also prints circular vectors.

(PrintX A:any): NIL expr

If given a normal list the behavior of this function is similar to that of Print; if it is given a looped or re-entrant data structures it prints it in a special format. The representation used by PrintX for re-entrant structures is based on the idea of labels for those nodes in the structure that are referred to more than once.

Consider the list created by the operations:

```
(SETQ R '(S W))
(RPLACA R (CDR R))
```

The function Print called on the list R gives

```
((W) W)
```

If PrintX is called on the list R, it discovers that the list (W) is referred to twice, and invents the label %L1 for it. The structure is then printed as

```
(%L1: (W) . %L1)
```

%L1: sets the label, and the other instance of %L1 refers back to it. Labeled sublists can appear anywhere within the list being printed. Thus the list created by the following statements

```
(SETQ L '(A B C))
(SETQ K (CDR L))
(SETQ X (CONS L K))
```

which is printed as

```
((A B C) B C)
```

by Print could be printed by PrintX as

```
((A %L1, B C) . %L1)
```

A label set with a comma (rather than a colon) is a label for part of a list, not for the sublist.

PrintX uses the globals PrinLevel and PrinLength to control the number of levels of an object that get printed and the number of items of a list or vector that get printed, respectively. See Chapter 10 for a fuller description.

!*SAVENAMES [Initially: NIL] switch

> If on, names assigned to substructures by PrintX are retained from one use to the next. Thus substructures common to different items will be shown as the same.

13.12. Internals and Customization

This section describes some internal details of the debug package which may be useful in customizing it for specific applications. The reader is urged to consult the source for further details.

13.12.1. User Hooks

These are all global variables whose values are normally NIL. If non-NIL, they should be exprs taking the number of variables specified, and are called as specified.

PUTDHOOK!* [Initially: NIL] global

> Takes one argument, the function name. It is called after the function has been defined, and any tracing under the influence of !*TRACEALL or !*INSTALL has taken place. It is not called if the function cannot be

defined (as might happen if the function has been flagged LOSE).

TRACENTRYHOOK!* [Initially: NIL] global

Takes two arguments, the function name and a list of the actual arguments. It is called by the trace package if a traced function is entered, but before it is executed. The execution of a surrounding EMB function takes place after TRACENTRYHOOK!* is called. This is useful if you need to call special user-provided print routines to display critical data structures, as are TRACEXITHOOK!* and TRACEXPANDHOOK!*.

TRACEXITHOOK!* [Initially: NIL] global

Takes two arguments, the function name and the value. It is called after the function has been evaluated.

TRACEXPANDHOOK!* [Initially: NIL] global

Takes two arguments, the function name and the macro expansion. It is only called for macros, and is called after the macro is expanded, but before the expansion has been evaluated.

TRINSTALLHOOK!* [Initially: NIL] global

Takes one argument, a function name. It is called if a function is redefined by the Debug package, as for example when it is first traced. It is called before the redefinition takes place.

13.12.2. Functions Used for Printing/Reading

These should all contain EXPRS taking the specified number of arguments. The initial values are given in square brackets.

PPFPRINTER!* [Initially: PRINT] global

Takes one argument. It is used by Ppf to print the body of an interpreted function.

PROPERTYPRINTER!* [Initially: PRETTYPRINT] global

Takes one argument. It is used by PList to print the values of properties.

STUBPRINTER!* [Initially: PRINTX] global

Takes one argument. Stubs defined with Stub/FStub use it to print their arguments.

STUBREADER!* [Initially: !-REDREADER] global

Takes no arguments. Stubs defined with Stub/FStub use it to read their return value.

TREXPRINTER!* [Initially: PRINT] global

Takes one argument. It is used to print the expansions of traced macros.

TRPRINTER!* [Initially: PRINTX] global

Takes one argument. It is used to print the arguments and values of traced functions.

TRSPACE!* [Initially: 0] global

Controls indentation.

13.13. Example

This contrived example demonstrates many of the available features. It is a transcript of an actual PSL session.

```
@PSL
PSL 3.1, 15-Nov-82
1 lisp> (LOAD DEBUG)
NIL
2 lisp> (DE FOO (N)
2 lisp>  (PROG (A)
2 lisp>   (COND ((AND (NEQ (REMAINDER N 2) 0) (LESSP N 0))
2 lisp>                 (SETQ A (CAR N)))) %Should err out if N is a number
2 lisp>   (COND ((EQUAL N 0) (RETURN 'BOTTOM)))
2 lisp>   (SETQ N (DIFFERENCE N 2))
2 lisp>   (SETQ A (BAR N))
2 lisp>   (SETQ N (DIFFERENCE N 2))
2 lisp>   (RETURN (LIST A (BAR N) A))))
FOO
3 lisp> (DE FOOBAR (N)
3 lisp>  (PROGN (FOO N) NIL))
FOOBAR
4 lisp> (TR FOO FOOBAR)
(FOO FOOBAR)
5 lisp> (PPF FOOBAR FOO)


EXPR procedure FOOBAR(N) [TRACED;Invoked 0 times]:
PROGN
(FOO N)
NIL


EXPR procedure FOO(N) [TRACED;Invoked 0 times]:
PROG
(A)
(COND ((AND (NEQ (REMAINDER N 2) 0) (LESSP N 0)) (SETQ A (CAR N))))
(COND ((EQUAL N 0) (RETURN 'BOTTOM)))
(SETQ N (DIFFERENCE N 2))
(SETQ A (BAR N))
(SETQ N (DIFFERENCE N 2))
(RETURN (LIST A (BAR N) A))

(FOOBAR FOO)
6 lisp> (ON COMP)
NIL
7 lisp> (DE BAR (N)
7 lisp>  (COND ((EQUAL (REMAINDER N 2) 0) (FOO (TIMES 2 (QUOTIENT N 4))))
7 lisp>        (T (FOO (SUB1 (TIMES 2 (QUOTIENT N 4)))))))
*** (BAR): base 275266, length 21 words
BAR
```

```
8 lisp> (OFF COMP)
NIL
9 lisp> (FOOBAR 8)
FOOBAR being entered
   N:   8
  FOO being entered
     N: 8
    FOO (level 2) being entered
       N:       2
      FOO (level 3) being entered
         N:     0
      FOO (level 3) = BOTTOM
      FOO (level 3) being entered
         N:     0
      FOO (level 3) = BOTTOM
    FOO (level 2) = (BOTTOM BOTTOM BOTTOM)
    FOO (level 2) being entered
       N:       2
      FOO (level 3) being entered
         N:     0
      FOO (level 3) = BOTTOM
      FOO (level 3) being entered
         N:     0
      FOO (level 3) = BOTTOM
    FOO (level 2) = (BOTTOM BOTTOM BOTTOM)
  FOO = (%L1: (BOTTOM BOTTOM BOTTOM) (BOTTOM BOTTOM BOTTOM)
%L1)
FOOBAR = NIL
NIL
10 lisp> % Notice how in the above PRINTX printed the return values
10 lisp> % to show shared structure
10 lisp> (TRST FOO)
(FOO)
11 lisp> (FOOBAR 8)
FOOBAR being entered
   N:   8
  FOO being entered
     N: 8
  N := 6
    FOO (level 2) being entered
       N:       2
    N := 0
      FOO (level 3) being entered
         N:     0
      FOO (level 3) = BOTTOM
    A := BOTTOM
```

```
      N := -2
        FOO (level 3) being entered
           N:     0
        FOO (level 3) = BOTTOM
      FOO (level 2) = (BOTTOM BOTTOM BOTTOM)
    A := (BOTTOM BOTTOM BOTTOM)
    N := 4
      FOO (level 2) being entered
         N:       2
      N := 0
        FOO (level 3) being entered
           N:     0
        FOO (level 3) = BOTTOM
      A := BOTTOM
      N := -2
        FOO (level 3) being entered
           N:     0
        FOO (level 3) = BOTTOM
      FOO (level 2) = (BOTTOM BOTTOM BOTTOM)
    FOO = (%L1: (BOTTOM BOTTOM BOTTOM) (BOTTOM BOTTOM BOTTOM)
  %L1)
  FOOBAR = NIL
  NIL
  12 lisp> (TR BAR)
  (BAR)
  13 lisp> (FOOBAR 8)
  FOOBAR being entered
     N:   8
    FOO being entered
       N: 8
      BAR being entered
         A1:      6
        FOO (level 2) being entered
           N:     2
          BAR (level 2) being entered
             A1:  0
            FOO (level 3) being entered
               N: 0
            FOO (level 3) = BOTTOM
          BAR (level 2) = BOTTOM
          BAR (level 2) being entered
             A1:  -2
            FOO (level 3) being entered
               N: 0
            FOO (level 3) = BOTTOM
          BAR (level 2) = BOTTOM
          FOO (level 2) = (BOTTOM BOTTOM BOTTOM)
        BAR = (BOTTOM BOTTOM BOTTOM)
        BAR being entered
           A1:     4
          FOO (level 2) being entered
             N:    2
            BAR (level 2) being entered
               A1:  0
              FOO (level 3) being entered
                 N: 0
              FOO (level 3) = BOTTOM
            BAR (level 2) = BOTTOM
            BAR (level 2) being entered
               A1:  -2
              FOO (level 3) being entered
                 N: 0
              FOO (level 3) = BOTTOM
            BAR (level 2) = BOTTOM
          FOO (level 2) = (BOTTOM BOTTOM BOTTOM)
        BAR = (BOTTOM BOTTOM BOTTOM)
      FOO = (%L1: (BOTTOM BOTTOM BOTTOM) (BOTTOM BOTTOM BOTTOM)
    %L1)
    FOOBAR = NIL
    NIL
    14 lisp> (OFF TRACE)
    NIL
    15 lisp> (FOOBAR 8)
    NIL
    16 lisp> (TR)
    *** Start of saved trace information ***
            BAR (level 2) = BOTTOM
          FOO (level 2) = (BOTTOM BOTTOM BOTTOM)
        BAR = (BOTTOM BOTTOM BOTTOM)
      FOO = (%L1: (BOTTOM BOTTOM BOTTOM) (BOTTOM BOTTOM BOTTOM)
    %L1)
    FOOBAR = NIL
    *** End of saved trace information ***
    NIL
    17 lisp> (FOOBAR 13)
    ***** An attempt was made to do CAR on '-1', which is not a pair
    Break loop
    18 lisp break>> Q
    19 lisp> (TR)
    *** Start of saved trace information ***
      FOO being entered
         N: 13
      BAR being entered
         A1:      11
        FOO (level 2) being entered
           N:     3
          BAR (level 2) being entered
             A1:  1
            FOO (level 3) being entered
               N: -1
*** End of saved trace information ***
NIL
20 lisp> (BTR)
*** Backtrace: ***
These functions were left abnormally:
  FOO
     N: -1
  BAR
     A1:        1
  FOO
     N: 3
  BAR
     A1:        11
  FOO
     N: 13
  FOOBAR
     N: 13
*** End of backtrace ***
NIL
21 lisp> (STUB (FOO N))
*** Function 'FOO' has been redefined
NIL
22 lisp> (FOOBAR 13)
 Stub FOO called

N: 13
Return? :
22 lisp> (BAR (DIFFERENCE N 2))
 Stub FOO called

N: 3
Return? :
22 lisp> (BAR (DIFFERENCE N 2))
 Stub FOO called

N: -1
Return? :
22 lisp> 'ERROR
```

```
NIL
23 lisp> (TR)
*** Start of saved trace information ***
  BAR being entered
     A1:        11
    BAR (level 2) being entered
       A1:      1
    BAR (level 2) = ERROR
  BAR = ERROR
FOOBAR = NIL
*** End of saved trace information ***
NIL
24 lisp> (OFF TRCOUNT)


FOOBAR(6)           ******************
BAR(16)             *******************************************************


NIL
22 lisp> (QUIT)
```

CHAPTER 14
MISCELLANEOUS USEFUL FEATURES

14.1. The HELP Mechanism

(Help [TOPICS:id]): NIL <u>fexpr</u>

If no arguments are given, a message describing Help itself and known topics is printed. Otherwise, each of the <u>id</u> arguments is checked to see if any help information is available. If it has a value under the property list indicator HelpFunction, that function is called. If it has a value under the indicator HelpString, the value is printed. If it has a value under the indicator HelpFile, the file is displayed on the terminal. By default, a file called "topic.HLP" on the logical device, "PH:" is looked for, and printed if found.

Help also prints out the values of the Top Loop fluids, and finally searches the current Id-Hash-Table for loaded modules.

HelpIn!* [<u>Initially:</u> NIL] <u>global</u>

The channel used for input by the Help mechanism.

HelpOut!* [<u>Initially:</u> NIL] <u>global</u>

The channel used for output by the Help mechanism.

14.2. Exiting PSL

The normal way to suspend PSL execution is to call the Quit function or to strike <Ctrl-C> on the DEC-20 or <Ctrl-Z> on the VAX.

(Quit): Undefined expr

Return from LISP to superior process. If the operating system permits a choice, QUIT is a continuable exit, and EXITLISP is a permanent exit (that terminates the PSL process).

(ExitLisp): Undefined expr

Return from LISP to superior process. If the operating system permits a choice, QUIT is a continuable exit, and EXITLISP is a permanent exit (that terminates the PSL process).

14.3. 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 Usermode!* to T.

The forms in the list FORMS will be evaluated when the new core image is started. For example

```
(SaveSystem "PSL 3.1" "PSL.EXE" '((Read-Init-File "PSL")
      (InitializeInterrupts)))
```

If RLISP has been loaded, SaveSystem will have been redefined to save the message in the global variable date!*, and redefine Main to call RlispMain, which uses date!* in Begin1. The older SaveSystem will be saved as the function LispSaveSystem.

LispBanner!* [Initially:] global

Records the welcome message given by a call to SaveSystem from PSL. Also contains the date, given by the function Date.

Date!* [Initially: Nil] global

Records the welcome message given by a call to SaveSystem from RLISP.

(DumpLisp FILE:string): Undefined expr

This calls Reclaim to compact the heap, and unmaps the unused pages (DEC-20) or moves various segment pointers (VAX) to decrease the core image. The core image is then written as an executable file, with the name FILE.

14.4. Init Files

Init files are available to make it easier for the user to customize PSL to his/her own needs. When PSL, RLISP, or PSLCOMP is executed, if a file PSL.INIT, RLISP.INIT, or PSLCOMP.INIT (.pslrc, rlisprc, or .pslcomprc on the VAX) is on the home directory, it will be read and evaluated. Currently all init files must be written in LISP syntax. They may use FASLIN or LOAD as needed.

The following functions are used to implement init files, and can be accessed by LOADing the INIT-FILE module.

(User-HomeDir-String): string expr

Returns a full pathname for the user's home directory.

(Init-File-String PROGRAMNAME:string): string expr

Returns the full pathname of the user's init file for the program PROGRAMNAME.

```
(Init-File-String  "PSL")
```

(Read-Init-File PROGRAMNAME:string): Nil expr

Reads and evaluates the init file with name PROGRAMNAME. Read-Init-File calls Init-File-String with argument PROGRAMNAME.

```
(Read-Init-File "PSL")
```

14.5. Miscellaneous Functions

(Reset): Undefined expr

Return to top level of LISP. Similar to <Ctrl-C> and Start on the DEC-20, but with the reset function, 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 16-Dec-82.

14.6. Garbage Collection

(Reclaim): Undefined expr

Reclaim is the user level call to the garbage collector. Internal system functions always use !%Reclaim.

(!%Reclaim): Undefined expr

!%Reclaim is is used within the system to call 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!*.

!*GC [Initially: NIL] switch

!*GC controls the printing of garbage collector messages. If NIL, no indication of garbage collection occurs. If non-NIL various system dependent messages may be displayed.

GCTime!* [Initially:] global

Time spent in garbage collection. Cumulative (but starting from when?).

GCKNT!* [Initially: 0] global

Records the number of times that Reclaim has been called to this point. GCKNT!* may be reset to another value to record counts incrementally, as desired.

CHAPTER 15
COMPILER

15.1. Introduction

The functions and facilities in the PSL LISP/SYSLISP compiler and supporting loaders (LAP and FASL) are described in this chapter.

15.2. The Compiler

The compiler is a version of the Portable LISP Compiler [Griss 81], modified and extended[1] to more efficiently support both LISP and SYSLISP compilation. See the later sections in this chapter and references [Griss 81] and [Benson 81] for more details.

15.2.1. Compiling Files

On some computer systems it is possible to compile a file by invoking PSLCOMP with a command line argument specifying the name of the file to be compiled. The Compile-File function is executed; see immediately below for a description of its behavior. When PSLCOMP is invoked with a command line argument, no break loop is entered in case of error, but the error message is printed along with a warning from the compiler and the compilation aborts.

---

[1]Many of the recent extensions to the PLC were implemented by John Peterson.

(Compile-File FILE:string): undefined expr

Load compl-extra to get this function.

Compiles a single file, producing a .B file of the same name. The .B file is written with the same directory specification as in the argument to compile-file. If you supply no suffix to compile-file, it will search for a source file with the name you specified and with one of the suffixes ".BUILD", ".SL", or ".RED", in that order. The compile-file function assumes that files with ".BUILD" or ".RED" suffix are in RLISP syntax. If the filename is given with a "random" extension, syntax is assumed to be LISP.

The conservative approach is to supply the suffix explicitly. This avoids some technical pitfalls related to "long" filenames that an operating system may truncate.

15.2.2. Compiling Functions into FASL Files

In order to produce files that may be input using Load or FaslIn, the FaslOut and FaslEnd pair may be used.

(FaslOut FILE:string): NIL expr

(FaslEnd): NIL expr

After the command FaslOut has been given, all subsequent S-expressions and function definitions typed in or input from files are processed by the Compiler, LAP and FASL as needed, and output to FILE. Functions are compiled and partially assembled, and output as in a compressed binary form, involving blocks of code and relocation bits. This activity continues until the function FaslEnd terminates this process. Note that a "b" file extension is automatically appended to the output file.

The FaslOut and FaslEnd pair also use the DFPRINT!* mechanism, turning on the switch !*DEFN, and redefining DFPRINT!* to trap the parsed input in the RLISP top-loop. Currently this is not usable from pure LISP level.

[??? Fix, by adding !*DEFN mechanism to basic top-loop. ???]

15.2.3. Compiling Functions into Memory

Functions can be compiled directly into memory using a loaded interpretive definition.

!*COMP [Initially: NIL] switch

If the compiler is loaded (which is usually the case, otherwise load the COMPILER module), turning on the switch !*COMP causes all subsequent procedure definitions of appropriate type to be compiled automatically and a message of the form

```
<function-name> COMPILED, <words> WORDS, <words> LEFT
```

to be printed. The first number is the number of words of binary program space the compiled function took, and the second number the number of words left unused in binary program space.

Currently, exprs, fexprs, nexprs and macros may be compiled. This is controlled by a flag ('COMPILE) on the property list of the procedure type.

If desired, uncompiled functions already resident may be compiled by using

(Compile NAMES:id-list): any expr

Compiling into memory can be particularly useful as a way of checking the efficiency (and correctness) of code generated by the compiler. The switches !*PLAP and !*PGWD control printing of the LAP (assembly) code generated by the compiler. See their documentation for details.

15.2.4. Fluid and Global Declarations

The FLUID and GLOBAL declarations must be used to indicate variables that are to be used as non-LOCALs in compiled code. Currently, the compiler defaults variables bound in a particular procedure to LOCAL. The effect of this is that the variable only exists as an "anonymous" stack location; its name is compiled away and called routines cannot see it (i.e., they would have to use the name). Undeclared non-LOCAL variables are automatically declared FLUID by the compiler with a warning. In many cases, this means that a previous procedure that bound this variable should have known about this as a FLUID. Declare it with FLUID, below, and recompile, since the caller cannot be automatically fixed.

[??? Should we provide an !*AllFluid switch to make the default Fluid, or should we make Interpreter have a LOCAL variable as default, or both ???]

Declaring a variable to be FLUID or GLOBAL causes the variable to be initialized at the point of declaration with the value NIL unless it already has a value at that point.

(Fluid NAMES:id-list): any expr

Declares each variable FLUID (if not previously declared); this means that it can be used as a Prog LOCAL, or as a parameter. On entry to the procedure, its current value is saved and all access is always to the VALUE cell of the variable; on exit (or Throw or Error), the old values are restored.

(Global NAMES:id-list): any expr

Declares each variable GLOBAL (if not previously declared); this means that it cannot be used as a LOCAL, or as a parameter. Access is always to the VALUE cell (SYMVAL) of the variable.

[??? Should we eliminate GLOBALs ???]

15.2.5. Conditional Compilation

(If_System SYS-NAME:id, TRUE-CASE:any, FALSE-CASE:any): any macro

This is a compile-time conditional macro for system-dependent code. FALSE-CASE can be omitted and defaults to NIL. SYS-NAME must be a member of the fluid variable System_List!*. For the Dec-20, System_List!* is (Dec20 PDP10 Tops20 KL10). For the VAX it is (VAX Unix VMUnix). An example of its use follows.

```
PROCEDURE MAIL();
IF_SYSTEM(TOPS20, RUNFORK "SYS:MM.EXE",
    IF_SYSTEM(UNIX, SYSTEM "/BIN/MAIL",
              STDERROR "MAIL COMMAND NOT IMPLEMENTED"));
```

15.2.6. Functions to Control the Time When Something is Done

Which expressions are evaluated during compilation only, which output to the file for load time evaluation, and which do both (such as macro definitions) can be controlled by the properties 'EVAL and 'IGNORE on certain function names, or the following functions.

(CommentOutCode U:form): NIL macro

Comment out a single expression; use <<U>> to comment out a block of code.

(CompileTime U:form): NIL expr

Evaluate the expression U at compile time only, such as defining auxiliary smacros and macros that should not go into the file.

Certain functions have the flag 'IGNORE on their property lists to achieve the same effect. E.g. FLAG('(LAPOUT LAPEND),'IGNORE) has been done.

(BothTimes U:form): U:form expr

Evaluate at compile and load time. This is equivalent in effect to executing Flag('(f1 f2),'EVAL) for certain functions.

(LoadTime U:form): U:form expr

Evaluate at load time only. Should not even compile code, just pass direct to file.

[??? EVAL and IGNORE are for compatibility, and enable the above sort of functions to be easily written. The user should avoid EVAL and IGNORE flags, if possible ???]

15.2.7. Order of Functions for Compilation

Non-expr procedures must be defined before their use in a compiled function, since the compiler treats the various function types differently. Macros are expanded and then compiled; the argument list of fexprs quoted; the arguments of nexprs are collected into a single list. Sometimes it is convenient to define a dummy version of the function of appropriate type, to be redefined later. This acts as an "External or Forward" declaration of the function.

[??? Add such a declaration. ???]

15.2.8. Switches Controlling Compiler

The compilation process is controlled by a number of switches, as well as the above declarations and the !*COMP switch, of course.

!*R2I [Initially: T] switch

> If T, causes recursion removal if possible, converting recursive calls on a function into a jump to its start. If this is not possible, it uses a faster call to its own "internal" entry, rather than going via the Symbol Table function cell. The effect in both cases is that tracing this function does not show the internal or eliminated recursive calls, nor the backtrace information.

!*NOLINKE [Initially: NIL] switch

> If T, inhibits use of !*LINKE cmacro. If NIL, "exit" calls on functions that would then immediately return. For example, the calls on FOO(x) and FEE(X) in

```
   PROCEDURE DUM(X,Y);
    IF X=Y THEN FOO(X) ELSE FEE(X+Y);
```

> can be converted into direct JUMP's to FEE or FOO's entry point. This is known as a "tail-recursive" call being converted to a jump. If this happens, there is no indication of the call of DUM on the backtrace stack if FEE or FOO cause an error.

!*ORD [Initially: NIL] switch

> If T, forces the compiler to compile arguments in Left-Right Order, even though more optimal code can be generated.
>
> [??? !*ORD currently has a bug, and may not be fixed for some time. Thus do NOT depend on evaluation order in argument lists ???]

!*MODULE [Initially: NIL] switch

> Indicates block compilation (a future extension of this compiler). When implemented, even more function and variable names are "compiled away".

Technically the following switches are part of the loader. See the documentation of compiler and loader implementation also.

!*PLAP [Initially: NIL] switch

If T, causes the printing of the portable cmacros produced by the the compiler. In LAP, causes LAP forms to printed before expansion. Used mainly to see output of compiler before assembly.

!*PGWD [Initially: NIL] switch

Causes LAP to print the actual DEC-20 mnemonics and corresponding assembled instruction in octal, displaying OPCODE, REGISTER, INDIRECT, INDEX and ADDRESS fields. Affects printing during compilation.

!*PCMAC [Initially: NIL] switch

A combination of !*PLAP and !*PGWD.

!*PWRDS [Initially: T] switch

Prints out the address and size of each compiled function.

15.2.9. Differences between Compiled and Interpreted Code

The following just re-iterates some of the points made above and in other sections of the manual regarding the "obscure" differences that compilation introduces.

[??? This needs some careful work, and perhaps some effort to reduce the list of differences ???]

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. If called from compiled code, the call on an open-compiled function is replaced by a series of online instructions. Most of these functions have some sort of indicator on their property lists: 'OPEN, 'ANYREG, 'CMACRO, 'COMPFN, etc. For example: SETQ, CAR, CDR, COND, WPLUS2, MAP functions, PROG, PROGN, etc. Also note that some functions are defined as macros, which convert to some other form (such as PROG), which itself might compile open.

Some optimizations are performed that cause inaccessible or redundant code to be removed, e.g. 0*foo(x) could cause foo(x) not to be called.

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 are not visible on the Binding Stack. Further more, these variables are NOT available to functions called in the dynamic scope of the function containing their binding.

Since compiled calls on macros, fexprs and nexprs are different from the default exprs, these functions must be declared (or defined) before compiling the code that uses them. While fexprs and nexprs may subsequently be 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 of course picks up the most recent definition of ANY function, and so functions can switch type as well as body.

[??? If we expand macros at PUTD time, then this difference will go away. ???]

As noted above, the !*R2I, !*NOLINKE and !*MODULE switches cause certain functions to call other functions (or themselves usually) by a faster route (JUMP or internal call). This means that the recursion or call may not be visible during tracing or backtrace.

15.2.10. Compiler Errors

A number of compiler errors are listed below with possible explanations of the error.

*** Function form converted to APPLY

This message indicates that the Car of a form is either

a. Non-atomic,
b. a local variable, or
c. a global or fluid variable.

The compiler converts (F X1 X2 ...), where F is one of the above, to (APPLY F (LIST X1 X2 ...)).

*** NAME already SYSLISP non-local

This indicates that NAME is either a WVAR or WARRAY in SYSLISP mode, but is being used as a local variable in LISP mode. No special action is taken.

*** WVAR NAME used as local

This indicates that NAME is a WVAR, but is being used as a bound variable in SYSLISP mode. The variable is treated as an an anonymous local variable within the scope of its binding.

*** NAME already SYSLISP non-local

This indicates that a variable was previously declared as a SYSLISP WVAR or WARRAY and is now being used as a LISP fluid or global. No special action is taken.

*** NAME already LISP non-local

This indicates that a variable was previously declared as a LISP fluid or global and is now being used as a SYSLISP WVAR or WARRAY. No special action is taken.

*** Undefined symbol NAME in Syslisp, treated as WVAR

A variable was encountered in SYSLISP mode which is not local nor a WVAR or WARRAY. The compiler declares it a WVAR. This is an error, all WVARs should be explicitly declared.

*** NAME declared fluid

A variable was encountered in LISP mode which is not local nor a previously declared fluid or global. The compiler declares it fluid. This is sometimes an error, if the variable was used strictly locally in an earlier function definition, but was intended to be bound non-locally. All fluids should be declared before being used.

CHAPTER 16
BIBLIOGRAPHY

The following books and articles either are directly referred to in the manual text, or will be helpful for supplementary reading.

[Allen 79] Allen, J. R.
The Anatomy of LISP.
McGraw-Hill, New York, New York, 1979.

[Baker 78] Baker, H. G.
Shallow Binding in LISP 1.5.
CACM 21(7):565, July, 1978.

[Benson 81] Benson, E. and Griss, M. L.
SYSLISP: A Portable LISP Based Systems Implementation Language.
Utah Symbolic Computation Group Report UCP-81, University of Utah, Department of Computer Science, February, 1981.

[Bobrow 76] Bobrow, R. J.; Burton, R. R.; Jacobs, J. M.; and Lewis, D.
UCI LISP MANUAL (revised).
Online Manual RS:UCLSP.MAN, University of California, Irvine, ??, 1976.

[Charniak 80] Charniak, E.; Riesbeck, C. K.; and McDermott, D. V.
Artificial Intelligence Programming.
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

[Fitch 77] Fitch, J. and Norman, A.
Implementing LISP in a High Level Language.
Software: Practise and Experience 7:713-xx, 1977.

[Foderaro 81] Foderaro, J. K. and Sklower, K. L.
The Franz LISP Manual.
, 1981.

[Frick 78] Frick, I. B.
Manual for Standard LISP on the DECSYSTEM 10 and 20.
Utah Symbolic Computation Group Technical Report TR-2, University of Utah, Department of Computer Science, July, 1978.

[Griss 77a] Griss, M. L.
BIL: A Portable Implementation Language for LISP-Like Systems.
Utah Symbolic Computation Group Opnote No. 36, University of Utah, Department of Computer Science, 1977.

[Griss 77b] Griss, M. L. and Swanson, M. R.
MBALM/1700 : A Micro-coded LISP Machine for the Burroughs B1726.
In Proceedings of Micro-10 ACM, pages 15. ACM, 1977.

[Griss 78a] Griss, M. L. and Kessler, R. R.
REDUCE 1700: A Micro-coded Algebra System.
In Proceedings of The 11th Annual Microprogramming Workshop, pages 130-138. IEEE, November, 1978.

[Griss 78b] Griss, M. L.
MBALM/BIL: A Portable LISP Interpreter.
Utah Symbolic Computation Group Opnote No. 38, University of Utah, Department of Computer Science, 1978.

[Griss 79a] Griss, M. L.; Kessler, R. R.; and Maguire, G. Q. Jr.
TLISP - A Portable LISP Implemented in P-code.
In Proceedings of EUROSAM 79, pages 490-502. ACM, June, 1979.

[Griss 79b] Griss, M. L. and Kessler, R. R.
A Microprogrammed Implementation of LISP and REDUCE on the Burroughs B1700/B1800 Computer.
Utah Symbolic Computation Group Report UCP 70, University of Utah, Department of Computer Science, 1979.

[Griss 81] Griss, M. L. and Hearn, A. C.
A Portable LISP Compiler.
Software - Practice and Experience 11:541-605, June, 1981.

[Griss 82] Griss, M. L.; Benson. E.; and Hearn, A. C.
Current Status of a Portable LISP Compiler.
In Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction, pages 276-283. ACM SIGPLAN, June, 1982.

{Harrison 73] Harrison, M. C.
Data structures and Programming.
Scott, Foresman and Company, Glenview, Illinois, 1973.

[Harrison 74] Harrison, M. C.
A Language Oriented Instruction Set for BALM.
In Proceedings of SIGPLAN/SIGMICRO 9, pages 161. ACM, 1974.

[Hearn 66] Hearn, A. C.
Standard LISP.
SIGPLAN Notices Notices 4(9):xx, September, 1966.
Also Published in SIGSAM Bulletin, ACM Vol. 13, 1969, p. 28-49. .

[Hearn 73] Hearn, A. C.
REDUCE 2 Users Manual.
Utah Symbolic Computation Group, Report UCP-19, University of Utah, Department of Computer Science, 1973.

[Kessler 79] Kessler, R. R.
PMETA - Pattern Matching META/REDUCE.
Utah Symbolic Computation Group, OpNote 40, University of Utah, Department of Computer Science, January, 1979.

[Lefaivre 78] Lefaivre, R.
RUTGERS/UCI LISP MANUAL.
Online Manual, RS:RUTLSP.MAN, Rutgers University, Computer Science Department, May, 1978.

[LISP360 xx] xx.
LISP/360 Reference Manual.
Technical Report, Stanford Centre for Information Processing, Stanford University, xx.

[MACLISP 76] xx.
MACLISP Reference Manual.
Technical Report, MIT, March, 1976.

[Marti 79] Marti, J. B., et al.
Standard LISP Report.
SIGPLAN Notices 14(10):48-68, October, 1979.

[McCarthy 73] McCarthy, J. C. et al.
LISP 1.5 Programmer's Manual.
M.I.T. Press, 1973.
7th Printing January 1973.

[Moore 76] J. Strother Moore II.
The INTERLISP Virtual Machine Specification.
CSL 76-5, Xerox, Palo Alto Research Center, 3333 Coyote Road,etc, September, 1976.

[Nordstrom 73] Nordstrom, M.
A Parsing Technique.
Utah Computational Physics Group Opnote No. 12, University of Utah, Department of Computer Science, November, 1973.

[Nordstrom 78] Nordstrom, M.; Sandewall, E.; and Breslaw, D.
LISP F3 : A FORTRAN Implementation of InterLISP.
Manual, Datalogilaboratoriet, Sturegatan 2 B, S 752 23, Uppsala, SWEDEN, 1978.
Mentioned by M. Nordstrom in 'Short Announcement of LISP F3', a handout at LISP80.

[Norman 81] Norman, A.C. and Morrison, D. F.
The REDUCE Debugging Package.
Utah Symbolic Computation Group Opnote No. 49, University of Utah, Department of Computer Science, February, 1981.

[Pratt 73] Pratt, V.
Top Down Operator Precedence.
In Proceedings of POPL-1, pages ??-??. ACM, 1973.

[Quam 69] Quam, L. H. and Diffie, W.
Stanford LISP 1.6 Manual.
Operating Note 28.7, Stanford Artificial Intelligence Laboratory, 1969.

[Sandewall 78] Sandewall, E.
Programming in an Interactive Environment : The LISP Experience.
Computing Surveys 10(1):35-72, March, 1978.

[Steele 81] Steele, G. L. and Fahlman, S. E.
Spice LISP Reference Manual.
Manual , Carnegie-Mellon University, Pittsburgh, September, 1981.
(Preliminary Common LISP Report).

[Teitelman 78] Teitelman, W.; et al.
Interlisp Reference Manual, (3rd Revision).
Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto,Calif. 94304, 1978.

[Teitelman 81] Teitleman, W. and Masinter, L.
The InterLISP Programming Environment.
IEEE Computer 14(4):25-34, 1981.

[Terashima 78] Terashima, M. and Goto, E.
Genetic Order and Compactifying Garbage Collectors.
Information Processing Letters 7(1):27-32, 1978.

[Weinreb 81] Weinreb, D. and Moon, D.
LISP Machine Manual.
, 1981.
Fourth edition.

[Weissman 67] Weissman.
LISP 1.5 Primer.
Dickenson Publishing Company, Inc., 1967.

[Winston 81] Winston, P. H., and Horn, B. K. P.
LISP.
Addison-Wesley Publishing Company, Reading, Mass., 1981.

CHAPTER 17
INDEX OF CONCEPTS

The following is an alphabetical list of concepts, with the page on which they are discussed.

CHAPTER 18
INDEX OF FUNCTIONS

The following is an alphabetical list of the PSL functions, with the page on which they are defined.

CHAPTER 19
INDEX OF GLOBALS AND SWITCHES

The following is an alphabetical list of the PSL global variables, with the page on which they are defined.

The Portable Standard LISP Users Manual

Part 2: Utilities

by
The Utah Symbolic Computation Group

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Version 3.2: 16 March 1984

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 the DEC-20, VAX, and MC68000.

Work supported in part by the Hewlett Packard Company, the International Business Machines Corporation and the National Science Foundation under Grant Numbers MCS80-07034 and MCS82-04247.

PREFACE

Part 2 of The Portable Standard Lisp User's Manual contains information about various utilities. It includes two chapters describing some of the many utility packages available on the utility directory. Many of these are not documented. A list of these undocumented utilities is given in Chapter 22. An objects package is described in Chapter 23.

Chapter 24 describes three editors that are no longer widely used: a simple structure editor; EMODE, an EMACS-like screen editor; and a full structure editor adapted from UCI Lisp. Most PSL users now make use of NMODE, a screen editor based on EMODE. Its documentation is available separately.

PSL gives the user the option of using an Algol-like syntax called RLisp. Chapter 25 describes the syntax of RLisp. Chapter 26 describes two parser writing tools: an extensible table-driven parser that is used for the RLisp parser, and the MINI parser.

Sections 1.2 and 1.3 and Chapter 3 were contributed by Cris Perdue, Alan Snyder, and other members of the Hewlett-Packard Research Center in Palo Alto.

TABLE OF CONTENTS

CHAPTER 1
MAJOR UTILITIES

1.1. Introduction

This chapter describes some of the larger utility packages available in PSL. Its purpose is to record the existence and capabilities of a number of tools. More information on existing packages can sometimes be found by looking at the current set of HELP files (in PH: on the DEC-20, $ph on the VAX).

1.2. Fast Numeric Operators

1.2.1. Introduction

The library module NUMERIC-OPERATORS defines a set of arithmetic functions that are a superset of the numeric operators defined by the Common LISP compatibility package. The operators are described in full detail here. All the operators in this package share the characteristic of a short name made of non-alphabetic characters, such as "+". They also all cause the compiler to generate very efficient integer arithmetic code for their occurrences when the switch "fast-integers" is turned on.

This module also modifies the FOR macro to use the numeric operators to implement the FROM clause; thus, FOR statement FROM clauses will use fast integer arithmetic when the FAST-INTEGERS switch is on.

The consequences of turning on the "fast-integers" switch are discussed in a separate section, below. The individual descriptions of the operators describe behavior with that switch off. Note that when we say certain argument values are "incorrect", we mean that the result is undefined. The implementation may or may not check for that situation and an error may or may not be signalled.

1.2.2. Common LISP operators

(= X:number Y:number): number expr

Numeric Equal. True if and only 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): number expr

Numeric Not Equal. True if X and Y are numbers of equal type and value; NIL if X and Y are numbers of unequal type or value. it is incorrect to supply a non-numeric argument. Agrees with the Common LISP operator if given two numeric arguments of the same type.

(< X:number Y:number): number expr

Numeric Less Than. True if X is less than Y, regardless of type. An error is signalled if either argument is not numeric. Agrees with Common LISP if two arguments are supplied.

(> X:number Y:number): number expr

Numeric Greater Than. True if X is greater than Y, regardless of type. An error is signalled if either argument is not numeric. Agrees with Common LISP if two arguments are supplied.

(<= X:number Y:number): number 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. Agrees with Common LISP if two arguments are supplied.

(>= X:number Y:number): number 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. Agrees with Common LISP if two arguments are supplied.

(+ [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. This is defined to agree with the definition of the Common LISP operator except that this is a macro.

(- 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. Agrees with the Common LISP operator except in that this operator is a macro.

(* [N:number]): number macro

Numeric Multiplication. 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 1. This is defined to agree with the definition of the Common LISP operator except that this is a macro.

(/ 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 subtracting succeeding arguments from the first argument. Signals an error if no arguments are supplied or if any argument is non-numeric. Agrees with the Common LISP operator except in that this operator is a macro.

1.2.3. Operators Not in Common LISP

(~= 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)).

1.2.4. The Fast-Integers Switch

Fast-Integers [Initially:]   switch

At compilation time the switch FAST-INTEGERS controls generation by the compiler of efficient, unchecked, inline machine code for occurrences of these operators. When the switch is on, uses of these operators will compile into appropriate machine instructions of the target machine. The arguments are assumed to be integers in the "INUM" range, no larger than about plus or minus 16 million on the HP9836. No checking of argument types is done, nor is the value returned checked for being in the legal range. Floating point operands are NOT handled by code compiled with the FAST-INTEGERS switch turned on.

1.2.5. Cautions

No checking of either arguments or results is done. The code that is generated is fast and can be intermixed with other LISP arithmetic operations because a LISP number within the "INUM" range is represented the same way that the host computer represents that number. An out of range result of one of these "fast operations" should be thought of as a "garbage" value.

A "garbage" value can cause the system to damage itself when treated as a tagged item. For example, the item might appear to be a pointer to a pair. If passed to a printing routine or seen by the garbage collector it could cause machine exceptions for an illegal memory reference or attempt to access an "odd address" (operand alignment error). A copying or compacting garbage collector might relocate the value.

Note that the * and << operators are particularly likely to be dangerous in this mode because they can produce large results from small operands.

1.3. Vector Operations

1.3.1. Introduction

Here we describe the library modules SLOW-VECTORS and FAST-VECTORS. These modules provide a set of operations on vectors that can be compiled into efficient in-line machine code. The functions defined here are used extensively in the NMODE editor and other modules.

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.

In many cases one just loads FAST-VECTORS, which makes available the facilities of SLOW-VECTORS for the use of interpretive code. The FAST-VECTORS module adds no new functions. It only sets up generation of efficient code for these operations by the compiler, controllable by a switch. To use these functions, load either module. To permit generation of very efficient code, load FAST-VECTORS.

1.3.2. Vector Operations

(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): any <u>expr</u>

Stores into a PSL VECTOR. Vector indexes start with 0.

(VECTOR-SIZE V:vector): integer <u>expr</u>

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 <u>expr</u>

Returns the greatest legal index for accessing or storing into a PSL VECTOR. See also just above.

(VECTOR-EMPTY? V:vector): boolean <u>expr</u>

True if the vector has at least one element, otherwise NIL.

1.3.3. The Fast-Vectors Switch

Fast-vectors [<u>Initially:</u>] <u>switch</u>

At compilation time the switch <u>FAST-INTEGERS</u> controls generation by the compiler of efficient, unchecked, inline machine code for occurrences of these operations. When the switch is on, uses of these operators will compile into appropriate machine instructions of the target machine. The switch is initially turned on when the module FAST-VECTORS is loaded, but the switch should be explicitly turned on and off within source files. A request to "(load FAST-VECTORS)" does not necessarily mean that any module will be loaded -- library modules are only loaded in response to the first request for a load.

1.3.4. Cautions

The types of the arguments are not checked. Integer arguments are assumed to be integers in the "INUM" range, no larger than about plus or minus 16 million on the HP9836. Vector arguments are assumed to be valid Lisp VECTORS. Range checking of vector indexes is not done.

All this means that it is possible to access or store into memory that is protected, does not exist, or at least does not contain an element of a vector. This is especially likely to happen if a non-integer is used as the vector index in one of these operations.

Storing into arbitrary memory locations is clearly very destructive. Accessing "garbage" values obtained from erroneous vector access is also dangerous, aside from causing your code to get incorrect results.

A "garbage" value can cause the system to damage itself when treated as a tagged item. For example, the item might appear to be a pointer to a pair. If passed to a printing routine or seen by the garbage collector it could cause machine exceptions for an illegal memory reference or attempt to access an "odd address" (operand alignment error). A copying or compacting garbage collector might relocate the value.

1.4. RCREF - Cross Reference Generator for PSL Files

RCREF is a Standard Lisp program for processing a set of Standard Lisp function definitions to produce:

a. A "Summary" showing:

   i. A list of files processed.
   ii. A list of "entry points" (functions which are not called or are called only by themselves).
   iii. A list of undefined functions (functions called but not defined in this set of functions).
   iv. A list of variables that were used non-locally but not declared GLOBAL or FLUID before their use.
   v. A list of variables that were declared GLOBAL but used as FLUIDs (i.e. bound in a function).
   vi. A list of FLUID variables that were not bound in a function so that one might consider declaring them GLOBALs.
   vii. A list of all GLOBAL variables present.
   viii. A list of all FLUID variables present.
   ix. A list of all functions present.

b. A "global variable usage" table, showing for each non-local variable:

   i. Functions in which it is used as a declared FLUID or GLOBAL.
   ii. Functions in which it is used but not declared before.
   iii. Functions in which it is bound.
   iv. Functions in which it is changed by SetQ.

c. A "function usage" table showing for each function:

   i. Where it is defined.
   ii. Functions which call this function.
   iii. Functions called by it.
   iv. 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.

1.4.1. 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.

SYSLisp procedures are not correctly analyzed.

1.4.2. 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:

```
@PSL:RLISP
LOAD RCREF;        % RCREF is now autoloading, so this may be omitted.

OUT "file.crf";    % later, CREFOUT ...
ON CREF;
IN "file1.ex1","file2.ex2";
OFF CREF;
SHUT "file.crf";   % later CREFEND
```

To process more files, more IN statements may be added, or the IN statement may be changed to include more files.

1.4.3. 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 Standard LISP 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 CADDR CDAAR CDADR CDDAR CDDDR CAAAAR CAAADR CAADAR
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 GENSYM GET GETD GETV
GLOBAL GLOBALP GO GREATERP IDP INTERN LENGTH LESSP LINELENGTH
LITER LPOSN 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.

1.5. Picture RLISP

[??? ReWrite ???]

Picture RLisp is an Algol-like graphics language for Teleray, HP2648a and Tektronix, in which graphics Model primitives are combined into complete Models for display. PRLISP is a 3D version; PRLISP2D is a faster, smaller 2D version which also drives more terminals. Two demonstration files, PR-DEMO.RED and PR-DEMO.SI, are available on PU. See the help files PH:PRLISP.HLP and PRLISP2D.HLP.

Model primitives include:

P:={x,y,z}; A point (y, and z may be omitted, default to 0).

PS:=P1_ P2_ ... Pn;
A Point Set is an ordered set of Points (Polygon).

G := PS1 & PS2 & ... PSn;
A Group of Polygons.

Point Set Modifiers
alter the interpretation of Point Sets within their scope.

BEZIER() causes the point-set to be interpreted as the specification points for a BEZIER curve, open pointset.

BSPLINE() does the same for a Bspline curve, closed pointset.

TRANSFORMS:
Mostly return a transformation matrix.

Translation: Move the specified amount along the specified axis. XMOVE(deltaX); YMOVE(deltaY); ZMOVE(deltaZ); MOVE(deltaX, deltaY, deltaZ);

Scale: Scale the Model SCALE (factor) XSCALE(factor); YSCALE(factor); ZSCALE(factor); SCALE1(x.scale.factor, y.scale.factor, z.scale.factor); SCALE<Scale factor>;. Scale along all axes.

Rotation: ROT(degrees); ROT(degrees, point.specifying.axis); XROT(degrees); YROT(degrees); ZROT(degrees);

Window (z.eye,z.screen):
The WINDOW primitives assume that the viewer is located along the z axis looking in the positive z direction, and that the viewing window is to be centered on both the x and y axis.

Vwport(leftclip,rightclip,topclip,bottomclip):
The VWPORT, which specifies the region of the screen which is used for display.

REPEATED (number.of.times, my.transform):
The Section of the Model which is contained within the scope of the Repeat Specification is replicated. Note that REPEATED is intended to duplicate a sub-image in several different places on the screen; it was not designed for animation.

Identifiers of other Models
the Model referred to is displayed as if it were part of the current Model for dynamic display.

Calls to PictureRLISP Procedures
This Model primitive allows procedure calls to be imbedded within Models. When the Model interpreter reaches the procedure identifier it calls it, passing it the portion of the Model below the procedure as an argument. The current transformation matrix and the current pen position are available to such procedures as the values of the global identifiers GLOBAL!.TRANSFORM and HEREPOINT. If normal procedure call syntax, i.e. proc.name (parameters), is used then the procedure is called at Model-building time, but if only the procedure's identifier is used then the procedure is imbedded in the Model.

ERASE() Clears the screen and leaves the cursor at the origin.

SHOW(pict) Takes a picture and displays it on the screen.

ESHOW (pict)
Erases the whole screen and display "pict".

HP!.INIT(), TEK!.INIT(), TEL!.INIT()
Initializes the operating system's view of the characteristics of HP2648A terminal, TEKTRONIX 4006-1 (also ADM-3A with Retrographics board, and Teleray-1061).

For example, the Model

```
(A _ B _ C  &  {1,2} _ B)  |  XROT (30)  |  'TRAN ;

%
% PictureRLISP Commands to SHOW lots of Cubes
%
% Outline is a Point Set defining the 20 by 20
%   square which is part of the Cubeface
%
Outline := { 10, 10} _ {-10, 10} _
          {-10,-10} _ { 10,-10} _ {10, 10};

% Cubeface also has an Arrow on it
%
Arrow := {0,-1} _ {0,2}  &  {-1,1} _ {0,2} _ {1,1};

% We are ready for the Cubeface

Cubeface   :=   (Outline & Arrow)  |  'Tranz;

% Note the use of static clustering to keep objects
%  meaningful as well as the quoted Cluster
%  to the as yet undefined transformation Tranz,
%  which results in its evaluation being
%  deferred until SHOW time

% and now define the Cube

Cube   :=   Cubeface
        &  Cubeface | XROT (180)  % 180 degrees
        &  Cubeface | YROT ( 90)
        &  Cubeface | YROT (-90)
        &  Cubeface | XROT ( 90)
        &  Cubeface | XROT (-90);
```

```
% In order to have a more pleasant look at
% the picture shown on the screen we magnify
% cube by 5 times.
BigCube := Cube | SCALE 5;

% Set up initial Z Transform for each cube face
%
Tranz    :=   ZMOVE (10);  % 10 units out

%
% GLOBAL!.TRANSFORM has been treated as a global variable.
% GLOBAL!.TRANSFORM should be initialized as a perspective
% transformation matrix so that a viewer can have a correct
% look at the picture as the viewing location changed.
% For instance, it may be set as the desired perspective
% with a perspective window centered at the origin and
% of screen size 60, and the observer at -300 on the z axis.
% Currently this has been set as default perspective transformation.

% Now draw cube
%
SHOW  BigCube;

%

% Draw it again rotated and moved left
%
SHOW  (BigCube | XROT 20 | YROT 30 | ZROT 10);

% Dynamically expand the faces out
%
Tranz    :=   ZMOVE 12;
%
SHOW  (BigCube | YROT 30 | ZROT 10);

% Now show 5 cubes, each moved further right by 80
%
Tranz    :=    ZMOVE 10;
%
SHOW (Cube | SCALE 2.5 | XMOVE (-240) | REPEATED(5, XMOVE 80));
```

```
%
% Now try pointset modifier.
% Given a pointset (polygon) as control points either a BEZIER or a
% BSPLINE curve can be drawn.
%
Cpts := {0,0} _ {70,-60} _ {189,-69} _ {206,33} _ {145,130} _ {48,130}
       _ {0,84} $
%
% Now draw Bezier curve
% Show the polygon and the Bezier curve
%
SHOW (Cpts & Cpts | BEZIER());

% Now draw Bspline curve
% Show the polygon and the Bspline curve
%
SHOW (Cpts & Cpts | BSPLINE());

% Now work on the Circle
% Given a center position and a radius a circle is drawn
%
SHOW ( {10,10} | CIRCLE(50));

%
% Define a procedure which returns a model of
% a Cube when passed the face to be used
%
Symbolic Procedure Buildcube;
 List 'Buildcube;
% put the name onto the property list
Put('buildcube, 'pbintrp, 'Dobuildcube);
Symbolic Procedure Dobuildcube Face$
       Face  &  Face | XROT(180)
             &  Face | YROT(90)
             &  Face | YROT(-90)
             &  Face | XROT(90)
             &  Face | XROT(-90) ;
% just return the value of the one statement
% Use this procedure to display 2 cubes, with and
%  without the Arrow - first do it by calling
%  Buildcube at time the Model is built
%
P := Cubeface | Buildcube() | XMOVE(-15) &
     (Outline | 'Tranz) | Buildcube() | XMOVE 15;
%
SHOW (P | SCALE 5);

% Now define a procedure which returns a Model of
%   a cube when passed the half size parameter

Symbolic Procedure Cubemodel;
 List 'Cubemodel;
%put the name onto the property list
Put('Cubemodel,'Pbintrp, 'Docubemodel);
Symbolic Procedure Docubemodel  HSize;
 << if idp HSize then HSize := eval HSize$
    { HSize,  HSize,  HSize}  _
    {-HSize,  HSize,  HSize}  _
    {-HSize, -HSize,  HSize}  _
    { HSize, -HSize,  HSize}  _
    { HSize,  HSize,  HSize}  _
    { HSize,  HSize, -HSize}  _
    {-HSize,  HSize, -HSize}  _
    {-HSize, -HSize, -HSize}  _
    { HSize, -HSize, -HSize}  _
    { HSize,  HSize, -HSize}  &
    {-HSize,  HSize, -HSize}  _
    {-HSize,  HSize,  HSize}  &
    {-HSize, -HSize, -HSize}  _
    {-HSize, -HSize,  HSize}  &
    { HSize, -HSize, -HSize}  _
    { HSize, -HSize,  HSize} >>;
```

```
% Imbed the parameterized cube in some Models
%
His!.cube :=  'His!.size | Cubemodel();
Her!.cube :=  'Her!.size | Cubemodel();
R  :=  His!.cube | XMOVE (60)  &
      Her!.cube | XMOVE (-60) ;

% Set up some sizes and SHOW them

His!.size := 50;
Her!.size := 30;
%
SHOW   R ;

%
% Set up some different sizes and SHOW them again
%
His!.size := 35;
Her!.size := 60;
%
SHOW R;

%
% Now show a triangle rotated 45 degree about the z axis.
Rotatedtriangle  :=  {0,0} _ {50,50} _
                       {100,0} _ {0,0} | Zrot (45);
%
SHOW Rotatedtriangle;
```

```
%
% Define a procedure which returns a model of a Pyramid
% when passed 4 vertices of a pyramid.
% Procedure Second,Third, Fourth and Fifth are primitive procedures
% written in the source program which return the second, the third,
% the fourth and the fifth element of a list respectively.
% This procedure simply takes 4 points and connects the vertices to
% show a pyramid.
Symbolic Procedure Pyramid (Point4); %.point4 is a pointset
       Point4 &
            Third Point4 _
            Fifth Point4 _
            Second Point4 _
            Fourth Point4 ;

% Now give a pointset indicating 4 vertices build a pyramid
% and show it
%
My!.vertices := {-40,0} _ {20,-40} _ {90,20} _ {70,100};
My!.pyramid := Pyramid Vertices;
%
SHOW ( My!.pyramid | XROT 30);

%
%  A procedure that makes a wheel with "count"
%  spokes rotated around the z axis.
%  in which "count" is the number specified.
Symbolic Procedure Dowheel(spoke,count)$
    begin scalar rotatedangle$
          count := first count$
          rotatedangle := 360.0 / count$
         return (spoke | REPEATED(count, ZROT rotatedangle))
    end$
%
% Now draw a wheel consisting of 8 cubes
%
Cubeonspoke :=  (Outline | ZMOVE 10 | SCALE 2) | buildcube();
Eight!.cubes := Cubeonspoke | XMOVE 50 | WHEEL(8);
%
SHOW Eight!.cubes;
```

```
%
%Draw a cube in which each face consists of just
% a wheel of 8 Outlines
%
Flat!.Spoke := outline | XMOVE 25$
A!.Fancy!.Cube := Flat!.Spoke | WHEEL(8) | ZMOVE 50 | Buildcube()$
%
SHOW A!.Fancy!.Cube;

%
% Redraw the fancy cube, after changing perspective by
% moving the observer farther out along Z axis
%
GLOBAL!.TRANSFORM := WINDOW(-500,60);
%
SHOW A!.Fancy!.Cube;

%
% Note the flexibility resulting from the fact that
% both Buildcube and Wheel simply take or return any
% Model as their argument or value
```

The current version of PictureRLISP runs on HP2648A graphics terminal and TEKTRONIX 4006-1 computer display terminal. The screen of the HP terminal is 720 units long in the X direction, and 360 units high in the Y direction. The coordinate system used in HP terminal places the origin in approximately the center of the screen, and uses a domain of -360 to 360 and a range of -180 to 180. Similarly, the screen of the TEKTRONIX terminal is 1024 units long in the X direction, and 780 units high in the Y direction. The same origin is used but the domain is -512 to 512 in the X direction and the range is -390 to 390 in the Y direction.

Procedures HP!.INIT and TEK!.INIT are used to set the terminals to graphics mode and initiate the lower level procedures on HP and TEKTRONIX terminals respectively. Basically, INIT procedures are written for different terminals depending on their specific characteristics. Using INIT procedures keeps terminal device dependence at the user's level to a minimum.

1.6. DefStruct

Load DEFSTRUCT to use the functions described below, or FAST!-DEFSTRUCT to use those functions but with fast vector operations used. DefStruct is similar to the Spice (Common) Lisp/Lisp machine/MacLisp flavor of struct definitions, and is expected to be subsumed by the Mode package. (Note: the MacLisp version is available in PSL; load NSTRUCT.) It is implemented in PSL[1] as a function which builds access macros and fns

[1]Defstruct was implemented by Russ Fish.

for "typed" vectors, including constructor and alterant macros, a type predicate for the structure type, and individual selector/assignment fns for the elements. DefStruct understands a keyword-option oriented structure specification. DefStruct is now autoloading.

First a few miscellaneous functions on types, before getting into the depths of defining DefStructs:

(DefstructP NAME:id): extra-boolean expr

This is a predicate that returns non-NIL (the Defstruct definition) if NAME is a structured type which has been defined using Defstruct, or NIL if it is not.

(DefstructType S:struct): id expr

This returns the type name field of an instance of a structured type, or NIL if S cannot be a Defstruct type.

(SubTypeP NAME1:id NAME2:id): boolean expr

This returns true if NAME1 is a structured type which has been !:Included in the definition of structured type NAME2, possibly through intermediate structure definitions. (In other words, the selectors of NAME1 can be applied to NAME2.)

Now the function which defines the beasties, in all its gory glory:

(Defstruct NAME-AND-OPTIONS:{id,list} [SLOT-DESCS:{id,list}]): id fexpr

Defines a record-structure data type. A general call to Defstruct looks like this: (in RLisp syntax)

```
(defstruct (struct-name option-1 option-2 ... )
           slot-description-1
           slot-description-2
            ...
          )
```

The name of the defined structure is returned.

Slot-descriptions are:

```
(slot-name default-init slot-option-1 slot-option-2 ... )
```

Struct-name and slot-name are ids. If there are no options following a name in a spec, it can be a bare id with no option argument list. The default-init form is optional and may be omitted. The default-init form is evaluated EACH TIME a structure is to be constructed and the value is used as the initial value of the slot. Options are either a keyword id, or the keyword followed by its argument list. Options are described below.

A call to a constructor macro has the form:

```
(MakeThing (slot-name-1 value-expr-1 )
           (slot-name-2 value-expr-2 )
            ... )
```

The slot-name:value lists override the default-init values which were part of the structure definition. Note that the slot-names look like unary functions of the value, so the parens can be left off. A call to MakeThing with no arguments of course takes all of the default values. The order of evaluation of the default-init forms and the list of assigned values is undefined, so code should not depend upon the ordering.

Implementors Note: Common/LispMachine Lisps define it this way, but Is this necessary? It wouldn't be too tough to make the order be the same as the struct defn, or the argument order in the constructor call. Maybe they think such things should not be advertised and thus constrained in the future. Or perhaps the theory is that constructs such as this can be compiled more efficiently if the ordering is flexible?? Also, should the overridden default-init forms be evaluated or not? I think not.

The alterant macro calls have a similar form:

```
(AlterThing thing
            (slot-name-1 value-expr-1)
            (slot-name-2 value-expr-2)
             ... )
```

The first argument evaluates to the struct to be altered. (The optional parens were left off here.) This is just a multiple-assignment form, which eventually goes through the slot depositors. Remember that the slot-names are used, not the depositor names. (See !:Prefix, below.) The altered structure instance is returned as the value of an Alterant macro.

Implementators note: Common/LispMachine Lisp defines this such that all of the slots are altered in parallel AFTER the new value forms are evaluated, but still with the order of evaluation of the forms undefined. This seemed to lose more than it gained, but arguments for its worth will be entertained.

1.6.1. Options

Structure options appear as an argument list to the struct-name. Many of the options themselves take argument lists, which are sometimes optional. Option ids all start with a colon (!:), on the theory that this distinguishes them from other things.

By default, the names of the constructor, alterant and predicate macros are MakeName, AlterName and NameP. "Name" is the struct-name. The !:Constructor, !:Alterant, and !:Predicate options can be used to override the default names. Their argument is the name to use, and a name of NIL causes the respective macro not to be defined at all.

The !:Creator option causes a different form of constructor to be defined, in addition to the regular "Make" constructor (which can be suppressed.) As in the !:Constructor option above, an argument supplies the name of the macro, but the default name in this case is CreateName. A call to a Creator macro has the form:

```
    (CreateThing slot-value-1 slot-value-2 ... )
```

All of the slot-values of the structure must be present, in the order they appear in the structure definition. No checking is done, other than assuring that the number of values is the same as the number of slots. For obvious reasons, constructors of this form are not recommended for structures with many fields, or which may be expanded or modified.

Slot selector macros may appear on either the left side or the right side of an assignment. They are by default named the same as the slot-names, but can be given a common prefix by the !:Prefix option. If !:Prefix does not have an argument, the structure name is the prefix. If there is an argument, it should be a string or an id whose print name is the prefix.

The !:Include option allows building a new structure definition as an extension of an old one. The required argument is the name of a previously defined structure type. The access functions for the slots of the source type also works on instances of the new type. This can be used to build hierarchies of types. The source types contain generic information in common to the more specific subtypes which !:Include them.

The !:IncludeInit option takes an argument list of "slot-name(default-init)" pairs, like slot-descriptors without slot-options, and files them away to modify the default-init values for fields inherited as part of the !:Included structure type.

1.6.2. Slot Options

Slot-options include the !:Type option, which has an argument declaring the type of the slot as a type id or list of permissible type ids. This is not enforced now, but anticipates the Mode system structures.

The !:UserGet and !:UserPut slot-options allow overriding the simple vector reference and assignment semantics of the generated selector macros with user-defined functions. The !:UserGet FNAME is a combination of the slot-name and a !:Prefix if applicable. The !:UserPut FNAME is the same, with "Put" prefixed. One application of this capability is building depositors which handle the incremental maintenance of parallel data structures as a side effect, such as automatically maintaining display file representations of objects which are resident in a remote display processor in parallel with modifications to the Lisp structures which describe the objects. The Make and Create macros bypass the depositors, while Alter uses them.

1.6.3. A Simple Example

(Input lines have a "> " prompt at the beginning.)

```
> % This example is in Rlisp syntax
> % (Do definitions twice to see what functions were defined.)
> macro procedure TWICE u; list( 'PROGN, second u, second u );
TWICE

> % A definition of Complex, structure with Real and Imaginary parts.
> % Redefine to see what functions were defined.  Give 0 Init values.
> TWICE
> Defstruct( Complex( !:Creator(Complex) ), R(0), I(0) );
*** Function 'MAKECOMPLEX' has been redefined
*** Function 'ALTERCOMPLEX' has been redefined
*** Function 'COMPLEXP' has been redefined
*** Function 'COMPLEX' has been redefined
*** Function 'R' has been redefined
*** Function 'PUTR' has been redefined
*** Function 'I' has been redefined
*** Function 'PUTI' has been redefined
*** Defstruct 'COMPLEX' has been redefined
COMPLEX
```

```
> C0 := MakeComplex();    % Constructor with default inits.
[COMPLEX 0 0]

> ComplexP C0;% Predicate.
T

> C1:=MakeComplex( R 1, I 2 );   % Constructor with named values.
[COMPLEX 1 2]

> R(C1); I(C1);% Named selectors.
1
2

> C2:=Complex(3,4) % Creator with positional values.
[COMPLEX 3 4]

> AlterComplex( C1, R(2), I(3) );     % Alterant with named values.
[COMPLEX 2 3]

> C1;
[COMPLEX 2 3]

> R(C1):=5; I(C1):=6; % Named depositors.
5
6

> C1;
[COMPLEX 5 6]

> % Show use of Include Option.  (Again, redef to show fns defined.)
> TWICE
> Defstruct( MoreComplex( !:Include(Complex) ), Z(99) );
*** Function 'MAKEMORECOMPLEX' has been redefined
*** Function 'ALTERMORECOMPLEX' has been redefined
*** Function 'MORECOMPLEXP' has been redefined
*** Function 'Z' has been redefined
*** Function 'PUTZ' has been redefined
*** Defstruct 'MORECOMPLEX' has been redefined
MORECOMPLEX
```

```
> MO := MakeMoreComplex();
[MORECOMPLEX 0 0 99]

> M1 := MakeMoreComplex( R 1, I 2, Z 3 );
[MORECOMPLEX 1 2 3]

> R C1;
5

> R M1;
1

> % A more complicated example: The structures which are used in the
> % Defstruct facility to represent defstructs.  (The EX prefix has
> % been added to the names to protect the innocent...)
> TWICE% Redef to show fns generated.
> Defstruct(
>     EXDefstructDescriptor( !:Prefix(EXDsDesc), !:Creator ),
>DsSize(!:Type int ),    % (Upper Bound of vector.)
>Prefix(!:Type string ),
>SlotAlist(   !:Type alist ), % (Cdrs are SlotDescriptors.)
>ConsName(    !:Type fnId ),
>AltrName(    !:Type fnId ),
>PredName(    !:Type fnId ),
>CreateName(  !:Type fnId ),
>Include(     !:Type typeid ),
>InclInit(    !:Type alist )
> );
```

```
*** Function 'MAKEEXDEFSTRUCTDESCRIPTOR' has been redefined
*** Function 'ALTEREXDEFSTRUCTDESCRIPTOR' has been redefined
*** Function 'EXDEFSTRUCTDESCRIPTORP' has been redefined
*** Function 'CREATEEXDEFSTRUCTDESCRIPTOR' has been redefined
*** Function 'EXDSDESCDSSIZE' has been redefined
*** Function 'PUTEXDSDESCDSSIZE' has been redefined
*** Function 'EXDSDESCPREFIX' has been redefined
*** Function 'PUTEXDSDESCPREFIX' has been redefined
*** Function 'EXDSDESCSLOTALIST' has been redefined
*** Function 'PUTEXDSDESCSLOTALIST' has been redefined
*** Function 'EXDSDESCCONSNAME' has been redefined
*** Function 'PUTEXDSDESCCONSNAME' has been redefined
*** Function 'EXDSDESCALTRNAME' has been redefined
*** Function 'PUTEXDSDESCALTRNAME' has been redefined
*** Function 'EXDSDESCPREDNAME' has been redefined
*** Function 'PUTEXDSDESCPREDNAME' has been redefined
*** Function 'EXDSDESCCREATENAME' has been redefined
*** Function 'PUTEXDSDESCCREATENAME' has been redefined
*** Function 'EXDSDESCINCLUDE' has been redefined
*** Function 'PUTEXDSDESCINCLUDE' has been redefined
*** Function 'EXDSDESCINCLINIT' has been redefined
*** Function 'PUTEXDSDESCINCLINIT' has been redefined
*** Defstruct 'EXDEFSTRUCTDESCRIPTOR' has been redefined
EXDEFSTRUCTDESCRIPTOR

> TWICE% Redef to show fns generated.
> Defstruct(
>     EXSlotDescriptor( !:Prefix(EXSlotDesc), !:Creator ),
>SlotNum(     !:Type int ),
>InitForm(    !:Type form ),
>SlotFn(!:Type fnId ), % Selector/Depositor id.
>SlotType(    !:Type type ), % Hm...
>UserGet(     !:Type boolean ),
>UserPut(     !:Type boolean )
> );
```

```
*** Function 'MAKEEXSLOTDESCRIPTOR' has been redefined
*** Function 'ALTEREXSLOTDESCRIPTOR' has been redefined
*** Function 'EXSLOTDESCRIPTORP' has been redefined
*** Function 'CREATEEXSLOTDESCRIPTOR' has been redefined
*** Function 'EXSLOTDESCSLOTNUM' has been redefined
*** Function 'PUTEXSLOTDESCSLOTNUM' has been redefined
*** Function 'EXSLOTDESCINITFORM' has been redefined
*** Function 'PUTEXSLOTDESCINITFORM' has been redefined
*** Function 'EXSLOTDESCSLOTFN' has been redefined
*** Function 'PUTEXSLOTDESCSLOTFN' has been redefined
*** Function 'EXSLOTDESCSLOTTYPE' has been redefined
*** Function 'PUTEXSLOTDESCSLOTTYPE' has been redefined
*** Function 'EXSLOTDESCUSERGET' has been redefined
*** Function 'PUTEXSLOTDESCUSERGET' has been redefined
*** Function 'EXSLOTDESCUSERPUT' has been redefined
*** Function 'PUTEXSLOTDESCUSERPUT' has been redefined
*** Defstruct 'EXSLOTDESCRIPTOR' has been redefined
EXSLOTDESCRIPTOR


> END;
NIL
```

1.7. Bignums

1.7.1. BigNum Structure and "Constants"

Load BIG to get the bignum package.[2] The current PSL bignum package was written using vectors of "Big Digits" or "Bigits". The first element of each vector is either BIGPOS or BIGNEG, depending whether the number is positive or negative. A bignum of the form

[BIGPOS a b c d]

has a value of

a + b * bbase!* + c * bbase!* ** 2 + d * bbase!* ** 3

BBase!* is a fluid variable which varies from one machine to another. For the VAX and the DEC-20, it is calculated as follows:

bbits!* := (n-1)/2; bbase!* := 2 ** bbits!*;

"n" is the total number of bits per word on the given machine. On the DEC-20, n is 36,

[2]This section is adapted from a help file and was written by Beryl Morrison.

so bbits!* is 17 and bbase!* is 131072. On the VAX, n is 32, so bbits!* is 15 and bbase!* is 32768.

1.7.2. The Functions in BigBig

The functions defined by BigBig for bignums are as follows:

BLOr	Takes two BigNum arguments, returning a bignum. Calls BSize, GtPos, PosIfZero.
BLXOr	Takes two BigNum arguments, returning a bignum. Calls BSize, GtPos, TrimBigNum1.
BLAnd	Takes two BigNum arguments, returning a bignum. Calls BSize, GtPos, TrimBigNum1.
BLNot	Takes one BigNum argument, returning a bignum. Calls BMinus, BSmallAdd.
BLShift	Takes two BigNum arguments, returning a bignum. Calls BMinusP, BQuotient, BTwoPower, BMinus, BTimes2.
BMinus	Takes one BigNum argument, returning a bignum. Calls BZeroP, BSize, BMinusP, GtPos, GtNeg.
BMinusP	Takes one BigNum argument, returning a bignum or NIL.
BPlus2	Takes two BigNum arguments, returning a bignum. Calls BMinusP, BDifference2, BMinus, BPlusA2.
BDifference	BZeroP, BMinus, BMinusP, BPlusA2, BDifference2.
BTimes2	Takes two BigNum arguments, returning a bignum. Calls BSize, BMinusP, GtPos, GtNeg, BDigitTimes2, PosIfZero, TrimBigNum1.
BDivide	Takes two BigNum arguments, returning a pair of bignums. Calls BSize, GtPos, BSimpleDivide, BHardDivide.
BGreaterP	Takes two BigNum arguments, returning a bignum or NIL. Calls BMinusP, BDifference.
BLessP	Takes two BigNum arguments, returning a bignum or NIL. Calls BMinusP, BDifference.
BAdd1	Takes a BigNum argument, returning a bignum. Calls BSmallAdd.
BSub1	Takes a BigNum argument, returning a bignum. Calls BigSmallDiff.
FloatFromBigNum	Takes a bignum, returning a float. Calls BZeroP, BGreaterP, BLessP, BSize, BMinusP.

BChannelPrin2
: Calls BigNumP, NonBigNumError, BSimpleDivide, BSize, BZeroP.

BRead
: Calls GtPos, BReadAdd, BMinus.

BigFromFloat
: Takes a float and converts to a bignum. Calls BNum, BPlus2, BTimes2, BTwoPower, FloatFromBigNum, BMinus, PosIfZero.

The following functions are support functions for those given above.

SetBits
: Takes as an argument the total number of bits per word on a given machine; sets some fluid variables accordingly. NOTE: FloatHi!* must be changed separately from this procedure by hand when moving to a new machine both in bigbig.red and in bigface.red. Calls TwoPower, BNum, BMinus, BSub1, BTwoPower, BAdd1.

BigNumP
: Checks if the argument is a bignum. Calls no special functions.

NonBigNumError
: Calls no special functions.

BSize
: Gives size of a bignum, i.e. total number of bigits (the tag "BIGPOS" or "BIGNEG" is number 0). Calls BigNumP.

PosIfZero
: Takes a bignum; if it is a negative zero, it is converted to a positive zero. Calls BPosOrNegZeroP, BMinusP.

BPosOrNegZeroP
: Takes a BigNum; checks if magnitude is zero. Calls BSize.

GtPos
: Takes an inum/fixnum. Returns a vector of size of the argument; first (i.e.0th) element is BIGPOS, others are NIL.

GtNeg
: Takes an inum/fixnum. Returns a vector of size of the argument; first (i.e.0th) element is BIGNEG, others are NIL.

TrimBigNum
: Takes a BigNum as an argument; truncates any trailing "NIL"s. Calls BigNumP, NonBigNumError, TrimBigNum1, BSize.

TrimBigNum1
: Does dirty work for TrimBigNum, with second argument the size of the BigNum.

Big2Sys
: Calls BLessP, BGreaterP, BSize, BMinusP.

TwoPower
: Takes and returns a fix/inum. 2**n.

BTwoPower
: Takes a fix/inum or bignum, returns a bignum of value 2**n. Calls BigNumP, Big2Sys, GtPos, TwoPower, TrimBigNum1.

BZeroP	Checks size of BigNum (0) and sign. Calls BSize, BMinusP.
BOneP	Calls BMinusP, BSize.
BAbs	Calls BMinusP, BMinus.
BGeq	Calls BLessP.
BLeq	Calls BGreaterP.
BMax	Calls BGeq.
BMin	Calls BLeq.
BExpt	Takes a BigNum and a fix/inum. Calls Int2B, BTimes2, BQuotient.
AddCarry	Support for trapping the carry in addition.
BPlusA2	Does the dirty work of addition of two BigNums with signs pre-checked and identical. Calls BSize, GtNeg, GtPos, AddCarry, PosIfZero, TrimBigNum1.
SubCarry	Mechanism to get carry in subtractions.
BDifference2	Does the dirty work of subtraction with signs pre-checked and identical. Calls BSize, GtNeg, GtPos, SubCarry, PosIfZero, TrimBigNum1.
BDigitTimes2	Multiplies the first argument (BigNum) by a single Bigit of the second BigNum argument. Returns the partially completed result. Calls no special functions.
BSmallTimes2	Takes a BigNum argument and a fixnum argument, returning a bignum. Calls GtPos, BMinusP, GtNeg, PosIfZero, TrimBigNum1.
BQuotient	Takes two BigNum arguments, returning a bignum. Calls BDivide.
BRemainder	Takes two BigNum arguments, returning a bignum. Calls BDivide.
BSimpleQuotient	Calls BSimpleDivide.
BSimpleRemainder	Calls BSimpleDivide.
BSimpleDivide	Used to divide a BigNum by an inum. Returns a dotted pair of quotient and remainder, both being bignums. Calls BMinusP, GtPos, GtNeg, PosIfZero, TrimBigNum1.
BHardDivide	

	Used to divide two "true" BigNums. Returns a pair of bignums. Algorithm taken from Knuth. Calls BMinusP, GtPos, GtNeg, BAbs, BSmallTimes2, BSize, BDifference, BPlus2, TrimBigNum1, BSimpleQuotient, PosIfZero.
BReadAdd	Calls BSmallTimes2, BSmallAdd.
BSmallAdd	Adds an inum to a BigNum, returning a bignum. Calls BZeroP, BMinusP, BMinus, BSmallDiff, BSize, GtPos, AddCarry, PosIfZero, TrimBigNum1.
BNum	Takes an inum and returns a BigNum of one bigit; test that the inum is less than bbase!* is assumed done. Calls GtPos, GtNeg.
BSmallDiff	Calls BZeroP, BMinusP, BMinus, BSmallAdd, GtPos, SubCarry, PosIfZero, TrimBigNum1.
Int2B	Takes a fix/inum and converts to a BigNum. Calls BNum, BRead.

CHAPTER 2
MISCELLANEOUS UTILITIES

2.1. Introduction

This chapter describes an assortment of utility packages. It also provides a list of many of the undocumented packages that reside on the utility directory with some indication of their purposes.

2.2. 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.

2.3. DefConst

(DefConst [U:id V:number]): Undefined macro

DefConst is a simple means for defining and using symbolic constants, as an alternative to the heavy-handed NEWNAM or DEFINE facility in Reduce/RLisp. Constants are defined thus:

```
(DefConst FooSize 3)
```

or as sequential pairs:

```
(DEFCONST FOOSIZE 3
          BARSIZE 4)
```

(Const U:id): number macro

They are referred to by the macro Const, so

```
(CONST FOOSIZE)
```

would be replaced by 3.

2.4. Hashing Cons

HCONS is a loadable module. The HCons function creates unique dotted pairs. In other words, (HCons A B) Eq (HCons C D) if and only if A Eq C and B Eq D. This allows rapid tests for equality between structures, at the cost of expending more time in creating the structures. The use of HCons may also save space in cases where lists share common substructure, since only one copy of the substructure is stored.

Hcons works by keeping a pair hash table of all pairs that have been created by HCons. (So the space advantage of sharing substructure may be offset by the space consumed by table entries.) This hash table also allows the system to store property lists for pairs--in the same way that Lisp has property lists for identifiers.

Pairs created by HCons should not be modified with RplacA and RplacD. Doing so will make the pair hash table inconsistent, as well as being very likely to modify structure shared with something that you don't wish to change. Also note that large numbers may be equal without being eq, so the HCons of two large numbers may not be Eq to the HCons of two other numbers that appear to be the same. (Similar warnings hold for strings and vectors.)

The following "user" functions are provided by HCONS:

(HCons [U:any]): pair macro

The HCons macro takes one or more arguments and returns their "hashed cons" (right associatively). With two arguments this corresponds to a call of Cons.

(HList [U:any]): list nexpr

HList is the "HCONS version" of the List function.

(HCopy U:any): any macro

HCopy is the HCONS version of the Copy function. Note that HCopy serves a very different purpose than Copy, which is usually used to copy a structure so that destructive changes can be made to the copy without changing the original. HCopy only copies those parts of the structure which haven't already been "Consed together" by HCons.

(HAppend U:list V:list): list expr

The HCons version of Append.

(HReverse U:list): list expr

The HCons version of Reverse.

The following two functions can be used to "Get" and "Put" properties for pairs or identifiers. The pairs for these functions must be created by HCons. These functions are known to the SetF macro.

(Extended-Put U:{id,pair} IND:id PROP:any): any expr

(Extended-Get U:{id,pair} IND:any): any expr

2.5. Graph-to-Tree

GRAPH-TO-TREE is a loadable module. PrintX obtained by loading DEBUG also prints circular lists.

(Graph-to-Tree A:form): form expr

The function Graph-to-Tree copies an arbitrary s-expression, removing circularity. It does NOT show non-circular shared structure. Places where a substructure is Eq to one of its ancestors are replaced by non-interned ids of the form <n> where n is a small integer. The parent is replaced by a two element list of the form (<n>: u) where the n's match, and u is the (de-circularized) structure. This is most useful in adapting any printer for use with circular structures.

(CPrint A:any): NIL expr

The function CPrint, also defined in the module GRAPH-TO-TREE, is simply (PrettyPrint (Graph-to-Tree X)).

Note that GRAPH-TO-TREE is very embryonic. It is MUCH more inefficient than it needs to be, heavily consing. A better implementation would use a stack (vector) instead of lists to hold intermediate expressions for comparison, and would not copy non-circular structure. In addition facilities should be added for optionally showing shared structure, for performing the inverse operation, and for also editing long or deep structures. Finally, the output representation was chosen at random and can probably be improved, or at least brought in line with CL or some other standard.

2.6. Inspect Utility

INSPECT is a loadable module. Currently INSPECT does not work in Lisp syntax.

(Inspect FILENAME:string): expr

This is a simple utility which scans the contents of a source file to tell what functions are defined in it. It will be embellished slightly to permit the on-line querying of certain attributes of files. Inspect reads one or more files, printing and collecting information on defined functions.

Usage:

```
(LOAD INSPECT)
(INSPECT "file-name") % Scans the file, and prints proc
                      % names.  It also
                      % builds the lists ProcedureList!*
                      % FileList!* and ProcFileList!*

                      % File-Name can DSKIN other files
```

On the Fly printing is controlled by !*PrintInspect, default is T. Other lists built include FileList!* and ProcFileList!*, which is a list of (procedure . filename) for multi-file processing.

For more complete process, do:

```
(LOAD INSPECT)
(OFF PRINTINSPECT)
(INSPECTOUT)
(DSKIN ...)
(DSKIN ...)
(INSPECTEND)
```

2.7. If_System

(If_System): any macro

This is a compile-time conditional macro for system-dependent code. FALSE-CASE can be omitted and defaults to NIL. SYS-NAME must be a member of the fluid variable System_List!*. For the Dec-20, System_List!* is (Dec20 PDP10 Tops20 KL10). For the VAX it is (VAX Unix VMUnix). Load IF_SYSTEM to use this macro. An example of its use follows.

```
(de mail ()
   (if_system tops20 (runfork "SYS:MM.EXE")
         (if_system unix (system "/bin/mail")
                 (stderror "Mail command not implemented"))))
```

2.8. Profiler for Compiled Functions

Load PROFILE to get a module that allows one to determine run times for compiled PSL functions.[1] This version does not yet try to account for overhead of PROFILE itself.

```
USAGE: After loading PROFILE module:

PRINT!-PROFILE();       will display default table, sorted alphabetically
                        This takes :KEYWORD options, any pair can
                        be omitted
e.g. PRINT!-PROFILE(!:MINCALLS,1,      % only show if called at least once
                   !:MINTIME,10,     % only show if time > 10 ms
                   !:NAMES,'(FOO FEE FUM));
PROFILE fnlist;         Explicitly profiles functions in fnlist
UNPROFILE fnlist;       Explicitly unprofiles functions in fnlist
UNPROFILE '!:ALL;       Explicitly unprofiles ALL profiled functions
CLEAR!-PROFILE fnlist;Resets all counters for functions on FNLIST
CLEAR!-PROFILE '!:ALL; Resets ALL
ON PROFILE;             will cause all new PUTD's to have PROFILE code added.
```

[1] Based on B. Hulshof's original version at RAND ~May 1983 Rewritten and optimized by M. Griss

```
OFF PROFILE;             will stop this redefinition, but will NOT change
                         already profiled code
```

A log of Profile's use on the Dec20 follows:

```
RLisp
[Keeping rlisp]
Extended 20-PSL 3.1 RLisp, 15-Jun-83
[1] load "pnew:profile";
NIL
[2] on profile;
NIL
[3] in "<griss>profile-fns.red";
% Profile-fns.red

procedure top N;
 <<subcall N; subcall N>>;
*** (TOP): base 1324652, length 10 words
TOP


procedure subcall N;
  for i:=1:n do fac1 N;
*** (SUBCALL): base 1324672, length 10 words
SUBCALL


procedure fac1 n;
 if n<=1 then 1 else n*fac2(n-1);
*** (FAC1): base 1324711, length 10 words
FAC1


procedure fac2 n;
 if n<=1 then 1 else n*fac1(n-1);
*** (FAC2): base 1324726, length 10 words
FAC2


End;
NIL
[4] off profile;
NIL
[5] on time;
NIL
TIME: 1101 MS
[6] top 20;
NIL
TIME: 1662 MS
[7] print!-profile();
```

```
function                calls           time (ms)       tree-time (ms)
FAC1                    400             793             1509
FAC2                    400             716             1432
SUBCALL                 2               90              1599
TOP                     1               2               1601
Total (4 fns):          803             1601
NIL
TIME: 384 MS
```

2.9. Timing Function Calls

Load TIME-FNC to get code to time function calls.

Usage:

```
        do
        (timef function-name-1 function-name-2 ...)

        Timef is a fexpr.
        It will redefine the functions named so that timing information is
        kept on these functions.
        This information is kept on the property list of the function name.
        The properties used are 'time' and 'number-of-calls'.

        (get function-name 'time) gives you the total time in the function.
        (not counting gc time).
        Note, this is the time from entrance to exit.
        The timef function redefines the function with an
        unwind-protect, so calls that are interrupted
        by *throws are counted.

        (get function-name 'number-of-calls) gives you the number of times
        the function is called.

        To stop timing do :
        (untimef function-name1 ..)
        or do (untimef) for all functions.
        (untimef) is a fexpr.

        To print timing information do
        (print-time-info function-name-1 function-name-2 ..)

        or do (print-time-info) for timing information on all function names.

        special variables used:
        *timed-functions* : list of all functions currently being timed.
```

```
*all-timed-functions* : list of all functions ever timed in the
        current session.

Comment: if tr is called on a called on a function that is already
being timed, and then untimef is called on the function, the
function will no longer be traced.
```

2.10. Parenthesis Checker

PCHECK will read a Lisp syntax (.SL) file, printing some of the top-level of each S-expression. It is meant to survey the file, and if the file has unbalanced parensthesis, will show where things get confused.

To use:

```
(LOAD PCHECK)
(PCHECK "foo.sl")
```

2.11. A Simple Rational Function Evaluator

POLY is a simple (pedagogic) Rational Function Evaluator.

After loading POLY, run function ALGG(); or RAT(); These accept a sequence of expressions:

```
<exp> ; | QUIT; (Semicolon terminator)
<exp> ::= <term> [+ <exp>  | - <exp>]
<term> ::= <primary> [* <term> | / <term>]
<primary> ::= <primary0> [^ <primary0> | ' <primary0> ]
        ^ is exponentiation, ' is derivative
<primary0> ::= <number> | <variable> | ( <exp> )
```

It includes a simple parser (RPARSE), 2 evaluators (RSIMP x) and (PRESIMP), and 2 prettyprinters, (RATPRINT) and (PREPRINT)

```
PREFIX Format: <number> | <id> | (op arg1 arg2)
               + -> PLUS2
               - -> DIFFERENCE (or MINUS)
               * -> TIMES2
               / -> QUOTIENT
               ^ -> EXPT
               ' -> DIFF

Canonical Formats: Polynomial: integer | (term . polynomial)
                   term      : (power . polynomial)
                   power     : (variable . integer)
                   Rational  : (polynomial .  polynomial)
```

2.12. Undocumented Utilities

This section lists most of the utilities available in PSL that are not documented. Consult the sources in directory pu: on the DEC-20, $pu on the VAX, or the utility directory on your system. These modules can be loaded.

ADDR2ID	Converts a code pointer to a symbol (function name)
ASSOCIATION	Mutable association lists
CLCOMP1	Incompatible Common Lisp compatibility
COMMON	Compile-time and read-time support for Common Lisp compatibility
EVALHOOK	Support for special evaluation
FAST-ARITH	Speed up generic arithmetic
FAST-EVECTORS	Fast-compiled evector operations
SLOW-STRINGS	Defines some string operations
FAST-STRINGS	Fast versions of the functions in SLOW-STRINGS
STRINGX	Some useful string functions
STRING-INPUT	Input from strings
STRING-SEARCH	General purpose searches for substrings
UN-RLISP	Translates a program written in RLisp syntax into Lisp syntax
UTIL	General utility/support functions
EXTENDED-CHAR	Nine-bit terminal input characters
HASH	Hash table package
HEAP-STATS	Part of heap statistics gathering package
H-STATS-1	Part of heap statistics gathering package
PARSE-COMMAND-STRING	

Parse command string given at invocation of PSL

PROGRAM-COMMAND-INTERPRETER
Redefine startup routine to read command given at invocation of PSL

PATHNAMEX
Useful functions involving pathnames

PSL-INPUT-STREAM
File input stream objects

PSL-OUTPUT-STREAM
File output stream objects

CHAPTER 3
THE OBJECTS MODULE

3.1. Introduction

[1] 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 goes over the basics of using flavored objects once briefly to give you an idea of what is involved, then goes into details.

A datatype is known as a flavor (don't ask). The definition of a flavor can be thought of in two parts: the DEFFLAVOR form ("flavor definition"), plus a set of DEFMETHOD forms ("method definitions") for operating on objects of that flavor.

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, all operations on flavored objects are automatically "generic" operations. This means that any programs you write that USE flavored objects have an extra degree of built-in generality.

What does it mean to say that operations on flavored objects are generic? This means that the operations can be done on an object of any flavor, just so 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

[1] This chapter is adapted from the file ph:objects.doc which was written by Cris Perdue and Alan Snyder

the object it is being done to.

We may 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 be done 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.

3.1.1. Defflavor

A flavor definition looks like:

```
(defflavor flavor-name (var1 var2 ...) () option1 option2 ...)
```

Example:

```
(defflavor complex-number
  (real-part
   (imaginary-part 0.0))
  ()
  gettable-instance-variables
  initable-instance-variables
 )
```

A flavor definition specifies the fields, components, or in our terminology, the "instance variables" that each object of that flavor is to have. 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 the 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". None, some of, or all of the instance variables may be specified in each option.

3.1.2. 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.

3.1.3. 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. (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 objects of its flavor. The flavor definition itself also defines some methods. For each "gettable" instance variable, a method of the same name is defined which returns the current value of that instance variable. For "settable" instance variables a method named "set-<variable name>" is defined. Given a new value for the instance variable, the method sets the instance variable to have that value.

3.1.4. Sanctity of Objects

Most Lisps and PSL in particular leave open the possibility for the user to perform illicit operations on Lisp objects. Objects defined by the objects package 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.

3.2. Reference Information

3.2.1. Loading the Module

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.

3.2.2. Defflavor

The macro DEFFLAVOR is used to define a new flavor of object.

```
(defflavor name:id instance-variables:list
    mixin-flavors:NIL [option:form]): id-list                    macro
```

Examples:

```
(defflavor complex-number (real-part imaginary-part) ()
   gettable-instance-variables
   initable-instance-variables
   )

(defflavor complex-number ((real-part 0.0)
                           (imaginary-part 0.0)
                           )
   ()
   gettable-instance-variables
   (settable-instance-variables real-part)
   )
```

The INSTANCE-VARIABLES form a list. Each member of the list is either a symbol (id) or a list of 2 elements. The 2-element list form consists of a symbol and a default initialization form.

Note: 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 about which occurrences of the symbol for an instance variable are actually uses of the variable, though it does understand the nature of QUOTE.

The MIXIN-FLAVORS list must be empty. In the Lisp Machine flavors facility, this may be a list of names of other flavors.

Recognized options are:

```
(GETTABLE-INSTANCE-VARIABLES var1 var2 ...)
(SETTABLE-INSTANCE-VARIABLES var1 var2 ...)
(INITABLE-INSTANCE-VARIABLES var1 var2 ...)

GETTABLE-INSTANCE-VARIABLES  [make all instance variables GETTABLE]
SETTABLE-INSTANCE-VARIABLES  [make all instance variables SETTABLE]
INITABLE-INSTANCE-VARIABLES  [make all instance variables INITABLE]
```

An empty list of variables is taken as meaning all variables rather than none, so (GETTABLE-INSTANCE-VARIABLES) is equivalent to GETTABLE-INSTANCE-VARIABLES.

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.

Initable instance variables may be initialized via options to MAKE-INSTANCE or INSTANTIATE-FLAVOR. See below.

3.2.3. Defmethod

The macro DEFMETHOD is used to define a method on an existing flavor.

(defmethod name-list:id-list [arg:id [expr:form]]): id-list macro

NAME-LIST is a two element id-list giving the flavor name on which the method is to be used and the name of the method being defined. Each ARG is an identifier. There may be zero or more ARGs.

Examples:

```
(defmethod (complex-number real-part) ()
  real-part)

(defmethod (complex-number set-real-part) (new-real-part)
  (setf real-part new-real-part))
```

The body of a method can refer to any instance variable of the flavor by using the name just like an ordinary variable. They can set them using SETF. All occurrences of instance variables (except within evectors or quoted lists) are translated to an invocation of the form (IGETV SELF n).

The body of a method can also freely use SELF much as though it were another instance variable. SELF is bound to the object that the method applies to. SELF may not be setq'ed or setf'ed.

Example using SELF:

```
(defmethod (toaster plug-into) (socket)
  (setf plugged-into socket)
  (=> socket assert-as-plugged-in self))
```

3.2.4. Creating New Instances of Flavors

There are two ways to create a new instance of a flavor: use Make-Instance or Instantiate-Flavor.

(make-instance flavor-name:id [instance-var:id init-val:any]): object macro

MAKE-INSTANCE takes as arguments a flavor name and an optional sequence of initializations, consisting of alternating pairs of instance variable names and corresponding initial values. Note that all the arguments are evaluated. It returns an object of the specified flavor.

Examples:

```
(setq x (make-instance 'complex-number))
(setq y (make-instance 'complex-number 'real-part 0.0
                                       'imaginary-part 1.0))
```

Initialization of a newly made object happens as follows:

Each instance variable with initialization specified in the call to make-instance is initialized to the value given. Any instance variables not initialized in this way, but having default initializations specified in the flavor definition are initialized by the default initialization specified there. All other instance variables are initialized to the symbol *UNBOUND*.

If a method named INIT is defined for this flavor of object, that method 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.

(Instantiate-Flavor flavor-name:id init-list:list): object <u>expr</u>

This is the same as Make-Instance, except that the initialization list is provided as a single (required) argument.

Example:

```
(instantiate-flavor 'complex-number
                    (list 'real-part (sin 30) 'imaginary-part (cos 30)))
```

3.3. Operating on Objects

Operations on an object are done by the methods of the flavor of the object. We say that a method is invoked, or we may say that a message is sent to the object. The notation suggests the sending of messages. In this metaphor, the name of the method to use is part of the message sent to the object, and the arguments of the method are the rest of the message. There are several approaches to invoking a method:

* => A convenient form for sending a message. 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 Send a message in which the message is evaluated. 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 Conditionally Send a Message (Evaluated Message Name) 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 Send a Message (Explicit "Rest" Argument List) Examples:

```
(lexpr-send foo 'bar a b c list)
```

The last argument to LEXPR-SEND is a list of the remaining arguments.

* LEXPR-SEND-IF-HANDLES This is the same as LEXPR-SEND, except that no error is reported if the object fails to handle the message.

* LEXPR-SEND-1 - 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 This is the same as LEXPR-SEND-1, except that no error is reported if the object fails to handle the message.

* EV-SEND - EXPR form of LEXPR-SEND-1 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.

3.4. Useful Functions on Objects

(Object-Type object:id): id,NIL <u>expr</u>

The Object-Type function returns the type (an <u>id</u>) 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.

3.5. 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 nested deeply 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.

3.6. Declare-Flavor and Undeclare-Flavor

*** 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.

Here are the syntactic details:

(DECLARE-FLAVOR FLAVOR-NAME VAR1 VAR2 ...) (UNDECLARE-FLAVOR VAR1 VAR2 ...)

*** Did you read the above warnings??? ***

3.7. Representation Information

(You don't need to know any of this to use this stuff.)

A flavor-name is an ID. It has the following properties:

VARIABLE-NAMES
: A list of the instance variables of the flavor, in order of their location in the instance evector. This property exists at compile time, dskin time, and load time.

INITABLE-VARIABLES
: A list of the instance variables that have been declared to be INITABLE. This property exists at dskin time and at load time.

METHOD-TABLE
: An association list mapping each method name (ID) defined for the flavor to the corresponding function name (ID) that implements the method. This property exists at dskin time and at load time.

INSTANCE-VECTOR-SIZE
: An integer that specifies the number of elements in the evector that represents an instance of this flavor. This property exists at dskin time and at load time. It is used by MAKE-INSTANCE.

The function that implements a method has a name of the form FLAVOR$METHOD. Each such function ID has the following properties:

SOURCE-CODE
: A list of the form (LAMBDA (SELF ...) ...) which is the untransformed source code for the method. This property exists at compile time and dskin time.

Implementation Note:

A tricky aspect of the code that implements the objects package is making sure that the right things happen at the right time. When a source file is read and evaluated (using DSKIN), then everything must happen at once. However, when a source file is compiled to produce a FASL file, then some actions must be performed at compile-time, whereas other actions are supposed to occur when the FASL file is loaded. Actions to occur at compile time are performed by macros; actions to occur at load time are performed by the forms returned by macros.

Another goal of the implementation is to avoid consing whenever possible during method invocation. The current scheme prefers to compile into (APPLY HANDLER (LIST args...)), for which the PSL compiler will produce code that performs no consing.

CHAPTER 4
EDITORS

4.1. A Mini Structure-Editor

PSL and RLisp provide a fairly simple structure editor, essentially a subset of the structure editor described below in section 4.3. This mini editor is usually resident in PSL and RLisp, or can be LOADed. It is useful for correcting errors in input, often via the E option in the BREAK loop. Do HELP(EDITOR) for more information.

To edit an expression, call the function Edit with the expression as an argument. The edited copy is returned. To edit the definition of a function, call EditF with the function name as an argument.

In the editor, the following commands are available (N indicates a non-negative integer):

P edit

Prints the subexpression under consideration. On entry, this is the entire expression. This only prints down PLevel levels, replacing all edited subexpressions by ***. Plevel is initially 3.

PL (N) edit

Changes PLEVEL to N.

N:integer edit-command

Sets the subexpression under consideration to be the nth subexpression of the current one. That is, walk down to the nth subexpression.

-N:integer edit-command

Sets the current subexpression to be the nth Cdr of the current one.

UP edit

Go to the subexpression you were in just before this one.

T edit

Go to the top of the original expression.

F (S) edit

Find the first occurrence of the S-expression S. The test is performed by Equal, not Eq. The current level is set to the first level in which S was found.

(N:integer) edit-command

Delete the Nth element of the current expression.

(N:integer [ARG]) edit-command

Replace the Nth element by ARGs.

(-N:integer [ARG]) edit-command

Insert the elements ARGs before the nth element.

(R S1 S2) edit

Replace all occurrences of S1 (in the tree you are placed at) by S2.

B edit

Enter a Break loop under the editor.

OK edit

Leave the editor, returning the edited expression.

HELP edit

Print an explanatory message.

If the editor is called from a Break loop, the edited value is assigned back to ErrorForm!*.

4.2. The EMODE Screen Editor

EMODE is an EMACS-like screen editor, written entirely in PSL. To invoke EMODE, call the function EMODE after LOADing the EMODE module. EMODE is modeled after EMACS, so use that fact as a guide.

After starting up EMODE, you can use one of the following commands to exit.

<Ctrl-X Ctrl-Z>
"quits" to the EXEC (you can continue or start again).

<Ctrl-Z Ctrl-Z>
goes back into "normal" I/O mode.

EMODE is built to run on a Teleray terminal as the default. To use some other terminal you must LOAD in a set of different driver functions after loading EMODE. The following drivers are currently available:

* HP2648A
* TELERAY
* VT100
* VT52
* AAA [Ann Arbor Ambassador]

The sources for these files are on <PSL.EMODE> (logical name PE:). It should be quite easy to modify one of these files for other terminals. See the file PE:TERMINAL-DRIVERS.TXT for some more information on how this works.

An important (but currently somewhat bug-ridden) feature of EMODE is the ability to evaluate expressions that are in your buffer. Use <Meta-E> to evaluate the expression starting on the current line. <Meta-E> (normally) automatically enters two window mode if anything is "printed" to the OUT_WINDOW buffer, which is shown in the lower window. If you don't want to see things being printed to the output window, you can set the variable !*OUTWINDOW to NIL. (Or use the RLisp command "OFF OUTWINDOW;".) This prevents EMODE from automatically going into two window mode if something is printed to OUT_WINDOW. You must still use the "<Ctrl-X> 1" command to enter one window mode initially.

You may also find the <Ctrl-Meta-Y> command useful. This inserts into the current buffer the text printed as a result of the last <Meta-E>.

The function "PrintAllDispatch" prints out the current dispatch table. You must call EMODE before this table is set up.

While in EMODE, the <Meta-?> (meta-question mark) character asks for a command character and tries to print information about it.

The basic dispatch table is (roughly) as follows:

Character	Function	Comments
<Ctrl-@>	SETMARK	
<Ctrl-A>	!$BEGINNINGOFLINE	
<Ctrl-B>	!$BACKWARDCHARACTER	
<Ctrl-D>	!$DELETEFORWARDCHARACTER	
<Ctrl-E>	!$ENDOFLINE	
<Ctrl-F>	!$FORWARDCHARACTER	
Linefeed	!$CRLF	Acts like carriage return
<Ctrl-K>	KILL_LINE	
<Ctrl-L>	FULLREFRESH	

Return	!$CRLF	
<Ctrl-N>	!$FORWARDLINE	
<Ctrl-O>	OPENLINE	
<Ctrl-P>	!$BACKWARDLINE	
<Ctrl-R>		Backward search for string, type a carriage return to terminate the string
<Ctrl-S>		Forward search for string
<Ctrl-U>		Repeat a command. Asks for count (terminate with a carriage return), then it asks for the command character
<Ctrl-V>	DOWNWINDOW	
<Ctrl-W>	KILL_REGION	
<Ctrl-X>	!$DOCNTRLX	As in EMACS, <Ctrl-X> is a prefix for "fancier" commands
<Ctrl-Y>	INSERT_KILL_BUFFER	Yanks back killed text
<Ctrl-Z>	DOCONTROLMETA	As in EMACS, acts like <Ctrl-Meta->
escape	ESCAPEASMETA	As in EMACS, escape acts like the <Meta-> key
rubout	!$DELETEBACKWARDCHARACTER	
<Ctrl-Meta-B>	BACKWARD_SEXPR	
<Ctrl-Meta-F>	FORWARD_SEXPR	
<Ctrl-Meta-K>	KILL_FORWARD_SEXPR	
<Ctrl-Meta-Y>	INSERT_LAST_EXPRESSION	Insert the last "expression" typed as the result of a <Meta-E>
<Ctrl-Meta-Z>	OLDFACE	Leave EMODE, go back to "regular" RLISP
<Meta-Ctrl-rubout>	KILL_BACKWARD_SEXPR	
<Meta-<>	!$BEGINNINGOFBUFFER	As in EMACS, move to beginning of buffer
<Meta->>	!$ENDOFBUFFER	As in EMACS, move to end of buffer
<Meta-?>	!$HELPDISPATCH	Asks for a character, tries to print information about it
<Meta-B>	BACKWARD_WORD	
<Meta-D>	KILL_FORWARD_WORD	
<Meta-E>		Evaluate an expression
<Meta-V>	UPWINDOW	As in EMACS, move up a window
<Meta-W>	COPY_REGION	
<Meta-X>	!$DOMETAX	As in EMACS, <Meta-X> is another prefix for "fancy" stuff
<Meta-Y>	UNKILL_PREVIOUS	As in EMACS
<Meta-Rubout>	KILL_BACKWARD_WORD	
<Ctrl-X> <Ctrl-B>	PRINTBUFFERNAMES	Prints a list of buffers
<Ctrl-X> <Ctrl-R>	CNTRLXREAD	Read a file into the buffer
<Ctrl-X> <Ctrl-W>	CNTRLXWRITE	Write the buffer out to a file
<Ctrl-X> <Ctrl-X>	EXCHANGEPOINTANDMARK	
<Ctrl-X> <Ctrl-Z>		As in EMACS, exits to the EXEC
<Ctrl-X> 1	ONEWINDOW	Go into one window mode
<Ctrl-X> 2	TWOWINDOWS	Go into two window mode
<Ctrl-X> B	CHOOSEBUFFER	EMODE asks for a buffer name, and then puts you in that buffer
<Ctrl-X> O	OTHERWINDOW	Select other window
<Ctrl-X> P	WRITESCREENPHOTO	Write a "photograph" of the screen to a file

4.2.1. Windows and Buffers in Emode

[??? This section to be completed at a later date. ???]

4.3. Introduction to the Full Structure Editor

PSL also provides an extremely powerful form-oriented editor[1]. This facility allows the user to easily alter function definitions, variable values and property list entries. It thereby makes it entirely unnecessary for the user to employ a conventional text editor in the maintenance of programs. This document is a guide to using the editor. Certain features of the UCI LISP editor have not been incorporated in the translated editor, and we have tried to mark all such differences.

4.3.1. Starting the Structure Editor

This section describes normal user entry to the editor (with the EditF, EditP and EditV fuunctions) and the editing commands which are available. This section is by no means complete. In particular, material covering programmed calls to the editor routines is not treated. Consult the UCI LISP manual for further details.

To edit a function named FOO do

```
*(EDITF FOO)
```

To edit the value of an atom named BAZ do

```
*(EDITV BAZ)
```

To edit the property list of an atom named FOOBAZ do

```
*(EDITP FOOBAZ)
```

These functions are described later in the chapter.

Warning: Editing the property list of an atom may position pointers at unprintable structures. It is best to use the F (find) command before trying to print property lists. This editor capability is variable from implementation to implementation.

The editor prompts with

```
-E-
*
```

You can then input any editor command. The input scanner is not very smart. It terminates its scan and begins processing when it sees a printable character immediately

[1]This version of the UCI LISP editor was translated to to Standard LISP by Tryg Ager and Jim MacDonald of IMSSS. Stanford, and adapted to PSL by E. Benson. The UCI LISP editor is derived from the Interlisp editor.

followed by a carriage return. Do not use escape to terminate an editor command. If the editor seems to be repeatedly requesting input type P<ret> (print the current expression) or some other command that ordinarily does no damage, but terminates the input solicitation.

The following set of topics makes a good "first glance" at the editor.

Entering the editor:	EDITF, EDITV.
Leaving the editor:	OK.
Editor's attention:	CURRENT-EXP.
Changing attention:	POS-INTEGER, NEG-INTEGER, 0, ^, NX, BK.
Printing:	P, PP.
Modification:	POS-INTEGER, NEG-INTEGER, A, B, :, N.
Changing parens:	BI, BO.
Undoing changes:	UNDO.

For the more discriminating user, the next topics might be some of the following.

Searches:	PATTERN, F, BF.
Complex commands:	R, SW, XTR, MBD, MOVE.
Changing parens:	LI, LO, RI, RO.
Undoing changes:	TEST, UNBLOCK, !UNDO.

Other features should be skimmed but not studied until it appears that they may be useful.

4.3.2. Structure Editor Commands

Note that arguments contained in angle brackets <> are optional.

A ([ARG]) edit

This command inserts the ARGs (arbitrary LISP expressions) After the current expression. This is accomplished by doing an UP and a (-2 exp1 exp2 ... expn) or an (N exp1 exp2 ... expn), as appropriate. Note the way in which the current expression is changed by the UP.

B ([ARG]) edit

This command inserts the ARGs (arbitrary LISP forms) Before the current expression. This is accomplished by doing an UP followed by a (-1 exp1 exp2 ... expn). Note the way in which the current expression is changed by the UP.

BELOW (COM, <N>) edit

This command changes the current expression in the following manner. The edit command COM is executed. If COM is not a recognized command, then (_ COM) is executed instead. Note that COM should cause ascent in the edit chain (i.e. should be equivalent to some number of zeros). BELOW then evaluates (note!) N and descends N links in the resulting edit chain. That is, BELOW ascends the edit chain (does repeated 0s) looking for the link specified by COM and stops N links below that (backs off N 0s). If N is not given, 1 is assumed.

BF (PAT, <FLG>) edit

Also can be used as:

BF PAT

This command performs a Backwards Find, searching for PAT (an edit pattern). Search begins with the expression immediately before the current expression and proceeds in reverse print order. (If the current expression is the top level expression, the entire expression is searched in reverse print order.) Search begins at the end of each list, and descends into each element before attempting to match that element. If the match fails, proceed to the previous element, etc. until the front of the list is reached. At that point, BF ascends and backs up, etc.

The search algorithm may be slightly modified by use of a second argument. Possible FLGs and their meanings are as follows.

T	begins search with the current expression rather than with the preceding expression at this level.
NIL	or missing - same as BF PAT.

NOTE: if the variable UPFINDFLG is non-NIL, the editor does an UP after the expression matching PAT is located. Thus, doing a BF for a function name yields a current expression which is the entire function call. If this is not desired, UPFINDFLG may be set to NIL. UPFINDFLG is initially T.

BF is protected from circular searches by the variable MAXLEVEL. If the total number of Cars and Cdrs descended into reaches MAXLEVEL (initially 300), search of that tail or element is abandoned exactly as though a complete search had failed.

BI (N1, N2) edit

This command inserts a pair of parentheses in the current expression; i.e. it is a Balanced Insert. (Note that parentheses are ALWAYS balanced, and hence must be added or removed in pairs.) A left parenthesis is inserted before element N1 of the current expression. A right parenthesis is inserted after element N2 of the current expression. Both N1 and N2 are usually integers, and element N2 must be to the right of element N1.

(BI n1) is equivalent to (BI n1 n1).

The NTH command is used in the search, so that N1 and N2 may be any location specifications. The expressions used are the first element of the current expression in which the specified form is found at any level.

BIND ([COM]) edit

This command provides the user with temporary variables for use during the execution of the sequence of edit commands coms. There are three variables available: #1, #2 and #3. The binding is recursive and BIND may be executed recursively if necessary. All variables are initialized to NIL. This feature is useful chiefly in defining edit macros.

BK edit

The current expression becomes the expression immediately preceding the present current expression; i.e. Back Up. This command generates an error if the current expression is the first expression in the list.

BO (N) edit

The BO command removes a pair of parentheses from the Nth element of the current expression; i.e. it is a Balanced Remove. The parameter N is usually an integer. The NTH command is used in the search, however, so that any location specification may be used. The expression referred to is the first element of the current expression in which the specified form is found at any level.

(CHANGE LOC To [ARG]) edit

This command replaces the current expression after executing the location specification LOC by ARGs.

(COMS [ARG]) edit

This command evaluates its ARGs and executes them as edit commands.

(COMSQ [ARG]) edit

This command executes each ARG as an edit command.

At any given time, the attention of the editor is focused on a single expression or form. We call that form the current expression. Editor commands may be divided into two broad classes. Those commands which change the current expression are called attention- changing commands. Those commands which modify structure are called structure modification commands.

DELETE edit

This command deletes the current expression. If the current expression is a tail, only the first element is deleted. This command is equivalent to (:).

(E FORM <T>) edit

This command evaluates FORM. This may also be typed in as:

```
E FORM
```

but is valid only if typed in from the TTY. (E FORM) evaluates FORM and prints the value on the terminal. The form (E FORM T) evaluates FORM but does not print the result.

(EditF FN:id): any expr

This function initiates editing of the function whose name is FN.

(EditFns FN-LIST:id-list, COMS:form): NIL fexpr

This function applies the sequence of editor commands, COMS, to each of several functions. The argument FN-LIST is evaluated, and should evaluate to a list of function names. COMS is applied to each function in FN-LIST, in turn. Errors in editing one function do not affect editing of others. The editor call is via EditF, so that values may also be edited in this way.

(EditP AT:id, COMS:form-list): any fexpr

This function initiates editing of the property list of the atom whose name is at. The argument COMS is a possibly null sequence of edit commands which is executed before calling for input from the terminal.

(EditV AT:id, COMS:forms-list): NIL fexpr

This function initiates editing of the value of the atom whose name is AT. The argument COMS is a possibly null sequence of edit commands which is executed before calling for input from the terminal.

(EMBED LOC In ARG) edit

This command replaces the expression which would be current after executing the location specification LOC by another expression which has that expression as a sub-expression. The manner in which the transformation is carried out depends on the form of ARG. If ARG is a list, then each occurrence of the atom '*' in ARG is replaced by the expression which would be current after doing LOC. (NOTE: a fresh copy is used for each substitution.) If ARG is atomic, the result is equivalent to:

```
(EMBED loc IN (arg *))
```

A call of the form

```
(EMBED loc IN exp1 exp2 ... expn)
```

is equivalent to:

```
(EMBED loc IN (exp1 exp2 ... expn *))
```

If the expression after doing LOC is a tail, EMBED behaves as though the expression were the first element of that tail.

(EXTRACT LOC1 From LOC2) edit

This command replaces the expression which would be current after doing the location specification LOC2 by the expression which would be current after doing LOC1. The expression specified by LOC1 must be a sub-expression of that specified by LOC2.

(F PAT <FLG>) edit

Also can be used as:

```
F PAT
```

This command causes the next command, PAT, to be interpreted as a pattern. The current expression is searched for the next occurrence of PAT; i.e. Find. If PAT is a top level element of the current expression, then PAT matches that top level occurrence and a full recursive search is not attempted. Otherwise, the search proceeds in print order. Recursion is done first in the Car and then in the Cdr direction.

The form (F PAT FLG) of the command may be used to modify the search algorithm according to the value of FLG. Possible values and their actions are:

N	suppresses the top-level check. That is, finds the next print order occurrence of PAT regardless of any top level occurrences.
T	like N, but may succeed without changing the current expression. That is, succeeds even if the current expression itself is the only occurrence of PAT.
positive integer	finds the nth place at which PAT is matched. This is equivalent

to (F PAT T) followed by n-1 (F PAT N)s. If n occurrences are not found, the current expression is unchanged.

NIL or missing
Only searches top level elements of the current expression. May succeed without changing the current expression.

NOTE: If the variable **UPFINDFLG** is non-NIL, F does an UP after locating a match. This ensures that F fn, in which fn is a function name, results in a current expression which is the entire function call. If this is undesirable, set **UPFINDFLG** to NIL. Its initial value is T.

As protection against searching circular lists, the search is abandoned if the total number of **Car-Cdr** descents exceeds the value of the variable **MAXLEVEL**. (The initial value is 300.) The search fails just as if the entire element had been unsuccessfully searched.

(FS [PAT]) edit

The FS command does sequential finds; i.e. Find Sequential. That is, it searches (in print order) first for the first PAT, then for the second PAT, etc. If any search fails, the current expression is left at that form which matched in the last successful search. This command is, therefore, equivalent to a sequence of F commands.

(F= EXP FLG) edit

This command is equivalent to (F (== exp) flg); i.e. Find Eq. That is, it searches, in the manner specified by FLG, for a form which is Eq to EXP. Note that for keyboard type-ins, this always fails unless EXP is atomic.

HELP edit

This command provides an easy way of invoking the HELP system from the editor.

(I COM [ARG]) edit

This command evaluates the ARGs and executes COM on the resulting values. This command is thus equivalent to: (com val1 val2 ... valn), Each vali is equal to (EVAL argi).

(IF ARG) edit

This command, useful in edit macros, conditionally causes an editor error. If (EVAL arg) is NIL (or if evaluation of arg causes a LISP error), then IF generates an editor error.

(**INSERT** [EXP ARG LOC]) edit

The **INSERT** command provides equivalents of the A, B and : commands incorporating a location specification, LOC. ARG can be AFTER, BEFORE, or FOR. This command inserts EXPs AFTER, BEFORE or FOR (in place of) the expression which is current after executing LOC. Note, however, that the current expression is not changed.

(LC LOC) edit

This command, which takes as an argument a location specification, explicitly invokes the location specification search; i.e. Locate. The current expression is changed to that which is current after executing LOC.

See LOC-SPEC for details on the definition of LOC and the search method in question.

(LCL LOC) edit

This command, which takes as an argument a location specification, explicitly invokes the location specification search. However, the search is limited to the current expression itself; i.e. Locate Limited. The current expression is changed to that which is current after executing LOC.

(LI N) edit

This command inserts a left parenthesis (and, of course, a matching right parenthesis); i.e. Left Parenthesis Insert. The left parenthesis is inserted before the Nth element of the current expression and the right parenthesis at the end of the current expression. Thus, this command is equivalent to (BI n -1).

The **NTH** command is used in the search, so that N, which is usually an integer, may be any location specification. The expression referred to is the first element of the current expression which contains the form specified at any level.

(LO N) edit

This command removes a left parenthesis (and a matching right parenthesis, of course) from the Nth element of the current expression; i.e. Left Parenthesis Remove. All elements after the Nth are deleted.

The command uses the **NTH** command for the search. The parameter N, which is usually an integer, may be any location specification. The expression actually referred to is the first element of the current expression which contains the specified form at any depth.

Many of the more complex edit commands take as an argument a location specification (abbreviated LOC throughout this document). A location specification is a list of edit commands, which are, with two exceptions, executed in the normal way. Any command not recognized by the editor is treated as though it were preceded by F. Furthermore, if one of the commands causes an error and the current expression has been changed by prior commands, the location operation continues rather than aborting. This is a sort of back-up operation. For example, suppose the location specification is (COND 2 3), and the first clause of the first Cond has only 2 forms. The location operation proceeds by searching for the next Cond and trying again. If a point were reached in which there were no more Conds, the location operation would then fail.

(LP COMS) edit

This command, useful in macros, repeatedly executes COMS (a sequence of edit commands) until an editor error occurs; i.e. Loop. As LP exits, it prints the number of OCCURRENCES; that is, the number of times COMS was successfully executed. After execution of the command, the current expression is left at what it was after the last complete successful execution of COMS.

The command terminates if the number of iterations exceeds the value of the variable MAXLOOP (initially 30).

(LPQ COMS) edit

This command, useful in macros, repeatedly executes COMS (a sequence of edit commands) until an editor error occurs; i.e. Loop Quietly. After execution of the command, the current expression is left at what it was after the last complete successful execution of COMS.

The command terminates if the number of iterations exceeds the value of the variable MAXLOOP (initially 30).

This command is equivalent to LP, except that OCCURRENCES is not printed.

(M (NAM) ([EXP) COMS)]) edit

This can also be used as:

```
(M NAM COMS)
```

or as:

```
(M (NAM) ARG COMS)
```

The editor provides the user with a macro facility; i.e. M. The user may define frequently used command sequences to be edit macros, which may then be invoked simply by giving the macro name as an edit command. The M command provides the user with a method of defining edit macros.

The first alternate form of the command defines an atomic command which takes no arguments. The argument NAM is the atomic name of the macro. This defines NAM to be an edit macro equivalent to the sequence of edit commands COMS. If NAM previously had a definition as an edit macro, the new definition replaces the old. NOTE: Edit command names take precedence over macros. It is not possible to redefine edit command names.

The main form of the M command as given above defines a list command, which takes a fixed number of arguments. In this case, NAM is defined to be an edit macro equivalent to the sequence of edit commands COMS. However, as (nam exp1 exp2 ... expn) is executed, the expi are substituted for the corresponding argi in COMS before COMS are executed.

The second alternate form of the M command defines a list command which may take an arbitrary number of arguments. Execution of the macro NAM is accomplished by substituting (exp1 exp2 ... expn) (that is, the Cdr of the macro call (nam exp1 exp2 ... expn)) for all occurrences of the atom ARG in COMS, and then executing COMS.

(MAKEFN (NAM VARS) ARGS N1 <N2>) edit

This command defines a portion of the current expression as a function and replaces that portion of the expression by a call to the function; i.e. Make Function. The form (NAM VARS) is the call which replaces the N1st through N2nd elements of the current expression. Thus, NAM is the name of the function to be defined. VARS is a sequence of local variables (in the current expression), and ARGS is a list of dummy variables. The function definition is formed by replacing each occurrence of an element in vars (the Cdr of (NAM VARS)) by the corresponding element of ARGS. Thus, ARGS are the names of the formal parameters in the newly defined function.

If N2 is omitted, it is assumed to be equal to N1.

MARK edit

This command saves the current position within the form in such a way that it can later be returned to. The return is accomplished via _ or __.

MBD (ARG) edit

This command replaces the current expression by some form which has the current expression as a sub-expression. If ARG is a list, MBD substitutes a fresh copy of the current expression for each occurrence of the atom '*' in ARG. If ARG is a sequence of expressions, as:

```
(MBD exp1 exp2 ... expn)
```

then the call is equivalent to one of the form:

(MBD (exp1 exp2 ... expn *))

The same is true if arg is atomic:

(MBD atom) = (MBD (atom *))

(MOVE <LOC1> To COM <LOC2>) edit

The MOVE command allows the user to Move a structure from one point to another. The user may specify the form to be moved (via LOC1, the first location specification), the position to which it is to be moved (via LOC2, the second location specification) and the action to be performed there (via COM). The argument COM may be BEFORE, AFTER or the name of a list command (e.g. :, N, etc.). This command performs in the following manner. Take the current expression after executing LOC1 (or its first element, if it is a tail); call it expr. Execute LOC2 (beginning at the current expression AS OF ENTRY TO MOVE -- NOT the expression which would be current after execution of LOC1), and then execute (COM expr). Now go back and delete expr from its original position. The current expression is not changed by this command.

If LOC1 is NIL (that is, missing), the current expression is moved. In this case, the current expression becomes the result of the execution of (COM expr).

If LOC2 is NIL (that is missing) or HERE, then the current expression specifies the point to which the form given by LOC2 is to be moved.

(N [EXP]) edit

This command adds the EXPs to the end of the current expression; i.e. Add at End. This compensates for the fact that the negative integer command does not allow insertion after the last element.

(-N:integer [EXP]) edit-command

Also can be used as:

-N

This is really two separate commands. The atomic form is an attention changing command. The current expression becomes the nth form from the end of the old current expression; i.e. Add Before End. That is, -1 specifies the last element, -2 the second from last, etc.

The list form of the command is a structure modification command. This command inserts exp1 through expn (at least one expi must be present) before the nth element (counting from the BEGINNING) of the current expression. That is, -1 inserts before the first element, -2 before the second, etc.

(NEX COM) edit

Also can be used as:

NEX

This command is equivalent to (BELOW COM) followed by NX. That is, it does repeated 0s until a current expression matching com is found. It then backs off by one 0 and does a NX.

The atomic form of the command is equivalent to (NEX _). This is useful if the user is doing repeated (NEX x)s. He can MARK at x and then use the atomic form.

(NTH LOC) edit

This command effectively performs (LCL LOC), (BELOW <), UP. The net effect is to search the current expression only for the form specified by the location specification LOC. From there, return to the initial level and set the current expression to be the tail whose first element contains the form specified by LOC at any level.

(NX N) edit

Also can be used as:

NX

The atomic form of this command makes the current expression the expression following the present current expression (at the same level); i.e. Next.

The list form of the command is equivalent to n (an integer number) repetitions of NX. If an error occurs (e.g. if there are not N expressions following the current expression), the current expression is unchanged.

OK edit

This command causes normal exit from the editor.

The state of the edit is saved on property LASTVALUE of the atom EDIT. If the next form edited is the same, the edit is restored. That is, it is (with the exception of a BLOCK on the undo-list) as though the editor had never been exited.

It is possible to save edit states for more than one form by exiting from the editor via the SAVE command.

(ORF [PAT]) edit

This command searches the current expression, in print order, for the first occurrence of any form which matches one of the PATs; i.e. Print Order Final. If found, an UP is executed, and the current expression becomes the expression so specified. This command is equivalent to (F (*ANY* pat1 pat2 ... patn) N). Note that the top level check is not performed.

(ORR [COMS]) edit

This command operates in the following manner. Each COMS is a list of edit commands. ORR first executes the first COMS. If no error occurs, ORR terminates, leaving the current expression as it was at the end of executing COMS. Otherwise, it restores the current expression to what it was on entry and repeats this operation on the second COMS, etc. If no COMS is successfully executed without error, ORR generates an error and the current expression is unchanged.

(P N1 <N2>) edit

Also can be used as:

P

This command prints the current expression; i.e. Print. The atomic form of the command prints the current expression to a depth of 2. More deeply nested forms are printed as &.

The form (P N1) prints the N1st element of the current expression to a depth of 2. The argument N1 need not be an integer. It may be a general location specification. The NTH command is used in the search, so that the expression printed is the first element of the current expression which contains the desired form at any level.

The third form of the command prints the N1st element of the current expression to a depth of N2. Again, N1 may be a general location specification.

If N1 is 0, the current expression is printed.

Many of the editor commands, particularly those which search, take as an argument a pattern (abbreviated PAT). A pattern may be any combination of literal list structure and special pattern elements.

The special elements are as follows.

&	this matches any single element.
ANY	if (CAR pat) is the atom *ANY*, then (CDR pat) must be a list of patterns. PAT matches any form which matches any of the

patterns in (Cdr PAT).

@	if an element of pat is a literal atom whose last character is @, then that element matches any literal atom whose initial characters match the initial characters of the element. That is, VER matches VERYLONGATOM.
--	this matches any tail of a list or any interior segment of a list.
==	if (Car PAT) is ==, then PAT matches X iff (Cdr PAT) is Eq to X.
:::	if PAT begins with :::, the Cdr of PAT is matched against tails of the expression.

(N:integer [EXP]) edit-command

Also can be used as:

N:integer

This command, a strictly positive integer N, is really two commands. The atomic form of the command is an attention-changing command. The current expression becomes the nth element of the current expression.

The list form of the command is a structure modification command. It replaces the Nth element of the current expression by the forms EXP. If no forms are given, then the Nth element of the current expression is deleted.

PP edit

This command Pretty-Prints the current expression.

(R EXP1 EXP2) edit

This command Replaces all occurrences of EXP1 by EXP2 in the current expression.

Note that EXP1 may be either the literal s-expression to be replaced, or it may be an edit pattern. If a pattern is given, the form which first matches that pattern is replaced throughout. All forms which match the pattern are NOT replaced.

(REPACK LOC) edit

Also can be used as:

REPACK

This command allows the editing of long strings (or atom names) one character at a time. REPACK calls the editor recursively on UNPACK of the specified atom. (In the atomic form of the command, the current expression is used unless it is a list; then, the first element is used. In the list form of the command, the form specified by the location specification is treated in the same way.) If the lower editor is exited via OK, the result is repacked and replaces the original atom. If STOP is used, no replacement is done. The new atom is always printed.

(RI N1 N2) edit

This command moves a right parenthesis. The parenthesis is moved from the end of the the N1st element of the current expression to after the N2nd element of the N1st element; i.e. Right Parenthesis Insert. Remaining elements of the N1st element are raised to the top level of the current expression.

The arguments, N1 and N2, are normally integers. However, because the NTH command is used in the search, they may be any location specifications. The expressions referred to are the first element of the current expression in which the specified form is found at any level, and the first element of that expression in which the form specified by N2 is found at any level.

(RO N) edit

This command moves the right parenthesis from the end of the nth element of the current expression to the end of the current expression; i.e. Right Parenthesis Remove. All elements following the Nth are moved inside the nth element.

Because the NTH command is used for the search, the argument N, which is normally an integer, may be any location specification. The expression referred to is the first element of the current expression in which the specified form is found at any depth.

(S VAR LOC) edit

This command Sets (via SetQ) the variable whose name is VAR to the current expression after executing the location specification LOC. The current expression is not changed.

SAVE edit

This command exits normally from the editor. The state of the edit is saved on the property EDIT-SAVE of the atom being edited. When the same atom is next edited, the state of the edit is restored and (with the exception of a BLOCK on the undo-list) it is as if the editor had never been exited. It is not necessary to use the SAVE command if only a single atom is being edited. See the OK command.

(SECOND LOC) edit

This command changes the current expression to what it would be after the location specification LOC is executed twice. The current expression is unchanged if either execution of LOC fails.

STOP edit

This command exits abnormally from the editor; i.e. Stop Editing. This command is useful mainly in conjunction with TTY: commands which the user wishes to abort. For example, if the user is executing

```
(MOVE 3 TO AFTER COND TTY:)
```

and he exits from the lower editor via OK, the MOVE command completes its operation. If, on the other hand, the user exits via STOP, TTY: produces an error and MOVE aborts.

(SW N1 N2) edit

This command Swaps the N1st and N2nd elements of the current expression. The arguments are normally but not necessarily integers. SW uses NTH to perform the search, so that any location specifications may be used. In each case, the first element of the current expression which contains the specified form at any depth is used.

TEST edit

This command adds an undo-block to the undo-list. This block limits the scope of UNDO and !UNDO commands to changes made after the block was inserted. The block may be removed via UNBLOCK.

(THIRD LOC) edit

This command executes the location specification loc three times. It is equivalent to three repetitions of (LC LOC). Note, however, that if any of the executions causes an editor error, the current expression remains unchanged.

(LOC1 THROUGH LOC2) edit

This command makes the current expression the segment from the form specified by LOC1 through (including) the form specified by LOC2. It is equivalent to (LC LOC1), UP, (BI 1 LOC2), 1. Thus, it makes a single element of the specified elements and makes that the current expression.

This command is meant for use in the location specifications given to the DELETE, EMBED, EXTRACT and REPLACE commands, and is not particularly useful by itself. Use of THROUGH with these commands sets a special flag so that the editor removes the extra set of parens added by THROUGH.

(LOC1 TO LOC2) edit

This command makes the current expression the segment from the form specified by LOC1 up to (but not including) the form specified by LOC2. It is equivalent to (LC LOC1), UP, (BI 1 loc), (RI 1 -2), 1. Thus, it makes a single element of the specified elements and makes that the current expression.

This command is meant for use in the location specifications given to the DELETE, EMBED, EXTRACT and REPLACE commands, and is not particularly useful by itself. Use of TO with these commands sets a special flag so that the editor removes the extra set of parens added by TO.

TTY: edit

This command calls the editor recursively, invoking a 'lower editor.' The user may execute any and all edit commands in this lower editor. The TTY: command terminates when the lower editor is exited via OK or STOP.

The form being edited in the lower editor is the same as that being edited in the upper editor. Upon entry, the current expression in the lower is the same as that in the upper editor.

UNBLOCK edit

This command removes an undo-block from the undo-list, allowing UNDO and !UNDO to operate on changes which were made before the block was inserted.

Blocks may be inserted by exiting from the editor and by the TEST command.

UNDO (COM) edit

Also can use as:

UNDO

This command undoes editing changes. All editing changes are undoable, provided that the information is available to the editor. (The necessary information is always available unless several forms are being edited and the SAVE command is not used.) Changes made in the current editing session are ALWAYS undoable.

The short form of the command undoes the most recent change. Note, however, that UNDO and !UNDO changes are skipped, even though they are themselves undoable.

The long form of the command allows the user to undo an arbitrary command, not necessarily the most recent. UNDO and !UNDO may also be undone in this manner.

UP edit

If the current expression is a tail of the next higher expression, UP has no effect. Otherwise the current expression becomes the form whose first element is the old current expression.

(XTR LOC) edit

This command replaces the current expression by one of its subexpressions. The location specification, LOC, gives the form to be used. Note that only the current expression is searched. If the current expression is a tail, the command operates on the first element of the tail.

0 edit-command

This command makes the current expression the next higher expression. This usually, but not always, corresponds to returning to the next higher left parenthesis. This command is, in some sense, the inverse of the POS-INTEGER and NEG- INTEGER atomic commands.

([COM:form]): any fexpr, edit-command

The value of this fexpr, useful mainly in macros, is the expression which would be current after executing all of the COMs in sequence. The current expression is not changed.

Commands in which this fexpr might be used (e.g. CHANGE, INSERT, and REPLACE) make special checks and use a copy of the expression returned.

^ edit-command

This command makes the top level expression the current expression.

? edit-command

This command prints the current expression to a level of 100. It is equivalent to (P 0 100).

?? edit-command

This command displays the entries on the undo-list.

_ edit-command

This command returns to the position indicated by the most recent MARK command. The MARK is not removed.

(_ PAT) edit-command

This command ascends (does repeated 0s), testing the current expression at each ascent for a match with PAT. The current expression becomes the first form to match. If pattern is atomic, it is matched with the first element of each expression; otherwise, it is matched against the entire form.

__ edit-command

This command returns to the position indicated by the most recent MARK command and removes the MARK.

(: [EXP]) edit-command

Also can be used as:

(:)

This command replaces the current expression by the forms EXP. If no forms are given (as in the second form of the command), the current expression is deleted.

(PAT :: LOC) edit-command

This command sets the current expression to the first form (in print order) which matches PAT and contains the form specified by the location specification LOC at any level. The command is equivalent to (F PAT N), (LCL LOC), (_ PAT).

\ edit-command

This command returns to the expression which was current before the last 'big jump.' Big jumps are caused by these commands: ^, _, __, !NX, all commands which perform a search or use a location specification, \ itself, and \P. NOTE: \ is shift-L on a teletype.

\P edit-command

This command returns to the expression which was current before the last print operation (P, PP or ?). Only the two most recent locations are saved. NOTE: \ is shift-L on a teletype.

!NX edit-command

This command makes the next expression at a higher level the current expression. That is, it goes through any number of right parentheses to get to the next expression.

!UNDO edit-command

This command undoes all changes made in the current editing session (back to the most recent block). All changes are undoable.

Blocks may be inserted by exiting the editor or by the TEST command.

They may be removed with the UNBLOCK command.

!0 <u>edit-command</u>

This command does repeated 0s until it reaches an expression which is not a tail of the next higher expression. That expression becomes the new current expression. That is, this command returns to the next higher left parenthesis, regardless of intervening tails.

CHAPTER 5
RLISP SYNTAX

5.1. Motivation for RLISP Interface to PSL

Some of the PSL users at Utah prefer to write Lisp code using an Algol-like (or Pascal-like) preprocessor language, RLisp, because of its similarity to the heavily used Pascal and C languages. RLisp was developed as part of the Reduce Computer Algebra Project [Hearn 73], and is the Algol-like user language as well as the implementation language. RLisp provides a number of syntactic niceties which we find convenient, such as vector subscripts, a case statement, an `If-Then-Else` statement, etc. We usually do not distinguish Lisp from RLisp, and can mechanically translate from one to the other in either direction using a parser and pretty-printer written in PSL. That is, RLisp is a convenience, but it is not necessary to use RLisp syntax rather than Lisp. A complete BNF-like definition of RLisp and its translation to Lisp using the MINI system is given in Section 6.4. Also discussed in Chapter 6 is an extensible table driven parser which is used for the current RLisp parser. There we give explicit tables which define RLisp syntax.

In this chapter we provide enough of an introduction to make the PSL sources readable and to assist the user in writing RLisp code.

5.2. An Introduction to RLISP

An RLisp program consists of a set of functional commands which are evaluated sequentially. RLisp expressions are built up from declarations, statements and expressions. Such entities are composed of sequences of numbers, variables, operators, strings, reserved words and delimiters (such as commas and parentheses), which in turn are sequences of characters. The evaluation proceeds by a parser first converting the Algol-like RLisp source language into Lisp S-expressions, and evaluating and printing the result. The basic cycle is thus `Parse-Eval-Print`, although the specific functions and additional processing are under the control of a variety of switches, described in appropriate sections.

5.2.1. LISP equivalents of some RLISP constructs

The following gives a few examples of RLisp statements and functions and their corresponding Lisp forms. To see the exact Lisp equivalent of RLisp code, set the switch **!*PEcho** to T [On PECHO;].

Assignment statements in RLisp and Lisp:

```
X := 1;                        (setq x 1)
```

A procedure to take a factorial, in RLisp:

```
LISP PROCEDURE FACTORIAL N;
  IF N <= 1 THEN 1
   ELSE N * FACTORIAL (N-1);
```

in Lisp:

```
(de factorial (n)
  (cond
    ((leq n 1)  1)
    (T
      (times n (factorial (difference n 1))))))
```

Take the Factorial of 5 in RLisp and in Lisp:

```
FACTORIAL 5;                   (factorial 5)
```

Build a list X as a series of "Cons"es in RLisp:

```
X := 'A . 'B . 'C . NIL;

in Lisp:
(setq x (cons 'a  (cons 'b (cons 'c nil))))
```

5.3. An Overview of RLISP and LISP Syntax Correspondence

The RLisp parser converts RLisp expressions, typed in at the terminal or read from a file, into directly executable Lisp expressions. For convenience in the following examples, the "==>" arrow is used to indicate the Lisp actually produced from the input RLisp. To see the Lisp equivalents of RLisp code on the machine, set the switch **!*PEcho** to T. As far as possible, upper and lower cases are used as follows:

a. Upper case tokens and punctuation represent items which must appear as is in the source RLisp or output Lisp.

b. Lower case tokens represent other legal RLisp constructs or corresponding Lisp translations. We typically use "e" for expression, "s" for statement, and "v" for variable; "-list" is tacked on for lists of these objects.

For example, the following rule describes the syntax of assignment in RLisp:

```
VAR := number;
   ==>  (SETQ VAR number)
```

Another example:

```
IF expression THEN action_1  ELSE action_2
   ==> (COND ((expression action_1) (T action_2)))
```

5.3.1. Function Call Syntax in RLISP and LISP

A function call with N arguments (called an N-ary function) is most commonly represented as "FN(X1, X2, ... Xn)" in RLisp and as "(FN X1 X2 ... Xn)" in Lisp. Commas are required to separate the arguments in RLisp but not in Lisp. A zero argument function call is "FN()" in RLisp and "(FN)" in Lisp. An unary function call is "FN(a)" or "FN a" in RLisp and "(FN a)" in Lisp; i.e., the parentheses may be omitted around the single argument of any unary function in RLisp.

5.3.2. RLISP Infix Operators and Associated LISP Functions

Many important PSL binary functions, particularly those for arithmetic operations, have associated infix operators, consisting of one or two special characters. The conversion of an RLisp expression "A op B" to its corresponding Lisp form is easy: "(fn A B)", in which "fn" is the associated function. The function name fn may also be used as an ordinary RLISP function call, "fn(A, B)".

Refer to Chapter 6 for details on how the association of "op" and "fn" is installed.

Parentheses may be used to specify the order of combination. "((A op_a B) op_b C)" in RLisp becomes "(fn_b (fn_a A B) C)" in Lisp.

If two or more different operators appear in a sequence, such as "A op_a B op_b C", grouping (similar to the insertion of parentheses) is done based on relative precedence of the operators, with the highest precedence operator getting the first argument pair: "(A op_a B) op_b C" if Precedence(op_a) >= Precedence(op_b); "A op_a (B op_b C)" if Precedence(op_a) < Precedence(op_b).

If two or more of the same operator appear in a sequence, such as "A op B op C", grouping is normally left-to-right (Left Associative; i.e., "(fn (fn A B) C)"), unless the operator is explicitly Right Associative (such as . for Cons and := for SetQ; i.e., "(fn A (fn B C))").

The operators + and * are N-ary; i.e., "A nop B nop C nop B" parses into "(nfn A B C D)" rather than into "(nfn (nfn (nfn A B) C) D)".

The current binary operator-function correspondence is as follows: (Operation groups higher on the list are done first.)

Operator	Function	Precedence
.	Cons	23 Right Associative
**	Expt	23
/	Quotient	19
*	Times	19 N-ary
-	Difference	17
+	Plus	17 N-ary
Eq	Eq	15
=	Equal	15
>=	Geq	15
>	GreaterP	15
<=	Leq	15
<	LessP	15
Member	Member	15
Memq	MemQ	15
Neq	Neq	15
And	And	11 N-ary
Or	Or	9 N-ary
:=	SetQ	7 Right Associative

Note: There are other INFIX operators, mostly used as key-words within other syntactic constructs (such as Then or Else in the If-..., or Do in the While-..., etc.). They have lower precedences than those given above. These key-words include: the parentheses "()", the brackets "[]", the colon ":", the comma ",", the semi-colon ";", the dollar sign "$", and the ids: Collect, Conc, Do, Else, End, Of, Procedure, Product, Step, Such, Sum, Then, To, and Until.

As pointed out above, a unary function FN can be used with or without parentheses: FN(a); or FN a;. In the latter case, FN is assumed to behave as a prefix operator with highest precedence (99) so that "FOO 1 ** 2" parses as "FOO(1) ** 2;". The operators +, -, and / can also be used as unary prefix operators, mapping to Plus, Minus and Recip, respectively, with precedence 26. Certain other unary operators (RLisp key-words) have low precedences or explicit special purpose parsing functions. These include: BEGIN, CASE, CONT, EXIT, FOR, FOREACH, GO, GOTO, IF, IN, LAMBDA, NOOP, NOT, OFF, ON, OUT, PAUSE, QUIT, RECLAIM, REPEAT, RETRY, RETURN, SCALAR, SHOWTIME, SHUT, WHILE and WRITE.

5.3.3. Referencing Elements of Vectors in RLISP

In RLisp syntax, X[i]; may be used to access the i'th element of an x-vector, and X[i]:=y; is used to change the i'th element to y. These functions correspond to the Lisp functions Indx and SetIndx.

5.3.4. Differences between Parse and Read

A single character can be interpreted in different ways depending on context and on whether it is used in a Lisp or in an RLisp expression. Such differences are not immediately apparent to a novice user of RLisp, but an example is given below.

The RLisp infix operator "." may appear in an RLisp expression and is converted by the Parse function to the Lisp function Cons, as in the expression x := 'y . 'z;. A dot may also occur in a quoted expression in RLisp mode, in which case it is interpreted by Read as part of the notation for pairs, as in x := '(y . z);. Note that Read called from Lisp uses slightly different scan tables than Read called from RLisp.

What constitutes a valid id name depends upon the scan table in use by the reader. In RLisp the scan table bound to RLISPSCANTABLE!*, shown below. ids begin with a letter or any character preceded by an escape character. They may contain letters, digits, underscores, and escaped characters. You will note that many characters such as "$" and "*" that are treated as letters by the Lisp scan table are treated as delimiters by the RLisp scan table. Characters regarded as delimiters by the RLisp scan table must be preceded by an escape character, currently "!", when they appear in an id name.

Here are some examples of valid id names using the RLisp scan table. Note that the first and second examples are read as the same identifier if !*RAISE is T.

```
* ThisIsALongIdentifier
* THISISALONGIDENTIFIER
* ThisIsALongIdentifierAndDifferentFromTheOther
* this_is_a_long_identifier_with_underscores
* an!-identifier!-with!-dashes
* !*RAISE
* !2222
```

RLISPSCANTABLE!* [Initially: as shown in following table] global

0	^@	IGNORE	32		IGNORE	64	@	DELIMITER	96	'	DELIMITER
1	^A	DELIMITER	33	!	IDESCAPECHAR	65	A	LETTER	97	a	LETTER
2	^B	DELIMITER	34	"	STRINGQUOTE	66	B	LETTER	98	b	LETTER
3	^C	DELIMITER	35	#	DELIMITER	67	C	LETTER	99	c	LETTER
4	^D	DELIMITER	36	$	DELIMITER	68	D	LETTER	100	d	LETTER
5	^E	DELIMITER	37	%	COMMENTCHAR	69	E	LETTER	101	e	LETTER
6	^F	DELIMITER	38	&	DELIMITER	70	F	LETTER	102	f	LETTER
7	^G	DELIMITER	39	'	DELIMITER	71	G	LETTER	103	g	LETTER
8	^H	DELIMITER	40	(	DELIMITER	72	H	LETTER	104	h	LETTER
9	<tab>	IGNORE	41	)	DELIMITER	73	I	LETTER	105	i	LETTER
10	<lf>	IGNORE	42	*	DIPHTHONGSTART	74	J	LETTER	106	j	LETTER
11	^K	DELIMITER	43	+	DELIMITER	75	K	LETTER	107	k	LETTER
12	^L	IGNORE	44	,	DELIMITER	76	L	LETTER	108	l	LETTER
13	<cr>	IGNORE	45	-	DELIMITER	77	M	LETTER	109	m	LETTER
14	^N	DELIMITER	46	.	DECIMALPOINT	78	N	LETTER	110	n	LETTER
15	^O	DELIMITER	47	/	DELIMITER	79	O	LETTER	111	o	LETTER
16	^P	DELIMITER	48	0	DIGIT	80	P	LETTER	112	p	LETTER
17	^Q	DELIMITER	49	1	DIGIT	81	Q	LETTER	113	q	LETTER
18	^R	DELIMITER	50	2	DIGIT	82	R	LETTER	114	r	LETTER
19	^S	DELIMITER	51	3	DIGIT	83	S	LETTER	115	s	LETTER
20	^T	DELIMITER	52	4	DIGIT	84	T	LETTER	116	t	LETTER
21	^U	DELIMITER	53	5	DIGIT	85	U	LETTER	117	u	LETTER
22	^V	DELIMITER	54	6	DIGIT	86	V	LETTER	118	v	LETTER
23	^W	DELIMITER	55	7	DIGIT	87	W	LETTER	119	w	LETTER
24	^X	DELIMITER	56	8	DIGIT	88	X	LETTER	120	x	LETTER
25	^Y	DELIMITER	57	9	DIGIT	89	Y	LETTER	121	y	LETTER
26	^Z	DELIMITER	58	:	DIPHTHONGSTART	90	Z	LETTER	122	z	LETTER
27	$	DELIMITER	59	;	DELIMITER	91	[	DELIMITER	123	{	DELIMITER
28	^\	DELIMITER	60	<	DIPHTHONGSTART	92	\	PACKAGE	124	|	DELIMITER
29	^]	DELIMITER	61	=	DELIMITER	93	]	DELIMITER	125	}	DELIMITER
30	^^	DELIMITER	62	>	DIPHTHONGSTART	94	^	DELIMITER	126	~	DELIMITER
31	^_	DELIMITER	63	?	DELIMITER	95	_	LETTER	127	<rubout>	DELIMITER

The Diphthong Indicator in the 128th entry is the identifier RLISPDIPHTHONG.

5.3.5. Procedure Definition

When defining a function in RLisp, one gives an "ftype" (one of the tokens EXPR, FEXPR, etc.) followed by the keyword PROCEDURE, followed by an "id" (the name of the function), followed by a "v-list" (the formal parameter names) enclosed in parentheses. A semicolon terminates the title line. The body of the function is a <statement> followed by a semicolon.

```
    mode ftype PROCEDURE name(v_1,...,v_n); body;
       ==> (Dx name (v_1 ... v_N) body)
```

In the general definition given above "mode" is usually optional; it can be LISP or SYMBOLIC (which mean the same thing), ALGEBRAIC (for Reduce code), or SYSLISP [only of importance if SYSLisp and Lisp are inter-mixed]. "Ftype" is expr, fexpr, macro, nexpr, or smacro (or can be omitted, in which case it defaults to expr). Name(v_1,...,v_N) is any legal form of call, including infix. Dx is De for _expr_, Df for _fexpr_, Dm for _macro_, Dn for _nexpr_, and Ds for _smacro_.

Examples:

```
EXPR PROCEDURE NULL(X);
  EQ(X, NIL);
   ==>  (DE NULL (X) (EQ X NIL))

PROCEDURE ADD1 N;
  N+1;
   ==> (DE ADD1 (N) (PLUS N 1))

MACRO PROCEDURE FOO X;
  LIST('FUM, CDR X, CDR X);
   ==> (DM FOO (X) (LIST 'FUM (CDR X) (CDR X))
```

The value returned by the procedure is the value of the body; no assignment to the function name (as in Algol or Pascal) is needed.

5.3.6. Compound Statement Grouping

A group of RLisp expressions may be used in any position in which a single expression is expected by enclosing the group of expressions in double angle brackets,"<<" and ">>", and separating them by the ";" delimiter.

The RLisp <<A; B; C; ... Z>> becomes (PROGN A B C ... Z) in Lisp. The value of the group is the value of the last expression, Z.
Example:

```
X:=<<PRINT X; X+1>>;          % prints old X then increments X
  ==> (SETQ X (PROGN (PRINT X) (PLUS X 1)))
```

5.3.7. Blocks with Local Variables

A more powerful construct, sometimes used for the same purpose as the "<< >>" group, is the Begin-End block in RLisp or Prog in Lisp. This construct also permits the allocation of 0 or more local variables, initialized to NIL. The normal value of a block is NIL, but it may be exited at a number of points, using the Return statement, and each can return a different value. The block also permits labels and a GoTo construct.

Example:

```
   BEGIN SCALAR X,Y;  % SCALAR declares locals X and Y
           X:='(1 2 3);
     L1:   IF NULL X THEN RETURN Y;
           Y:=CAR X;
           X:=CDR X;
           GOTO L1;
   END;

    ==> (PROG (X Y)
          (SETQ X '(1 2 3))
     L1   (COND ((NULL X)  (RETURN Y)))
          (SETQ Y (CAR X))
          (SETQ X (CDR X))
          (GO L1))
```

5.3.8. The If Then Else Statement

The Lisp function Cond corresponds to the If statement of most programming languages. In RLisp this is simply the familiar If ... Then ... Else construct. For example:

```
   IF predicate THEN action1
    ELSE action2

      ==> (COND (predicate action1)
                (T action2))
```

Action1 is evaluated if the predicate has a non-NIL evaluation; otherwise, action2 is evaluated. Dangling Elses are resolved in the Algol manner by pairing them with the nearest preceding Then. For example:

```
   IF F(X) THEN
    IF G(Y) THEN PRINT(X)
     ELSE PRINT(Y);
```

is equivalent to

```
   IF F(X) THEN
    << IF G(Y) THEN PRINT(X)
        ELSE PRINT(Y) >>;
```

Note that if F(X) is NIL, nothing is printed.

The If...Then construct also exists.

```
    IF predicate THEN action;

      ==>  (COND (predicate action))
```

The actions may also contain the special functions Go, Return, Exit, and Next, subject to the constraints on placement of these functions described in the control flow chapter of Part 1 of this manual.

5.3.9. Case Statement

PSL provides a numeric case statement that is compiled quite efficiently; some effort is made to examine special cases (compact vs. non-compact sets of cases, short vs. long sets of cases, etc.). It has mostly been used in SYSLisp mode, but can also be used from Lisp mode provided that case-tags are numeric. The FEXPR Case can also be used interpretively.

The RLisp syntax is:

```
Case-Statement ::= CASE expr OF case-clause END

Case-clause     ::=  Case-expr [; Case-clause ]

Case-expr       ::=  Tag-expr : expr

Tag-expr        ::=  DEFAULT | OTHERWISE   |
                     tag | tag, tag ... tag |
                     tag TO tag

Tag             ::=  Integer | Wconst-Integer
```

An example in RLisp is:

```
            CASE i OF
               1:       Print("First");
               2,3:     Print("Second");
               4 to 10: Print("Third");
               Default: Print("Fourth");
            END
```

5.4. Looping Statements

RLisp provides While, Repeat, For and For Each loops. These are discussed in greater detail in the chapter on control flow in Part 1 of this manual. Some examples follow:

5.4.1. While Loop

```
   WHILE e DO s;            % As long as e NEQ NIL, do s
      ==>  (WHILE e s)
```

5.4.2. Repeat Loop

```
   REPEAT s UNTIL e;        % repeat doing s until "e" is not NIL
      ==>  (REPEAT s e)
```

5.4.3. Next and Exit

Next and Exit, described in the control flow chapter of Part 1, are available in RLisp. Care must be taken in using them in While and Repeat loops. While and Repeat each macro expand into a Prog; Next and Exit are macro expanded into a Go and a Return respectively to the Prog immediately containing the Next or Exit. Thus using a Next or an Exit within a Prog within a While or Repeat will result only in an exit of the internal Prog.

In RLisp be careful to use

```
    WHILE E DO << S1;...;EXIT(1);...;Sn>>
```

not

```
    WHILE E DO BEGIN S1;...;EXIT(1);...;Sn;END;
```

5.4.4. For Each Loop

The For Each loops provide various mapping options, processing elements of a list in some way and sometimes constructing a new list.

```
   FOR EACH x IN y DO s;    % y is a list, x traverses list bound to each
                            % element in turn.
      ==>  (FOREACH x IN y DO s)

   FOR EACH x ON y DO s;    % y is a list, x traverses list Bound to successive
                            % Cdr's of y.
      ==>  (FOREACH x ON y DO s)
```

Other options can return modified lists, etc. See the chapter on control flow in Part 1 of this manual.

Note that FOR EACH may be written as FOREACH

Examples of use of the ForEach function follow.

```
[1] X := '(1 3 5);
[2] Foreach Y in X do Print Y;
1
3
5
NIL
[3] Foreach Y in X collect add1 Y;
(2 4 6)
[4] Foreach Y on X do Print Y;
(1 3 5)
(3 5)
(5)
NIL
```

5.4.5. For Loop

The For loop permits an iterative form with a compacted control variable. Other options can compute sums and products.

```
FOR i := a:b DO s;       % step i successively from a to b in
                         % steps of 1.
   ==> (FOR (FROM I a b 1) DO s)

FOR i := a STEP b UNTIL c DO s; % More general stepping
   ==> (FOR (FROM I a c b) DO s)
```

5.4.6. Loop Examples

```
  LISP PROCEDURE count lst; % Count elements in lst
   BEGIN SCALAR k;
         k:=0;
         WHILE PAIRP lst DO <<k:=k+1; lst:=CDR lst>>;
         RETURN k;
   END;

     ==>  (DE COUNT (LST)
             (PROG (K)
                (SETQ K 0)
                (WHILE (PAIRP LST)
                        (PROGN
                          (SETQ K (PLUS K 1))
                          (SETQ LST (CDR LST))))
                (RETURN K)))
```

or

```
  LISP PROCEDURE CountNil lst; % Count  NIL elements in lst
   BEGIN SCALAR k;
         k:=0;
         FOR EACH x IN lst DO If Null x then k:=k+1;
         RETURN k;
   END;

     ==>  (DE COUNTNIL (LST)
             (PROG (K)
                (SETQ K 0)
                (FOREACH X IN LST DO (COND
                        ((NULL X) (SETQ K (PLUS K 1)))))
                (RETURN K)))
```

5.5. RLISP Specific Input/Output

RLisp provides some special commands not available in Lisp syntax for file input and output. These are described in this section. Also specific to RLisp is a function RPrint that prints a form in RLisp format.

(RPrint U:form): NIL <u>expr</u>

Print in RLisp format. Autoloading.

5.5.1. RLISP File Reading Functions

The following functions are present in RLisp, they can be used from Bare-PSL by loading RLISP.

(In [L:string]): None Returned macro

Similar to DskIn but expects RLisp syntax in the files it reads unless it can determine that the files are not in RLisp syntax. Also In can take more than one file name as an argument. On most systems the function In expects files with extension .LSP and .SL to be written in Lisp syntax, not in RLisp. This is convenient when using both Lisp and RLisp files. It is conventional to use the extension .RED (or .R) for RLisp files and use .LSP or .SL only for fully parenthesized Lisp files. There are some system programs, such as TAGS on the DEC-20, which expect RLisp files to have the extension .RED.

If it is not desired to have the contents of the file echoed as it is read, either end the In command with a "$" in RLisp, as

```
In "FILE1.RED","FILE2.SL"$
```

or include the statement "Off ECHO;" in your file.

(EvIn L:string-list): None Returned expr

L must be a list of strings that are filenames. EvIn is the function called by In after evaluating its arguments. In is useful only at the top-level, while EvIn can be used inside functions with file names passed as parameters.

5.5.2. RLISP File Output

(Out U:string): None Returned macro

Opens file U for output, redirecting standard output. Note that Out takes a string as an argument, while Wrs takes an io-channel.

(EvOut L:string-list): None Returned expr

L is a list containing one file name which must be a string. EvOut is the function called by Out after evaluating its argument.

(Shut [L:string]): None Returned macro

Closes the output files in the list L. Note that Shut takes file names as arguments, while Close takes an io-channel. The RLisp IN function maintains a stack of (file-name . io-channel) associations for this purpose. Thus a shut will also correctly select the previous file for further output.

(EvShut L:string-list): none Returned *expr*

Does the same as Shut but evaluates its arguments.

5.6. Transcript of a Short Session with RLISP

The following is a transcript of RLisp running on the DEC-20.

```
@psl:rlisp
Extended 20-PSL 3.2 Rlisp, 21-Jul-83
[1] % Notice the numbered prompt.
[1] % Comments begin with "%" and do not change the prompt number.
[1] Z := '(1 2 3);              % Make an assignment for Z.
(1 2 3)
[2] Cdr Z;                      % Notice the change in the prompt number.
(2 3)
[3] Procedure Count L;     % "Count" counts the number of elements
[3]   If Null L Then 0          %    in a list L.
[3]     Else 1 + Count Cdr L;
COUNT
[4] Count Z;                    % Try out "Count" on Z.
3
[5] Tr Count;          % Trace the recursive execution of "Count".
(COUNT)
[6]                    % A call on "Count" now shows the value of
[6]                    %   "Count" and of its argument each time it
[6] Count Z;           %   is called.
COUNT being entered
   L:   (1 2 3)
  COUNT (level 2) being entered
     L: (2 3)
    COUNT (level 3) being entered
       L:      (3)
      COUNT (level 4) being entered
         L:     NIL
      COUNT (level 4) = 0
    COUNT (level 3) = 1
  COUNT (level 2) = 2
COUNT = 3
3
[7] UnTr Count;
NIL
[8] Count 'A;
***** An attempt was made to do CDR on 'A', which is not a pair
Break loop
1 lisp break> ?
BREAK():{Error,return-value}
----------------------------
This is a Read-Eval-Print loop, similar to the top level loop, except
that the following IDs at the top level cause functions to be called
rather than being evaluated:
?         Print this message, listing active Break IDs
T         Print stack backtrace
Q         Exit break loop back to ErrorSet
```

```
C         Return last value to the ContinuableError call
R         Reevaluate ErrorForm!* and return
M         Display ErrorForm!* as the "message"
E         Invoke a simple structure editor on ErrorForm!*
                (For more information do Help Editor.)
I         Show a trace of any interpreted functions

See the manual for details on the Backtrace, and how ErrorForm!* is
set.  The Break Loop attempts to use the same TopLoopRead!* etc, as
the calling top loop, just expanding the PromptString!*.
NIL
2 lisp break>           % Get a Trace-Back of the
2 lisp break> I         %    interpreted functions.
Backtrace, including interpreter functions, from top of stack:
CDR COUNT PLUS2 PLUS COND COUNT
NIL
3 lisp break> Q             % To exit the Break Loop.
[9]                         % Load in a file, showing the file
[9] In "small-file.red";    % and its execution.
X := 'A . 'B . NIL;(A B)    % Construct a list with "." for Cons.

Count X;2                   % Call "Count" on X.

Reverse X;(B A)             % Call "Reverse" on X.

NIL
[10]                        % This leaves RLISP and enters
[10] End;                   %   LISP mode.
Entering LISP...
PSL, 27-Oct-82
6 lisp> (SETQ X 3)          % A LISP assignment statement.
3
7 lisp> (FACTORIAL 3)       % Call "Factorial" on 3.
6
8 lisp> (BEGINRLISP)        % This function returns us to RLISP.
Entering RLISP...
[11] Quit;                  % To exit call "Quit".
@continue
"Continued"
[12] X;                     % Notice the prompt number.
3
[13] ^C                     % One can also quit with <Ctrl-C>.
@start                      % Alternative immediate re-entry.
[14] Quit;
@
```

CHAPTER 6
PARSER TOOLS

6.1. Introduction

In many applications, it is convenient to define a special "problem-oriented" language, tailored to provide a natural input format. Examples include the RLisp Algol-like surface language for algebraic work, graphics languages, boolean query languages for data-base, etc. Another important case is the requirement to accept existing programs in some language, either to translate them to another language, to compile to machine language, to be able to adapt existing code into the PSL environment (e.g. mathematical libraries, etc.), or because we wish to use PSL based tools to analyze a program written in another language. One approach is to hand-code a program in PSL (called a "parser") that translates the input language to the desired form; this is tedious and error prone, and it is more convenient to use a "parser-writing-tool".

In this Chapter we describe in detail two important parser writing tools available to the PSL programmer: an extensible table-driven parser that is used for the RLisp parser (described in Chapter 5), and the MINI parser generator. The table-driven parser is most useful for languages that are simple extensions of RLisp, or in fact for rapidly adding new syntactic constructs to RLisp. The Mini system is used for the development of more complete user languages.

6.2. The Table Driven Parser

The parser is a top-down recursive descent parser, which uses a table of Precedences to control the parse; if numeric precedence is not adequate, Lisp functions may be inserted into the table to provide more control. The parser described here was developed by Nordstrom [Nordstrom 73], and is very similar to parser described by Pratt [Pratt 73], and apparently used for the CGOL language, another Lisp surface language.

The parser reads tokens from an input stream using a function Scan. Scan calls the ChannelReadToken function described in the chapter on I/O in Part 1 of this manual, and performs some additional checks, described below. Each token is defined to be one of the following:

non-operator	O
right operator	O->
binary operator	<-O->

All combinations of . . .O-> O. . . and O <-O->. . . are supposed to be legal, while the combinations . . .O-> <-O->. . ., . . .<-O-> <-O->. . . and O O. . . are normally illegal (error ARG MISSING and error OP MISSING, respectively).

With each operator (which must be an id) is associated a construction function, a right precedence, and for binary operators, a left precedence.

The Unary Prefix operators have this information stored under the indicator 'RlispPrefix and Binary operators have it stored under 'RlispInfix. (Actually, the indicator used at any time during parsing is the VALUE of GramPrefix or GramInfix, which may be changed by the user).

6.2.1. Flow Diagram for the Parser

In this diagram RP stands for Right Precedence, LP for Left Precedence and CF for Construction Function. OP is a global variable which holds the current token.

```
     procedure PARSE(RP);
      RDRIGHT(RP,SCAN()); % SCAN reads next token

                 RDRIGHT(RP,Y)
                       |
                      \|/
                       |
            ------------------------
            |                      |yes
            |      Y is Right OP   |-----> Y:=APPLY(Y.CF,
            |                      |                RDRIGHT(Y.RP));
            ------------------------                .
                       |                            .
                      \|/ no                        .
                       |                            .
            ------------------------                .
ERROR    yes|                      | no             .
ARG    <----|      Y is Binary OP  |----> OP:=      .
MISSING     |                      |       SCAN();  .
            ------------------------           .    .
                       |--------<------------<------*
            RDLEFT:   \|/                           ^
                       |                            ^
            ------------------------                ^
ERROR     no|                      |                ^
 OP    <----|     OP is Binary     |                ^
MISSING     |                      |                ^
            ------------------------                ^
                       |                            ^
                      \|/  yes                      ^
                       |                            ^
            ------------------------                ^
RETURN   yes|                      |no              ^
 (Y)   <----|   RP > OP.lp         |---> Y:=APPLY(OP.cf,Y,
            ------------------------       PARSE(OP.lp,SCAN());
```

This diagram reflects the major behavior, though some trivial additions are included in the RLisp case to handle cases such as OP-> <-OP, '!;, etc. [See PU:RLISP-PARSER.RED for full details.]

The technique involved may also be described by the following figure:

```
. . . 0-> Y <-0 . . .
      rp   lp
```

Y is a token or an already parsed expression between two operators (as indicated). If 0->'s RP is greater than <-0's LP, then 0-> is the winner and Y goes to 0->'s construction function (and vice versa). The result from the construction function is a "new Y" in another parse situation.

By associating precedences and construction functions with the operators, we are now able to parse arithmetic expressions (except for function calls) and a large number of syntactical constructions such as IF - THEN - ELSE - ; etc. The following discussion of how to expand the parser to cover a language such as RLisp (or Algol) may also be seen as general tools for handling the parser and defining construction functions and precedences.

6.2.2. Associating the Infix Operator with a Function

The Scan, after calling RAtomHook, checks ids and special ids (those with TokType!* = 3) to see if they should be renamed from external form to internal form (e.g. '!+ to Plus2). This is done by checking for a NEWNAM or NEWNAM!-OP property on the id. For special ids, the NEWNAM!-OP property is first checked. The value of the property is a replacement token, i.e.

```
PUT('!+,'NEWNAM!-OP,'PLUS2)
```

has been done.

Scan also handles the ' mark, calling RlispRead to get the S-expression. RlispRead is a version of Read, using a special ScanTable, RlispReadScanTable!*.

The function Scan also sets SEMIC!* to '!; or '!$ if CURSYM!* is detected to be '!*SEMICOL!* (the internal name for '!; and "!$). This controls the RLisp echo/no-echo capability. Finally, if the renamed token is 'COMMENT then characters are ReadCh'd until a '!; or '!$.

6.2.3. Precedences

To set up precedences, it is often helpful to set up a precedence matrix of the operators involved. If any operator has one "precedence" with respect to one particular operator and another "precedence" with respect to some other, it is sometimes not possible to run the parser with just numbered precedences for the operators without introducing ambiguities. If this is the case, replace the number RP by the operator RP and test with something like:

IF RP *GREATER* OP . . .

GREATER may check in the precedence matrix. An example in which such a scheme might be used is the case for which Algol uses ":" both as a label marker and as an index separator (although in this case there is no need for the change above). It is also a good policy to have even numbers for right precedences and odd numbers for left precedences (or vice versa).

6.2.4. Special Cases of 0 <-0 and 0 0

If . . .0 0. . . is a legal case (i.e. F A may translate to (F A)), ERROR OP MISSING is replaced by:

Y:=REPCOM(Y,RDRIGHT(99,OP)); GO TO RDLEFT;

The value 99 is chosen in order to have the first object (F) behave as a right operator with maximum precedence. If . . .0 <-0. . . is legal for some combinations of operators, replace ERROR ARG MISSING by something equivalent to the illegal RLisp statement:

```
IF ISOPOP(OP,RP,Y)
        THEN <<OP:=Y;
               Y:=(something else, i.e. NIL);
               GOTO RDLEFT>>
      ELSE ERROR ARG MISSING;
```

ISOPOP is supposed to return T if the present situation is legal.

6.2.5. Parenthesized Expressions

(a) is to be translated to a.

E.g.
BEGIN a END translates to (PROG a).

Define "(" and BEGIN as right operators with low precedences (2 and -2 respectively). Also define ")" and END as binary operators with matching left precedences (1 and -3 respectively). The construction functions for "(" and BEGIN are then something like: [See pu:RLISP-PARSER.RED for exact details on ParseBEGIN]

```
BEGIN     (X);PROG2(OP:=SCAN();MAKEPROG(X));
"("       (X);PROG2(IF OP=') THEN OP:=SCAN()
                                 ELSE ERROR, x);
```

Note that the construction functions in these cases have to read the next token; that is the effect of ")" closing the last "(" and not all earlier "("'s. This is also an example of binary operators declared only for the purpose of having a left precedence.

6.2.6. Binary Operators in General

As almost all binary operators have a construction function like

LIST(OP,X,Y);

it is assumed to be of that kind if no other is given. If OP is a binary operator, then "a OP b OP c" is interpreted as "(a OP b) OP c" only if OP's LP is less than OP's RP.

Example:

A + B + C translates to (A + B) + C
because +'RP = 20 and +'LP = 19

A ^ B ^ C translates to A ^ (B ^ C)
because ^'RP = 20 and ^'LP = 21

If you want some operators to translate to n-ary expressions, you have to define a proper construction function for that operator.

Example:

```
PLUS   (X,Y); IF CAR(X) = 'PLUS THEN NCONC(X,LIST(Y))
                                ELSE LIST('PLUS,X,Y);
```

By defining "," and ";" as ordinary binary operators, the parser automatically takes care of constructions like . . .e,e,e,e,e. . . and . . . stm;stm;stm;stm;. . . It is then up to some other operators to remove the "," or the ";" from the parsed result.

6.2.7. Assigning Precedences to Key Words

If you want some operators to have control immediately, insert

IF RP = NIL THEN RETURN Y ELSE

as the very first test in RDRIGHT and set the right precedence of those to NIL. This is sometimes useful for key-word expressions. If entering a construction function of such an operator, X is the token immediately after the operator. E.g.: We want to parse PROCEDURE EQ(X,Y); . . . Define PROCEDURE as a right operator with NIL as precedence. The construction function for PROCEDURE can always call the parser and set the rest of the expression. Note that if PROCEDURE was not defined as above, the parser would misunderstand the expression in the case of EQ as declared as a binary operator.

6.2.8. Error Handling

For the present, if an error occurs a message is printed but no attempt is made to correct or handle the error. Mostly the parser goes wild for a while (until a left precedence less than current right precedence is found) and then goes on as usual.

6.2.9. The Parser Program for the RLISP Language

SCAN();

The purpose of this function is to read the next token from the input stream. It uses the general purpose table driven token scanner described in Chapter INPUT, with a specially set up ReadTable, RlispScanTable!*. As RLisp has multiple identifiers for the same operators, Scan uses the following translation table:

=	EQUAL	>=	GEQ
+	PLUS	>	GREATERP
-	DIFFERENCE	<=	LEQ
/	QUOTIENT	<	LESSP
.	CONS	*	TIMES
:=	SETQ	**	EXPT

In these cases, Scan returns the right hand side of the table values. Also, two special cases are taken care of in Scan:

a. ' is the QUOTE mark. If a parenthesized expression follows ' then the syntax within the parenthesis is that of Lisp, using a special scan table, RlispReadScanTable!*. The only major difference from ordinary Lisp is that ! is required for all special characters.

b. ! in RLisp means actually two things:

   i. the following symbol is not treated as a special symbol (but belongs to the print name of the atom in process);

   ii. the atom created cannot be an operator.

Example: !(in the text behaves as the atom "(".

To signal to the parser that this is the case, the flag variable ESCAPEFL must be set to T if this situation occurs.

6.2.10. Defining Operators

To define operators use:

DEFINEROP(op,p{,stm});
  For right or prefix operators.

DEFINEBOP(op,lp,rp{,stm});
  For binary operators.

These use the VALUE of DEFPREFIX and DEFINFIX to store the precedences and construction functions. The default is set for RLisp, to be 'RLISPPREFIX and 'RLISPINFIX. The same identifier can be defined both as the right and binary operator. The context defines which one applies.

Stm is the construction function. If stm is omitted, the common defaults are used:

LIST(OP,x) prefix case, x is parsed expression following, x=RDRIGHT(p,SCAN()).

LIST(OP,x,y) binary case, x is previously parsed expression, y is expression following, y=RDRIGHT(rp,SCAN()).

If stm is an id, it is assumed to be a procedure of one or two arguments, for "x" or "x,y". If it is an expression, it is embedded as (LAMBDA(X) stm) or (LAMBDA(X Y) stm), and should refer to X and Y, as needed.

Also remember that the free variable OP holds the last token (normally the binary operator which stopped the parser). If "p" or "rp" is NIL, RDRIGHT is not called by default, so that only SCAN() (the next token) is passed.

For example,

```
DEFINEBOP('DIFFERENCE,17,18);
        % Most common case, left associative, stm=LIST(OP,x,y);

DEFINEBOP('CONS,23,21);
        % Right Associative, default stm=LIST(OP,x,y)

DEFINEBOP('AND,11,12,ParseAND);
        % Left Associative, special function
    PROCEDURE ParseAND(X,Y);
       NARY('AND,X,Y);

DEFINEBOP('SETQ,7,6,ParseSETQ);
        % Right Associative, Special Function
    PROCEDURE ParseSETQ(LHS,RHS);
      LIST(IF ATOM LHS THEN 'SETQ ELSE 'SETF, LHS, RHS);

DEFINEROP('MINUS,26);    % default C-fn, just (list OP arg)

DEFINEROP('PLUS,26,ParsePLUS1); %

DEFINEROP('GO,NIL,ParseGO );
       % Special Function, DO NOT use default PARSE ahead
    PROCEDURE ParseGO X;    X is now JUST next-token
      IF X EQ 'TO THEN LIST('GO,PARSEO(6,T))
               % Explicit Parse ahead
           ELSE <<OP := SCAN(); % get Next Token
                  LIST('GO,X)>>;

DEFINEROP('GOTO,NIL,ParseGOTO );
```

```
      % Suppress Parse Ahead, just pass NextToken
  PROCEDURE ParseGOTO X;
    <<OP := SCAN();
      LIST('GO,X)>>;
```

6.3. The MINI Translator Writing System

Note that MINI is now autoloading.

6.3.1. A Brief Guide to MINI

The following is a brief introduction to MINI, the reader is referred to [Marti 79] for a more detailed discussion of the META/RLISP operators, which are very similar to those of MINI.

The MINI system reads in a definition of a translator, using a BNF-like form. This is processed by MINI into a set of Lisp functions, one for each production, which make calls on each other, and a set of support routines that recognize a variety of simple constructs. MINI uses a stack to perform parsing, and the user can access sub-trees already on the stack, replacing them by other trees built from these sub-trees. The primitive functions that recognize ids, integers, etc. each place their recognized token on this stack.

For example,

```
 FOO: ID '!- ID +(PLUS2 #2 #1) ;
```

defines a rule FOO, which recognizes two identifiers separated by a minus sign (each ID pushes the recognized identifier onto the stack). The last expression replaces the top 2 elements on the stack (#2 pops the first ID pushed onto the stack, while #1 pops the other) with a Lisp statement.

(Id): boolean expr

See if current token is an identifier and not a keyword. If it is, then push onto the stack and fetch the next token.

(AnyId): boolean expr

See if current token is an id whether or not it is a key word.

(AnyTok): boolean expr

Always succeeds by pushing the current token onto the stack.

(Num): boolean expr

Tests to see if the current token is a number, if so it pushes the number onto the stack and fetches the next token.

(Str): boolean expr

Same as Num, except for strings.

Specification of a parser using MINI consists of defining the syntax with BNF-like rules and semantics with Lisp expressions. The following is a brief list of the operators:

'	Used to designate a terminal symbol (i.e. 'WHILE, 'DO, '!=).
Identifier	Specifies a nonterminal.
( )	Used for grouping (i.e. (FOO BAR) requires rule FOO to parse followed immediately by BAR).
< >	Optional parse, if it fails then continue (i.e. <FOO> tries to parse FOO).
/	Optional rules (i.e. FOO / BAR allows either FOO or BAR to parse, with FOO tested first).
STMT*	Parse any number of STMT.
STMT[ANYTOKEN]*	Parse any number of STMT separated by ANYTOKEN, create a list and push onto the stack (i.e. ID[,]* parses a number of identifiers separated by commas, like in an argument list).
##n	Refer to the nth stack location (n must be an integer).
#n	Pop the nth stack location (n must be an integer).
+(STMT)	Push the unevaluated (STMT) onto the stack.
.(SEXPR)	Evaluate the SEXPR and ignore the result.
=(SEXPR)	Evaluate the SEXPR and test if result non-NIL.
+.(SEXPR)	Evaluate the SEXPR and push the result on the stack.
@ANYTOKEN	Specifies a statement terminator; used in the error recovery mechanism to search for the occurrence of errors.
@@ANYTOKEN	Grammar terminator; also stops scan, but if encountered in error-recovery,

terminates grammar.

6.3.2. Pattern Matching Rules

In addition to the BNF-like rules that define procedures with 0 arguments and which scan tokens by calls on NEXT!-TOK() and operate on the stack, MINI also includes a simple TREE pattern matcher and syntax to define PatternProcedures that accept and return a single argument, trying a series of patterns until one succeeds.

```
E.g.        template     ->  replacement

PATTERN = (PLUS2 &1 0) -> &1,
          (PLUS2 &1 &1) -> (LIST 'TIMES2 2 &1),
          &1            -> &1;
```

defines a pattern with 3 rules. &n is used to indicate a matched sub-tree in both the template and replacement. A repeated &n, as in the second rule, requires Equal sub-trees.

6.3.3. A Small Example

```
% A simple demo of MINI, to produce a LIST-NOTATION reader.
% INVOKE 'LSPLOOP reads S-expressions, separated by ;

mini 'lsploop;                  % Invoke MINI, give name of ROOT
                                % Comments can appear anywhere,
                                % prefix by % to end-of-line
lsploop:lsp* @@# ;              % @@# is GRAMMAR terminator
                                %  like '# but stops TOKEN SCAN
lsp:    sexp @;                 % @; is RULE terminator, like ';
        .(print #1)             %  but stops SCAN, to print
        .(next!-tok) ;          %  so call NEXT!-TOK() explicitly
sexp:   id / num / str / '( dotexp ') ;
dotexp: sexp* < '. sexp +.(attach #2 #1)  > ;
fin

symbolic procedure attach(x,y);
<<for each z in reverse x do y:=z . y; y>>;
```

6.3.4. Loading Mini

MINI is loaded from PH: using LOAD MINI;.

6.3.5. Running Mini

A MINI grammar is run by calling Invoke rootname;. This installs appropriate Key Words (stored on the property list of rootname), and start the grammar by calling the Rootname as first procedure.

6.3.6. MINI Error messages and Error Recovery

If MINI detects a non-fatal error, a message be printed, and the current token and stack is shown. MINI then calls NEXT!-TOK() repeatedly until either a statement terminator (@ANYTOKEN) or grammar terminator (@ANYTOKEN) is seen. If a grammar terminator, the grammar is exited; otherwise parsing resumes from the ROOT.

[??? Interaction with BREAK loop rather poor at the moment ???]

6.3.7. MINI Self-Definition

```
% The following is the definition of the MINI meta system in terms of
% itself.  Some support procedures are needed, and exist in a
% separate file.
% To define a grammar, call the procedure MINI with the argument
% being the root rule name.  Then when the grammar is defined it may
% be called by using INVOKE root rule name.

%   The following is the MINI Meta self definition.

MINI 'RUL;

%   Define the diphthongs to be used in the grammar.
DIP: !#!#, !-!>, !+!., !@!@ ;

%   The root rule is called RUL.
RUL: ('DIP ': ANYTOK[,]* .(DIPBLD #1) '; /
     (ID  .(SETQ !#LABLIST!# NIL)
       ( ': ALT            +(DE #2 NIL #1) @; /
         '= PRUL[,]* @;    .(RULE!-DEFINE '(PUT(QUOTE ##2)(QUOTE RB)
                             (QUOTE #1)))
                           +(DE ##1 (A)
                             (RBMATCH A (GET (QUOTE #1) (QUOTE RB))
                                                        NIL)))
       .(RULE!-DEFINE #1) .(NEXT!-TOK) ))* @@FIN ;

%   An alternative is a sequence of statements separated by /'s;
ALT: SEQ < '/ ALT +(OR #2 #1) >;

%   A sequence is a list of items that must be matched.
```

```
SEQ: REP < SEQ +(AND #2 (FAIL!-NOT #1)) >;

%   A repetition may be 0 or more single items (*) or 0 or more items
%    separated by any token (ID[,]* parses a list of ID's separated
%    by ,'s.
REP: ONE
      <'[ (ID +(#1) /
           '' ANYKEY +(EQTOK!-NEXT (QUOTE #1)) /
     ANYKEY +(EQTOK!-NEXT (QUOTE #1))) '] +(AND #2 #1) '* BLD!-EXPR /
        '* BLD!-EXPR>;

%   Create an sexpression to build a repetition.
BLD!-EXPR: +(PROG (X) (SETQ X (STK!-LENGTH))
                   $1 (COND (#1 (GO $1)))
                      (BUILD!-REPEAT X)
                      (RETURN T));

ANYKEY: ANYTOK .(ADDKEY ##1) ;  % Add a new KEY

%   One defines a single item.
ONE: '' ANYKEY  +(EQTOK!-NEXT (QUOTE #1)) /
     '@ ANYKEY  .(ADDRTERM ##1)  +(EQTOK (QUOTE #1)) /
     '@@ ANYKEY .(ADDGTERM ##1)  +(EQTOK (QUOTE #1)) /
     '+ UNLBLD  +(PUSH #1) /
     '. EVLBLD  +(PROGN #1 T) /
     '= EVLBLD  /
     '< ALT '>  +(PROGN #1 T) /
     '( ALT ')  /
     '+. EVLBLD +(PUSH #1) /
     ID         +(#1) ;

%   This rule defines an un evaled list.  It builds a list with
%   everything quoted.
UNLBLD: '( UNLBLD ('. UNLBLD ') +(CONS #2 #1) /
                    UNLBLD* ') +(LIST . (#2 . #1)) /
                   ') +(LIST . #1)) /
        LBLD    /
        ID      +(QUOTE #1) ;

%   EVLBLD builds a list of evaled items.
EVLBLD: '( EVLBLD ('. EVLBLD ') +(CONS #2 #1) /
                    EVLBLD* ') +(#2 . #1) /
                   ') ) /
        LBLD /
        ID      ;
```

```
LBLD: '# NUM    +(EXTRACT #1) /
      '## NUM   +(REF #1) /
      '$ NUM    +(GENLAB #1) /
      '& NUM    +(CADR (ASSOC #1 (CAR VARLIST))) /
      NUM       /
      STR       /
      '' ('( UNLBLD* ') +(LIST . #1) /
           ANYTOK +(QUOTE #1));

%   Defines the pattern matching rules (PATTERN -> BODY).
PRUL: .(SETQ INDEXLIST!* NIL)
      PAT '-> (EVLBLD)*
             +(LAMBDA (VARLIST T1 T2 T3) (AND . #1))
             .(SETQ PNAM (GENSYM))
             .(RULE!-DEFINE (LIST 'PUTD (LIST 'QUOTE PNAM)
                '(QUOTE EXPR) (LIST 'QUOTE #1)))
             +.(CONS #1 PNAM);

%   Defines a pattern.
%   We now allow the . operator to be the next to last in a ().
PAT: '& ('< PSIMP[/]* '> NUM
             +.(PROGN (SETQ INDEXLIST!* (CONS ##1 INDEXLIST!*))
                  (LIST '!& #2 #1) ) /
             NUM
               +.(COND ((MEMQ ##1 INDEXLIST!*)
                         (LIST '!& '!& #1))
                  (T (PROGN (SETQ INDEXLIST!* (CONS ##1 INDEXLIST!*))
                         (LIST '!& #1)))) )
        / ID
        / '!( PAT* <'. PAT +.(APPEND #2 #1)> '!)
        / '' ANYTOK
        / STR
        / NUM ;

%   Defines the primitives in a pattern.
PSIMP: ID / NUM / '( PSIMP* ') / '' ANYTOK;

%   The grammar terminator.
FIN
```

6.3.8. The Construction of MINI

MINI is actually described in terms of a support package for any MINI-generated parser and a self-description of MINI. The useful files (on PU: and PL:) are as follows:

MINI.MIN	The self definition of MINI in MINI.
MINI.SL	A Standard LISP version of MINI.MIN, translated by MINI itself.
MINI.RED	The support RLISP for MINI.
MINI-PATCH.RED and MINI.FIX	Some additions being tested.
MINI.LAP	The precompiled LAP file. Use LOAD MINI.
MINI-LAP-BUILD.CTL	A batch file that builds PL:MINI.LAP from the above files.
MINI-SELF-BUILD.CTL	A batch file that builds the MINI.SL file by loading and translating MINI.MIN.

6.3.9. History of MINI Development

The MINI Translator Writing System was developed in two steps. The first was the enhancement of the META/RLISP [Marti 79] system with the definition of pattern matching primitives to aid in describing and performing tree-to-tree transformations. META/RLISP is very proficient at translating an input programming language into LISP or LISP-like trees, but did not have a good method for manipulating the trees nor for direct generation of target machine code. PMETA (as it was initially called) [Kessler 79] solved these problems and created a very good environment for the development of compilers. In fact, the PMETA enhancements have been fully integrated into META/RLISP.

The second step was the elimination of META/RLISP and the development of a smaller, faster system (MINI). Since META/RLISP was designed to provide maximum flexibility and full generality, the parsers that is creates are large and slow. One of its most significant problems is that it uses its own single character driven LISP functions for token scanning and recognition. Elimination of this overhead has produced a faster translator. MINI uses the hand coded scanner in the underlying RLisp. The other main aspect of MINI was the elimination of various META/RLISP features to decrease the size of the system (also decreasing the flexibility, but MINI has been successful for the various purposes in COG). MINI is now small enough to run on small LISP systems (as long as a token scanner is provided). The META/RLISP features that MINI has changed or eliminated include the following:

a. The ability to backup the parser state upon failure is supported in META/RLISP. However, by modifying a grammar definition, the need for backup can be mostly avoided and was therefore eliminated from MINI.

b. META/RLISP has extensive mechanisms to allow arbitrary length diphthongs. MINI only supports two character diphthongs, declared prior to their use.

c. The target machine language and error specification operators are not supported because they can be implemented with support routines.

d. RLISP subsyntax for specification of semantic operations is not supported (only LISP is provided).

Although MINI lacks many of the features of META/RLISP, it still has been quite sufficient for a variety of languages.

6.4. BNF Description of RLisp Using MINI

The following formal scheme for the translation of RLisp syntax to Lisp syntax is presented to eliminate misinterpretation of the definitions. We have used the above MINI syntactic form since it is close enough to BNF and has also been checked mechanically.

Recall that the transformation scheme produces an S-expression corresponding to the input RLisp expression. A rule has a name by which it is known and is defined by what follows the meta symbol :. Each rule of the set consists of one or more "alternatives" separated by the meta symbol /, being the different ways in which the rule is matched by source text. Each rule ends with a ;. Each alternative is composed of a "recognizer" and a "generator". The "generator" is a MINI + expression which builds an S-expression from constants and elements loaded on the stack. The result is then loaded on the stack. The #n and ##n refer to elements loaded by MINI primitives or other rules. The "generator" is thus a template into which previously generated items are substituted. Recall that terminals in both recognizer and generator are quoted with a ' mark.

This RLisp/SYSLisp syntax is based on a series of META and MINI definitions, started by R. Loos in 1970, continued by M. Griss, R. Kessler and A. Wang.

[??? This MINI.RLISP grammar is a bit out of date ???]

[??? Need to confirm for latest RLISP ???]

```
mini 'rlisp;

dip: !: , !<!< , !>!> , !:!= , !*!* , !<!= , !>!= , !' , !#!# ;

termin: '; / '$ ;               % $ used to not echo result
rtermin: @; / @$ ;

rlisp: ( cmds rtermin  .(next!-tok) )* ; % Note explicit Scan

cmds:  procdef / rexpr ;

%------ Procedure definition:

procdef: emodeproc (ftype procs/ procs) /
         ftype procs / procs ;

ftype:   'fexpr .(setq FTYPE!* 'fexpr) /  % function type
         'macro .(setq FTYPE!* 'macro) /
         'smacro .(setq FTYPE!* 'smacro) /
         'nmacro .(setq FTYPE!* 'nmacro) /
```

```
          ('expr / =T) .(setq FTYPE!* 'expr) ;


emodeproc: 'syslsp .(setq EMODE!* 'syslsp)/
           ('lisp/'symbolic/=T) .(setq EMODE!* 'symbolic) ;


procs: 'procedure id proctail
           +(putd (quote #2) (quote FTYPE!* ) #1) ;

proctail: '( id[,]* ')  termin  rexpr +(quote (lambda #2 #1)) /
           termin  rexpr +(quote (lambda nil #1)) /
          id  termin  rexpr +(quote (lambda (#2) #1)) ;

%------ Rexpr definition:

rexpr: disjunction ;

disjunction: conjunction (disjunctail / =T) ;

disjunctail: ('or conjunction ('or conjunction)*)
              +.(cons 'or  (cons #3 (cons #2 #1))) ;

conjunction: negation (conjunctail / =T) ;

conjunctail: ('and negation ('and negation)*)
             +.(cons (quote and) (cons #3 (cons #2 #1))) ;

negation: 'not negation +(null #1) /
          'null negation +(null #1) /
          relation ;

relation: term reltail ;

reltail: relop term +(#2 #2 #1) / =T ;

term: ('- factor +(minus #1) / factor) termtail ;

termtail: (plusop factor +(#2 #2 #1) termtail) / =T ;

factor: powerexpr factortail ;

factortail: (timop powerexpr +(#2 #2 #1) factortail) / =T ;

powerexpr: dotexpr powtail ;
```

```
powtail: ('** dotexpr +(expt #2 #1) powtail) / =T ;

dotexpr: primary dottail ;

dottail: ('. primary +(cons #2 #1) dottail) / =T ;

primary: ifstate / groupstate / beginstate /
         whilestate / repeatstate / forstmts /
         definestate / onoffstate / lambdastate /
         ('( rexpr ') ) /
         ('' (lists / id / num) +(quote #1)) /
         id primtail / num ;

primtail:(':= rexpr +(setq #2 #1)) /
         (': labstmts ) /
         '( actualst / (primary +(#2 #1)) / =T ;

lists: '( (elements)* ') ;

elements: lists / id / num ;

%------ If statement:

ifstate: 'if rexpr 'then rexpr elserexpr
              +(cond (#3 #2) (T #1)) ;

elserexpr: 'else rexpr / =T +nil ;

%------ While statement:

whilestate: 'while rexpr 'do rexpr
             +(while #2 #1) ;

%----- Repeat statement:

repeatstate: 'repeat rexpr 'until rexpr
              +(repeat #2 #1) ;

%---- For statement:

forstmts: 'for fortail ;

fortail: ('each foreachstate) / forstate ;

foreachstate: id inoron rexpr actchoice rexpr
              +(foreach #5 #4 #3 #2 #1) ;
inoron: ('in +in / 'on +on) ;

actchoice: ('do +do / 'collect +collect / 'conc +conc) ;

forstate: id ':= rexpr loops ;

loops: (': rexpr types rexpr
       +(for #5 (#4 1 #3) #2 #1) ) /
       ('step rexpr 'until rexpr types rexpr
       +(for #6 (#5 #4 #3) #2 #1) ) ;

types: ('do +do / 'sum +sum / 'product +product) ;

%----- Function call parameter list:

actualst: ') +(#1) / rexpr[,]* ') +.(cons #2 #1) ;

%------ Compound group statement:

groupstate: '<< rexprlist '>> +.(cons (quote progn) #1) ;

%------ Compound begin-end statement:

beginstate: 'begin blockbody 'end ;

blockbody: decllist blockstates
            +.(cons (quote prog) (cons #2 #1)) ;

decllist: (decls[;]* +.(flatten #1)) / (=T +nil) ;

decls: ('integer  / 'scalar) id[,]* ;

blockstates: labstmts[;]* ;

labstmts: ('return rexpr +(return #1)) /
          (('goto / 'go 'to) id +(go #1)) /
          ('if rexpr 'then labstmts blkelse
              +(cond (#3 #2) (T #1))) /
          rexpr ;

blkelse: 'else labstmts / =T +nil ;

rexprlist: rexpr [;]* ;

lambdastate: 'lambda lamtail ;
```

```
lamtail: '( id[,]* ')  termin  rexpr +(lambda #2 #1) /
          termin  rexpr +(lambda nil #1) /
         id  termin  rexpr +(lambda (#2) #1) ;

%------ Define statement: (id and value are put onto table
%       named DEFNTAB:

definestate: 'define delist +.(cons (quote progn) #1) ;

delist: (id '= rexpr +(put (quote #2)  (quote defntab)
              (quote #1)))[,]* ;

%------ On or off statement:

onoffstate: ('on +T / 'off +nil) switchlists ;

switchlists: 'defn +(set '!*defn #1) ;

timop: ('* +times / '/ +quotient) ;

plusop: ('+ +plus2 / '- +difference) ;

relop: ('< +lessp / '<= +lep / '= +equal /
          '>= +gep / '> +greaterp) ;


FIN
```

CHAPTER 7
INDEX OF CONCEPTS

The following is an alphabetical list of concepts, with the page on which they are discussed.

CHAPTER 8
INDEX OF FUNCTIONS

The following is an alphabetical list of the PSL functions, with the page on which they are defined.

CHAPTER 9
INDEX OF GLOBALS AND SWITCHES

The following is an alphabetical list of the PSL global variables, with the page on which they are defined.

The Portable Standard LISP Users Manual

Part 3: System Dependent Information

by
The Utah Symbolic Computation Group

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Version 3.2: 16 March 1984

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 the DEC-20, VAX, and MC68000.

Work supported in part by the Hewlett Packard Company, the International Business Machines Corporation and the National Science Foundation under Grant Numbers MCS80-07034 and MCS82-04247.

PREFACE

Part 3 of The Portable Standard Lisp User's Manual contains system-dependent information. It includes chapters about PSL on the Dec-20, the Vax running Berkeley Unix, and the Apollo. Each chapter provides information on the PSL executables, the PSL file structure, and how to get started using PSL on the particular system. Documentation on the interface between PSL and each operating system is also given.

These chapters were adated from chapters in the January 19, 1983 version of the manual. Contributions were made by Russ Fish, Will Galway, Robert Kessler, Bobbie Othmer, and John W. Peterson.

TABLE OF CONTENTS

CHAPTER 1
PSL ON THE DECSYSTEM-20

1.1. Purpose of This Chapter

This chapter is for beginning users of PSL on the DecSystem-20. It begins with descriptions of how to set up various logical device definitions required by PSL and how to run PSL. The chapter continues with a description of the file structure associated with PSL, an example on use of the PSL system, and miscellaneous hints and reminders. It concludes with a discussion of interfacing to the Tops-20 operating system.

1.2. Logical Device Names for DecSystem-20 PSL

In order to make all DecSystem-20 versions of PSL compatible, PSL makes heavy use of logical names for access to various directories. Also, as each release of PSL is prepared, we may find it convenient to change the names and number of subdirectories. The DecSystem-20 allows these logical names to be used as if they were directory names. These definitions are edited at installation time to reflect local usage, and stored in a file whose name is something like "logical-names.xxx". This file is normally placed on an appropriate directory like ss:<psl> at Utah. Contact your installer to find its exact location and name. It is absolutely essential that TAKE <PSL>LOGICAL-NAMES.CMD be inserted in your LOGIN.CMD file, or COMAND.CMD file (if you frequently push to new execs), or executed at EXEC level before using PSL. PSL is written to rely on these logical device definitions in place of "hard-coded" directory names. PSL also uses TOPS-20 search paths; for example, "PL:" is the directory (or search list) on which PSL looks for Lap and Fasl files of the form "xxxx.b".

The logical name "PSL:" is defined to be the directory on which the PSL executables reside. Thus "PSL:PSL.EXE" should start PSL executing. There should usually be a PSL:BARE-PSL.EXE, PSL:PSL.EXE, PSL:RLISP.EXE and PSL:PSLCOMP.exe. These executables are described in the next section.

1.3. PSL Executables

After defining the device names, type either PSL:RLISP or PSL:PSL to the at-sign prompt, @. A welcome message indicates the nature of the system running, usually with a date and version number. This information may be useful in describing problems.

BARE-PSL.EXE is a "bare" PSL using Lisp (i.e. parenthesis) syntax. This is a small core-image and is ideal for simple Lisp execution. It also includes a resident Fasl, so additional modules can be loaded. BARE-PSL is used as the base for creating all of the other executables.

Certain modules are not present in the "kernel" or "bare-psl" system, but can be loaded as options. Optional modules can be loaded by executing

```
(LOAD modulename)
```

PSL.EXE is an installation dependent "enhanced" BARE-PSL. Your system installer has determined which modules are to be included in your base system. At Utah, the only additional modules in PSL.EXE are the ones necessary to permit the the reading of an initialization file upon startup (see Section 1.6 for details of the initialization files). You can discover which modules are loaded into your system by examining the variable OPTIONS!* upon startup of PSL.EXE.

RLISP.EXE is also an "enhanced" BARE-PSL with additional modules loaded; typically including the compiler, the RLisp parser and the init-file module. For more information about RLisp see the RLISP chapter in Part 2 of the PSL Reference Manual.

PSLCOMP.EXE is an executable that permits compilation of Lisp, RLisp or build files into loadable binary modules. PSLCOMP.EXE is built from BARE-PSL.EXE and includes the following modules: pslcomp-main, init-file, objects, common, strings, pathnames, fast-vector and nstruct. The pslcomp-main module implements a function that will read the information on the PSLCOMP.EXE executation line, interpret that as the name of a file and compile the file. When the file name does not include an extension, PSLCOMP first looks for <filename>.BUILD (an RLISP build file that usually reads in a number of files to make into a single loadable module), <filename>.SL (a PSL Lisp syntax source file) and finally <filename>.RED (an RLISP source file). The file will be compiled and a binary file will be created in the SAME DIRECTORY as the input file. Therefore, if there is a file FOO.SL in the XX: directory:

```
@PSL:PSLCOMP XX:FOO

% Will compile xx:foo.sl into xx:foo.b
```

NMODE.EXE is the NMODE text editor and PSL environment (see the NMODE reference manual for further details).

It is assumed by PSL and RLisp that file names be of the form "*.sl" or "*.lsp" for Lisp files, "*.red" for RLisp files, "*.b" for Fasl files, and "*.lap" for Lap files.

1.4. File Structure on the DecSystem-20

At Utah, the DecSystem-20 is used as the primary system for source code for all of the PSL supported machines. Thus a hierarchical directory structure is utilized to maintain the various sources. At the top of the tree is the PSL: directory. This directory contains the executables, the logical-names definition file, a couple of news files and a set of NMODE description files. Under the PSL: directory is a set of directories that generally contain the target machine independent files for a particular area of the system. Then under that directory are a set of target machine dependent directories. This scheme is reflected in the use of the logical names. The generic logical name consists of three parts:

```
p<target machine abbreviation><directory abbreviation>
```

The 'p' denotes that all of the directories are associated with PSL. The <target machine abbreviation> is a code for the specific target machine as follows:

''	An empty <target machine abbreviation> signifies the target machine independent directory.
'20'	DecSystem-20.
'68'	68000 (Not available in all cases).
'A'	Apollo (in some cases, this is a subdirectory of the 68000 directory).
'CR'	Cray.
'HP'	HP9836 (in some cases, this is a subdirectory of the 68000 directory).
'V'	Vax.

The directory abbreviations are as follows:

'C'	Compiler.
'Dist'	Distribution directory, as an aid to the installer.
'D'	Documentation.
'G'	Novak's GLISP (there are no subdirectories).
'H'	Obsolete Help directory (there are no subdirectories). It now contains only a few brief files describing some of the modules, not described in the manual.

'K'	Kernel specific files. NOTE: due to historical reasons, the K is NOT included when referencing machine specific kernel files (e.g. the DecSystem-20 kernel specific files are p20: and not p20k:).
'L'	Lap (binary Fasl files). In some installations (Utah in particular), a subdirectory PLN: (<psl.lap.new>) contains the latest versions of loadable modules. This has been arranged so that users who desire a stable system can access the released modules on PL:, while those wishing to use experimental versions can access PLN: (there are no other subdirectories).
'LPT'	The various chapters of this manual in line printer format (has overprinting and underlining) (there are no other subdirectories).
'N'	The NMODE text editor sources. It has one subdirectory PNB:, which contains the various binary files that make up NMODE.
'NK'	NonKernel specific files. Those files that are not included in the basic kernel, but are 'LOAD'ed in to make BARE-PSL.EXE.
'T'	Test directory, contains a set of timing and test files.
'U'	Sources for most utilities, useful as examples of PSL and RLISP code, and for customization.
'W'	Sources for NMODE window management.

The following are examples of the usage of the above convention for finding directories:

pc:	Machine independent compiler sources.
pv:	Vax specific kernel sources.
p20u:	DecSystem-20 specific utilities.

1.5. Sample Session with DecSystem-20 PSL

The following is a transcript of running PSL on the DecSystem-20.

```
@psl:psl
Extended 20-PSL 3.1, 15-Jun-83

1 Lisp> % Notice the numbered prompt.
1 Lisp> % Comments begin with "%" and do not change the prompt
1 Lisp> % number.
1 Lisp> (Setq Z '(1 2 3))  % Make an assignment for Z.
(1 2 3)
2 Lisp> (Cdr Z)            % Notice the change in prompt number.
(2 3)
3 Lisp> (De Count (L)      % Count counts the number or elements
3 Lisp>    (Cond ((Null L) 0)  % in a list L.
3 Lisp>          (T (Add1 (Count (Cdr L))))))
COUNT
4 Lisp> (Count Z)          % Call Count on Z.
3
5 Lisp> (Tr Count)  % Trace the recursive execution of "Count".
(COUNT)
6 Lisp>             % A call on "Count" now shows the value of
6 Lisp>             % "Count" and of its arguments each time
6 Lisp> (Count Z)   % it is called.
COUNT being entered
   L:   (1 2 3)
  COUNT (level 2) being entered
     L: (2 3)
    COUNT (level 3) being entered
       L:       (3)
      COUNT (level 4) being entered
         L:     NIL
      COUNT (level 4) = 0
    COUNT (level 3) = 1
  COUNT (level 2) = 2
COUNT = 3
3
7 Lisp> (Untr Count)
NIL
8 Lisp> (Count 'A)  % This generates an error causing the break
                           % loop to be entered.
***** An attempt was made to do CDR on 'A', which is not a pair
Break loop
9 Lisp break>> ?
BREAK():{Error,return-value}
----------------------------
This is a Read-Eval-Print loop, similar to the top level loop,
except that the following IDs at the top level cause functions to
be called rather than being evaluated:
```

```
?         Print this message, listing active Break IDs
T         Print stack backtrace
Q         Exit break loop back to ErrorSet
A         Abort to top level, i.e. restart PSL
C         Return last value to the ContinuableError call
R         Reevaluate ErrorForm!* and return
M         Display ErrorForm!* as the "message"
E         Invoke a simple structure editor on ErrorForm!*
                 (For more information do Help Editor.)
I         Show a trace of any interpreted functions

See the manual for details on the Backtrace, and how ErrorForm!* is
set.  The Break Loop attempts to use the same TopLoopRead!* etc, as
the calling top loop, just expanding the PromptString!*.
NIL
10 Lisp break>>          % Get a Trace-Back of the
10 Lisp break>> I        % interpreted functions.
Backtrace, including interpreter functions, from top of stack:
CDR COUNT ADD1 COND COUNT
NIL
11 Lisp break>> Q        % To exit the Break Loop.
12 Lisp>                 % Load in a file, showing its execution.
12 Lisp>                 % The file contains the following:
12 Lisp>                 % (Setq X (Cons 'A (Cons 'B Nil)))
12 Lisp>                 % (Count X)
12 Lisp>                 % (Reverse X)
12 Lisp> (Dskin "small-file.sl")
(A B)
2
(B A)
NIL
13 Lisp> (Quit)
@continue
"Continued"
14 Lisp> ^C
@start

15 Lisp> (Quit)
```

1.6. Init files

Init files are available to make it easier for the user to customize PSL to his/her own needs. When PSL, RLISP, or PSLCOMP is executed, if a file PSL.INIT, RLISP.INIT, or PSLCOMP.INIT is on the home directory, it will be read and evaluated. Currently all init files must be written in Lisp syntax. They may use Faslin or Load as needed. More details on init files are available in Part 1 of this manual.

1.7. Error and Warning Messages

Many functions detect and signal appropriate errors (see the chapter on error handling in Part 1 of this manual for details); 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.

Sometimes one sees a prompt of the form:

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

This means you have tried to define a function with the same name as a function used by the PSL system. A Y, N, YES, NO, or B response is required. B starts a break loop. After quitting the break loop, answer Y, N, Yes, or No to the query. See the definition of YesP in the input/output chapter in Part 1 of this manual. An affirmative response is extremely dangerous and should be given only if you are a system expert. Usually this means that your function must be given a different name.

A common warning message is

*** Function "FOO" has been redefined

If this occurs without the query above, you are redefining your own function. This happens normally if you read a file, edit it, and read it in again.

1.8. Reporting Errors and Misfeatures

Send bug MAIL to PSL-BUGS@UTAH-20. The message will be distributed to a list of users concerned with bugs and maintenance, and a copy will be kept in <PSL>BUGS-MISSFEATURES.TXT at UTAH-20.

(Bug): `undefined`	DEC-20 only, expr

> The function Bug(); can be called from within PSL:RLisp. This starts MAIL (actually MM) in a lower fork, with the To: line set up to PSL-BUGS@UTAH-20. Simply type the subject of the complaint, and then the message.
>
> After typing message about a bug or a misfeature end finally with a <Ctrl-Z>.
>
> <Ctrl-N> aborts the message.

1.9. Tops-20 Interface

In the DecSystem-20 implementation, there are a set of functions that permit the user to access specific Tops-20 services. These include the ability to submit commands to be run in a "lower fork", such as starting an editor, submitting a system print command, listing directories, and so on. We also provide an interface to the DecSystem-20 Jsys function with appropriate support functions (such as bit operations, byte-pointers, etc.).

1.9.1. User Level Interface

The basic function of interest is DoCmds, which takes a list of strings as arguments, concatenates them together, starts a lower fork, and submits this string (via the Rescan buffer). The string should include appropriate <CR><LF>, "POP" etc. A global variable, CRLF, is provided with the <CR><LF> string. Some additional entry points, and common calls have been defined to simplify the task of submitting these commands. Load EXEC to get these functions.

(DoCmds L:string-list): any <u>expr</u>

Concatenate strings into a single string (using ConcatS), place into the rescan buffer using PutRescan, and then run a lower EXEC, trying to use an existing Exec fork if possible.

CRLF [<u>Initially:</u> "<cr><lf>"] <u>global</u>

This variable is "CR-LF", to be appended to or inserted in Command strings for fnc(DoCmds). It is (STRING(Char CR,Char LF)).

(ConcatS L:string-list): string <u>expr</u>

Concatenate string-list into a single string, ending with CRLF.

[??? Probably ConcatS should be in STRING, we add final CRLF in PutRescan ???]

(Cmds [L:string]): any <u>fexpr</u>

Submit a set of commands to lower EXEC

E.g. (CMDS "VDIR *.RED " CRLF "DEL *.LPT" CRLF "POP").

The following useful commands are defined:

(VDir L:string): any <u>expr</u>

Display a directory and return to PSL, e.g. (VDIR "R.*"). Defined as (DoCmds (LIST "VDIR " L CRLF "POP")).

(HelpDir): any expr

Display PSL help directory. Defined as (DoCmds (LIST "DIR PH:*.HLP" CRLF "POP")).

(Sys L:string): any expr

Defined as (DoCmds (LIST "SYS " L CRLF "POP")).

(Take L:list): any expr

Defined as (DoCmds (LIST "Take " FileName CRLF "POP")).

(Type L:string): any expr

Type out files. Defined as (DoCmds (LIST "TYPE " L CRLF"POP")).

While definable in terms of the above DoCmds via a string, more direct execution of files and fork manipulation is provided by the following functions. Recall that file names are simply Strings, e.g. "<psl>foo.exe", and that ForkHandles are allocated by TOPS-20 as large integers.

(Run FILENAME:string): any expr

Create a fork, into which file name will be loaded, then run it, waiting for completion. Finally Kill the fork.

(Exec): any expr

Continue a lower EXEC, return with POP. The Fork will be created the first time this is run, and the ForkHandle preserved in the global variable ExecFork.

(Emacs): any expr

Continue a lower EMACS fork. The Fork will be created the first time this is run, and the ForkHandle preserved in the global variable EmacsFork.

(MM): any expr

Continue a lower MM fork. The Fork will be created the first time this is run, and the ForkHandle preserved in the global variable MMFork.

[??? MM looks in the rescan buffer for commands, so fairly useful mailers (e.g. for BUG reports) can be created. Perhaps make MM(s:string) for this purpose. ???]

(Reset): None Returned expr

This function causes the system to be restarted.

1.9.2. The Basic Fork Manipulation Functions

(GetFork JFN:integer): integer expr

Create a fork handle for a file; a GET on the file is done.

(StartFork FH:integer): None Returned expr

Start a fork running, don't wait, do something else. Can also be used to Restart a fork, after a WaitFork.

(WaitFork FH:integer): Unknown expr

Wait for a running fork to terminate.

(RunFork FH:integer): Unknown expr

Start and Wait for a FORK to terminate.

(KillFork FH:integer): Unknown expr

Kill a fork (may not be restarted).

(OpenFork FILENAME:string): integer expr

Get a file into a Fork, ready to be run.

(PutRescan S:string): Unknown expr

Copy a string into the rescan buffer, and announce to system, so that next PBIN will get this characters. Used to pass command strings to lower forks.

(GetRescan): {NIL,string} expr

See if there is a string in the rescan buffer. If not, Return NIL, else extract that string and return it. This is useful for getting command line arguments in PSL, if MAIN() is rewritten by the user. This will also include the program name, under which this is called.

1.9.3. File Manipulation Functions

These mostly return a JFN, as a small integer.

(GetOldJfn FILENAME:string): integer expr

Get a Jfn on an existing file.

(GetNewJfn FILENAME:string): integer expr

Get a Jfn for an new (non-existing) file.

(RelJfn JFN:integer): integer expr

Return Jfn to TOPS-20 for re-use.

(FileP FILENAME:string): boolean expr

Check if FILENAME is existing file; this is a more efficient method than the kernel version that uses ErrorSet.

(OpenOldJfn JFN:integer): integer expr

Open file on Jfn to READ 7-bit bytes.

(OpenNewJfn JFN:integer): Unknown expr

Open file on Jfn to write 7 bit bytes.

(GtJfn FILENAME:string BITS:integer): integer expr

Get a Jfn for a file, with standard Tops-20 Access bits set.

(NameFromJfn JFN:integer): string expr

Find the name of the File attached to the Jfn.

1.9.4. Miscellaneous Functions

(GetUName): string expr

Get USER name as a string

(GetCDir): string expr

Get Connected DIRECTORY

(ClockTime): string expr

Get the time of day in the form "hh:mm:ss".

(GetLoadAverage): string expr

Get the load average over the last minute in the form "dd.dd".

(InFile [FILS:id-list]): Unknown fexpr

Either solicit user for file name (InFile), and then open that file, else open specified file, for input.

1.9.5. Jsys Interface

The Jsys interface and jsys-names (as symbols of the form jsXXX) are defined in the source file P20U:JSYS.RED.

The access to the Jsys call is modeled after IDapply to avoid CONS and register reloads. These could easily be done open-coded

The following SYSLISP calls, XJsys'n', expect W-values in the registers, R1...R4, a W-value for the Jsys number, Jnum and the contents of the 'nth' register. Unused registers should be given 0. Any errors detected will result in the JsysError being called, which will use the system ErStr JSYS to find the error string, and issue a StdError.

(XJsys0 R1:s-integer R2:s-integer R3:s-integer R4:s-integer Jnum:s-integer): s-integer expr

Used if no result register is needed.

(XJsys1 R1:s-integer R2:s-integer R3:s-integer R4:s-integer Jnum:s-integer): s-integer expr

(XJsys2 R1:s-integer R2:s-integer R3:s-integer R4:s-integer Jnum:s-integer): s-integer expr

(XJsys3 R1:s-integer R2:s-integer R3:s-integer R4:s-integer Jnum:s-integer): s-integer expr

(XJsys4 R1:s-integer R2:s-integer R3:s-integer R4:s-integer Jnum:s-integer): s-integer expr

The following functions are the Lisp level calls, and expect integers or s for the arguments, which are converted into s-integers by the function JConv, below. We will use JS to indicate the argument type. The result returned is an integer, which should be converted to appropriate type by the user, depending on the nature of the Jsys. See the examples below for clarification.

(Jsys0 R1:JS R2:JS R3:JS R4:JS Jnum:s-integer): integer expr

Used is no result register is needed.

(Jsys1 R1:JS R2:JS R3:JS R4:JS Jnum:s-integer): integer expr

(Jsys2 R1:JS R2:JS R3:JS R4:JS Jnum:s-integer): integer expr

(Jsys3 R1:JS R2:JS R3:JS R4:JS Jnum:s-integer): integer expr

(Jsys4 R1:JS R2:JS R3:JS R4:JS Jnum:s-integer): integer expr

The JConv converts the argument type, JS, to an appropriate s-integer, representing either an integer, or string pointer, or address.

(JConv J:{integer,string}): s-integer expr

An integer J is directly converted to a s-integer, by (Int2Sys J). A string J is converted to a byte pointer by the call (Lor 8#10700000000 (Strinf J)). Otherwise a (StdError "'J' not known in Jconv") is produced.

Additional conversions of interest may be performed by the functions Int2Sys, Sys2Int, and the following functions:

(Str2Int S:string): integer expr

Returns the physical address of the string start as an integer; this can CHANGE if a GC takes place, so should be done just before calling the Jsys.

(Int2Str J:integer): string expr

J is assumed to be the address of a string, and a legal, tagged string is created.

1.9.6. Bit, Word and Address Operations for Jsys Calls

(RecopyStringToNULL S:w-string): string <u>expr</u>

S is assumed to be the address of a string, and a legal, tagged string is create d, by searching for the terminating NULL, allocating a HEAP string, and copying the characters into it. This is used to ensure that addresses not in the Lisp heap are not passed around "cavalierly" (although PSL is designed to permit this quite safely).

(Swap X:integer): integer <u>expr</u>

Swap half words of <u>X</u>; actually (Xword (LowHalfWord X) (HighHalfWord X)).

(LowHalfWord X:integer): integer <u>expr</u>

Return the low-half word of the machine representation of <u>X</u>. Actually (Land X 8#777777).

(HighHalfWord X:integer): integer <u>expr</u>

Return the upper half word as a small integer, of the machine word representatio n of <u>X</u>. Actually (Lsh (Land X 8#777777000000) -18).

(Xword X:integer Y:integer): integer <u>expr</u>

Build a Word from Half-Words, actually (Lor(Lsh(LowHalfWord X) 18) (LowHalfWord Y)).

(JBits L:list): integer <u>expr</u>

Construct a word-image by OR'ing together selected bits or byte-fields. L is list of integers or integer pairs. A single integer in the range 0...35, BitPos, represents a single bit to be turned on. A pair of integers, (FieldValue . RightBitPos), causes the integer FieldValue to be shifted so its least significant bit (LSB) will fall in the position, RightBitPos. This value is then OR'ed into the result. Recall that on the DEC-20, the most significant bit (MSB), is bit 0 and that the LSB is bit 35.

(Bits L:list): integer <u>macro</u>

A convenient access to Jbits: (JBits (cdr L)).

1.9.7. Examples

The following range of examples illustrate the use of the above functions. More examples can be found in P20U:exec.red.

```
(Jsys1 0 0 0 0 jsPBIN)
        % Reads a character, returns the ASCII code.

(Jsys0 ch 0 0 0 jsPBOUT)
        % Takes ch as Ascii code, and prints it out.

(De OPENOLDJfn (Jfn)        %. OPEN to READ
 (JSYS0 Jfn (Bits ( (7 . 5) 19)) 0 0 jsOPENF))

(De GetFork (Jfn)      %. Create Fork, READ File on Jfn
    (Prog (FH)
      (Setq FH (JSYS1(Bits '(1)) 0 0 0 jsCFork))
      (JSYS0 (Xword FH Jfn) 0 0 0 jsGet)
      (return FH)))

(De GetOLDJfn (FileName) %. test If file OLD and return Jfn
   (Prog (Jfn)
      (Cond ((NULL (StringP FileName)) (return NIL)))
      (Setq Jfn (JSYS1(Bits'(2,3,17)) FileName 0 0 jsGTJfn))
         % OLD!MSG!SHORT
      (Cond ((Lessp Jfn 0) (return NIL)))
      (return Jfn)))

(De GetUNAME ()      %. USER name
  (Prog (S)
   (Setq S (Mkstring 80))              % Allocate a 80 char buffer
   (JSYS0 s (JSYS1 0 0 0 0 jsGJINF) 0 0 jsDIRST)
   (Return RecopyStringToNULL S)))
                % Since a NULL may be appear before end

(De ReadTTY ()
  (Prog (S)
        (Setq S (MkString 30))    % Allocate a String Buffer
        (Jsys0 S (BITS '(10 (30 . 35))) "Retype it!" 0 jsRDTTY)
              % Sets a length halt (Bit 10),
              % and length 30 (field at 35) in R2
              % Gives a Prompt string in R3
              % The input is RAISE'd to upper case.
              % The Prompt will be typed if <Ctrl-R> is input
        (Return RecopyStringToNULL S)))
              % Since S will now possibly have a shorter
        % string returned
```

CHAPTER 2
PSL UNDER VAX UNIX

2.1. Purpose of This Chapter

This chapter is for beginning users of PSL on the Vax running Berkeley Unix. It begins with descriptions of how to set up various csh variables required by PSL and how to run PSL. The chapter continues with a description of the file structure associated with PSL, an example of use of the PSL system, and miscellaneous hints and reminders. It concludes with a discussion of interfacing to the Unix operating system.

2.2. Getting started on Vax Unix

PSL under Unix makes extensive use of C-shell variables ("dollar sign" variables) to describe pathnames to the various PSL subdirectories.[1] These variables must be defined for PSL to work. To do this you should put the following line at the end of your `.cshrc` file. (The file which is executed whenever you start up a new shell, including when you first log in.)

```
source ~psl/dist/psl-names
```

This line reads a file from the `$psl` directory to define all the csh variables used by PSL. This is absolutely required only for systems personnel installing or updating PSL. It is optional for other users, but eases reference to the PSL sources. These variables are automatically known in PSL. See section 2.4 and the release notes for more details.

[1] Most of this information depends on the use of the Berkeley C-shell (csh) and will need modification (or might not work) if the Bourne shell (sh) is your command shell of choice.

Depending on where the PSL executables have been installed on your system, you may have to change the definition of path in your .login file. The PSL executables typically reside on the $psys directory, so a line something like the following will make them available:

```
set path=(. $psys /bin /usr/bin)
```

However, it is more common for the installer of PSL to create symbolic links to the PSL executables in some standard directory of executable files, which makes the above unnecessary. In short, talk to your local PSL installer to get the details on what is available and how to run it.

2.3. PSL Executables

There are several different executable forms of PSL available, one or more of them may be installed on your system. Type either psl or rlisp to the C-shell. A welcome message indicates the nature of the system running, usually with a date and version number. This information may be useful in describing problems.

bare-psl is a "bare" PSL using Lisp (i.e., parenthesis) syntax. This is a small core-image and is ideal for simple Lisp execution. It also includes a resident loader, so additional modules can be loaded. Bare-psl is used as the base for creating all of the other executables.

Certain modules are not present in the "kernel" or "bare-psl" system, but can be loaded as options. Optional modules can be loaded by executing

```
(LOAD modulename)
```

psl is an installation dependent "enhanced" bare-psl. Your system installer will determined which modules are to be included in your base system. At Utah, the only additional modules in psl are the ones necessary to permit the reading of an initialization file upon startup (see Section 2.6 for details of the initialization files). You can discover which modules are loaded into your system by examining the variable OPTIONS!* upon startup of psl.

rlisp is also an "enhanced" bare-psl with additional modules loaded; typically including the compiler, the RLisp parser and the init-file module. For more information about RLisp see the RLisp chapter in Part 2 of the PSL Reference Manual.

pslcomp is an executable that permits compilation of Lisp, RLisp, or build files into loadable binary modules. pslcomp includes a function that will read the information on the pslcomp executation line, interpret that as the name of a file and compile the file. When the file name does not include an extension, pslcomp first looks for filename.BUILD (an RLisp build file that usually reads in a number of files to make into a single loadable module), filename.SL (a PSL Lisp syntax source file) and finally filename.RED (an RLisp source file). The file will be compiled and a binary file will be created in the SAME DIRECTORY as the input file. Therefore, if there is a file foo.sl in the xx directory:

```
pslcomp foo.sl

  % Will compile xx/foo.sl into xx/foo.b
```

nmode is the NMODE text editor and PSL environment (see the NMODE reference manual for further details).

It is assumed by PSL and RLisp that file names be of the form `*.sl` or `*.lsp` for Lisp files, `*.red` for RLisp files, `*.b` for Fasl files, and `*.lap` for Lap files.

2.4. Unix File Structure in PSL

The specific location of PSL directories is left to the installer, to reflect the conventions of local usage and file system layout. The root of the PSL tree is (on a "typical" installation) located in the home directory of a pseudo-user named "psl", and hence may be accessed as "~psl".

The C-shell provides two kinds of substitution features which can map filenames into their "true" form. One kind of substitution is called "variable substitution", and is used for "$ variables" (i.e., "C-shell variables"). The other kind of substitution is called "filename substitution", and supports "*" and "?" wildcards; and also "~" (twiddle) at the beginning of filenames to refer to home directories. PSL supports "$" and "~" substitutions, but only at the beginning of filenames. As in csh, "environment" variables are automatically known as "$" variables.

Nearly all of the PSL subdirectories have corresponding C-shell variables. The definitions for these variables are in the file ~psl/dist/psl-names. Some of the most commonly used variables are:[2]

$pu	holds sources for many utility routines.
$pvu	holds sources for utilities specific to vax-unix.
$pv	holds sources for those parts of the PSL "kernel" specific to vax-unix.
$pl	holds most PSL "binaries".[3] This is one of the default directories searched by PSL's load function.
$pll	holds "local binaries" and is also searched by the Load function.

Variable substitution in PSL is implemented by having PSL read the file config-names.sl from the PSL root directory and psl-names.sl from the user's home directory.

[2]These variables are also put in the shell environment via setenv to avoid each of the PSL maintenance scripts sourcing psl-names everytime they are executed.

[3]These were once known as LAP files, thus the name "pl" for "PSL LAP" .

These files contain expressions that give the mapping from variables to their values. For example, the expression:

```
(put 'psl 'pslnames "/v/misc/psl/dist")
```

defines the "$psl" variable for use in PSL.

Rather than have the user create psl-names.sl by hand, PSL automatically creates the file for the user and writes an appropriate psl-names.sl file. It reads through the .cshrc and .cshinit files on the home directory and all files "source"d by them collecting "set" statements. The message "creating psl-names.sl, please wait ..." is displayed while the file is being created. When psl-names.sl is being read at PSL startup time, the write times of the C-shell source files are checked and if any are newer than psl-names.sl, then psl-names.sl is created.

PSL uses the concept of a current "working" directory (or dot) as the shell does. Unix filenames are treated as paths from the current directory unless they begin with a slash (in which case they start from the "root" directory). PSL's working directory is initially the same as the working directory of the shell when PSL is started.

Following are some PSL functions available on the Vax Unix version of PSL that pertain to directories.

(cd DIR:string): boolean expr

Like the shell's "change directory" ("cd") utility, sets the current working directory for PSL. Unless cd is executed, the working directory will remain the same as the working directory of the shell at the time the PSL was started. Cd returns T if it successfully finds DIR, NIL otherwise.

(pwd): string expr

Similar to the shell's "print working directory" ("pwd") utility. Returns the current directory of the PSL as a string, terminated with a slash so filenames may be directly concated to it. The trailing slash is ignored by cd.

(path S:string): string expr

This is the function used to perform "variable substitution" on filenames. It examines the argument string and if it starts with "$" or "~", extracts the next string up to the "/" (if any), and converts it to (an upper-case) id. Then an associated string is looked for on the id's property list under the indicator PSLNAMES and in the environment via the getenv function. "~" user names are looked up via getputnam, and the home directory is substituteed. "~/" is the user's home directory. If an associated string is not found, an Error is generated. If S does not start with "$" or "~", it is returned unchanged.

For example, one could type

```
   (cd (path "$pu"))
```

to change to the PSL utility directory as the working directory. (Note that the call to path is done automatically in this case.) When VAX-PATH is loaded (which is the case except in a very "bare" PSL), Open is redefined to apply Path to the filename. Thus Open, In, Dskin, Out, FileP, etc. can use "$" variables in file names without calling Path explicitly.

2.5. Sample Session with Vax Unix PSL

The following is a transcript of running PSL under Unix.

```
1 cs> psl
PSL 3.2, 26-Oct-83

1 Lisp> % Notice the numbered prompt.
1 Lisp> % Comments begin with "%" and do not change the prompt
1 Lisp> % number.
1 Lisp> (Setq Z '(1 2 3))  % Make an assignment for Z.
(1 2 3)
2 Lisp> (Cdr Z)            % Notice the change in prompt number.
(2 3)
3 Lisp> (De Count (L)      % Count counts the number or elements
3 Lisp>    (Cond ((Null L) 0)  % in a list L.
3 Lisp>          (T (Add1 (Count (Cdr L))))))
COUNT
4 Lisp> (Count Z)          % Call Count on Z.
3
5 Lisp> (Tr Count)  % Trace the recursive execution of "Count".
(COUNT)
6 Lisp>             % A call on "Count" now shows the value of
6 Lisp>             % "Count" and of its arguments each time
6 Lisp> (Count Z)   % it is called.
COUNT being entered
   L:   (1 2 3)
  COUNT (level 2) being entered
     L: (2 3)
    COUNT (level 3) being entered
       L:      (3)
      COUNT (level 4) being entered
         L:     NIL
      COUNT (level 4) = 0
    COUNT (level 3) = 1
  COUNT (level 2) = 2
COUNT = 3
3
7 Lisp> (Untr Count)
NIL
8 Lisp> (Count 'A)  % This generates an error causing the break
                           % loop to be entered.
***** An attempt was made to do CDR on 'A', which is not a pair
Break loop
9 Lisp break>> ?
BREAK():{Error,return-value}
----------------------------
This is a Read-Eval-Print loop, similar to the top level loop,
except that the following IDs at the top level cause functions to
be called rather than being evaluated:
```

```
?        Print this message, listing active Break IDs
T        Print stack backtrace
Q        Exit break loop back to ErrorSet
A        Abort to top level, i.e. restart PSL
C        Return last value to the ContinuableError call
R        Reevaluate ErrorForm!* and return
M        Display ErrorForm!* as the "message"
E        Invoke a simple structure editor on ErrorForm!*
               (For more information do Help Editor.)
I        Show a trace of any interpreted functions

See the manual for details on the Backtrace, and how ErrorForm!* is
set.  The Break Loop attempts to use the same TopLoopRead!* etc, as
the calling top loop, just expanding the PromptString!*.
NIL
10 Lisp break>>          % Get a Trace-Back of the
10 Lisp break>> I        % interpreted functions.
Backtrace, including interpreter functions, from top of stack:
CDR COUNT ADD1 COND COUNT
NIL
11 Lisp break>> Q        % To exit the Break Loop.
12 Lisp>                 % Load in a file, showing its execution.
12 Lisp>                 % The file contains the following:
12 Lisp>                 % (Setq X (Cons 'A (Cons 'B Nil)))
12 Lisp>                 % (Count X)
12 Lisp>                 % (Reverse X)
12 Lisp> (Dskin "small-file.sl")
(A B)
2
(B A)
NIL
13 Lisp> (Quit)
2 cs> jobs

[1] +Stopped    psl

3 cs> % 1
"Continued"
14 Lisp> (ExitLisp)
4 cs>
```

2.6. Init Files for PSL

On starting up, psl will read and execute the file .pslrc on your home directory, if the file is present. Similarly, rlisp will use the file .rlisprc; pslcomp will use .pslcomprc; and nmode will use .nmoderc. These files must use Lisp syntax, even the one for RLisp.

These initialization files are typically used to load modules of personal interest, and to initialize variables (such as !*BREAK) to suit personal tastes.

2.7. Error and Warning Messages

Many functions detect and signal appropriate errors (see the chapter on error handling in Part 1 of this manual for details); 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.

Sometimes one sees a prompt of the form:

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

This means you have tried to define a function with the same name as a function used by the PSL system. A Y, N, YES, NO, or B response is required. (Note that Y and N don't work in NMODE.) B starts a break loop. After quitting the break loop, answer Y, N, Yes, or No to the query. See the definition of YesP in the input/output chapter of this manual. An affirmative response is extremely dangerous and should be given only if you are a system expert. Usually this means that your function must be given a different name.

A common warning message is

*** Function "FOO" has been redefined

If this occurs without the query above, you are probably redefining one of your own functions. This happens normally if you read a file, edit it, and read it in again.

2.8. Reporting Errors and Misfeatures

Send bug mail to PSL-BUGS@UTAH-20(Arpanet) or utah-cs!psl-bugs(uucp address). The message will be distributed to a list of users concerned with bugs and maintenance, and a copy will be kept in SS:<PSL>BUGS-MISSFEATURES.TXT at UTAH-20.

2.9. Unix Interface

2.9.1. Miscellaneous Unix Functions

(ExitLisp): undefined expr

Since the PSL quit behaves like a ^Z, saving your core image, a separate function is needed when you really want the PSL to terminate. ExitLisp does it. (A ^\ from the keyboard has the same effect, assuming you have your core-dump limit set low.)

(ExitLispWithStatus CODE:integer): undefined   expr

Like ExitLisp but returns an integer status code to Unix, available as $status in csh. ExitLisp always returns status 0, to signal normal exit.

(GetEnv ENVVARNAME:string): string   expr

Returns value of the specified Unix environment variable as a string, or NIL if the argument is not a string or the environment variable is not set.

(System UNIXCMD:string): undefined   expr

Starts up a sub-shell and passes the Unix command to it via the Unix "system" command. The working directory of the command will be the same as PSL's current working directory.

(FileStatus FileName:string, DoStrings:boolean): Alist   expr

Returns an alist of information about the file, or NIL if the file was not found. Both the numerical file information and the string interpretation are given if DoStrings is T. Time and consing can be saved if the interpretations are not needed by calling with DoStrings NIL. An example of the return value:

```
'(  ( Owner  . ( "psl" . 39 ))
    ( Group  . ( "small" . 8 ))
    ( Mode . ( "-rwxr-xr--" . 8#764 ))
    ( Size . ( "" . 123456789 ))
    ( WriteTime . ( "Tue Nov 22 13:48:26 1983" . 438382106 ))
    ( AccessTime . ( "Tue Nov 22 13:48:26 1983" . 438382106 ))
    ( StatusChangeTime .
                  ( "Tue Nov 22 13:48:26 1983" . 438382106 ))
)
```

File size is given in bytes. Numerical times are Unix system time counts in seconds since 00:00:00 GMT, Jan. 1, 1970, and can be used for file time comparisons or sorting.

2.9.2. Loading C code into PSL

(oload load-specs:{string, NIL}): boolean   expr

Oload[4] provides a mechanism for linking Unix ".o" and ".a" files ("object" and "archive" files) into a running PSL. It was developed to get access to existing C code driver libraries for graphics devices, but should work for any Unix compiled code with C calling conventions.

The single argument to oload is a string containing arguments for the Unix "ld" loader, separated by blanks. File names ending in ".o" are compiled relocatable code files. ".a" files are "ar" load libraries, which are assumed to contain a set of ".o" files, all of which are to be loaded. Other loader arguments should follow, specifying whatever libraries are necessary to satisfy all external references from the ".o" and ".a" files mentioned. Library specs are in the form "-lfoo" to search the "libfoo.a" library on /lib, /usr/lib, /usr/local/lib, etc., e.g., -lc for the C library.

Oload performs an "incremental" (-A flag) load. Symbols which are already known in the running PSL will be linked to the existing addresses.

If <u>LOAD-SPECS</u> is NIL, an attempt is made to re-load from an existing .oload.out file. This can only be done if the BPS and WARRAY base addresses are <u>exactly</u> the same as they were on the previously done, full oload. An error message results if the BPS locations are different. This is meant to facilitate rapidly repeating an oload at startup time, and is probably useful only to a PSL system expert.

A customized version of PSL may be saved by the function SaveSystem, after first performing oloads and loading PSL code which interfaces to the oloaded code.

Oload returns a status code of T if it succeeds, or NIL if not.

2.9.3. Calling oloaded functions

[??? We need to clarify the (rather nontrivial) process by which one can convert code that uses oload into "kernel" code. ???]

All entry points and global data objects loaded by oload are made known to the PSL system. C functions may be called from compiled code <u>only</u>, and are flagged ForeignFunction by oload. Data areas are flagged ForeignData, with a property containing a pair of the data location and size in bytes for use by SYSLisp interface code.

Foreign function calls may be compiled into Fasl files, in which case the C function names must be flagged foreign function at compile time in order to get the proper calling sequence.

The names of oloaded C functions within PSL are the "true" names, which have an underscore ("_") prefixed to the C name. This makes it easy to make a compiled "pass through" interface function which gives the same name within PSL as the C names. For example, your PSL definition might look like this:

[4]The name "oload" comes from the fact that the function loads object, or ".o", files.

```
(de foo ()
  (_foo))
```

while your C program might read:

```
foo()
{
    fflush( stdout );
}
```

Note that the C language version of foo must be all lower case, and does not start with an underscore (the underscore is added by the C compiler).

Functions which take integer arguments can be called directly, due to the "invisible tagging" of integers with up to 27 bits in the Vax implementation of PSL.[5] Similarly, integer return values will be passed back from the C functions. String or structured arguments will require a bit of conversion code in the interface functions, requiring removal of tags when passing arguments to C functions, and adding them to return values. For returning strings from C functions, we have provided the function ImportForeignString.

(ImportForeignString C_STRING:word): string expr

Constructs and returns a PSL string, given a C string (something of type "pointer to character"). A NULL (i.e. 0) string pointer is returned as NIL.

Note that, currently, foreign function calls may have no more than five arguments and that no support is provided for floating point and struct values. We hope to eventually remedy this problem.. These restrictions may be circumvented by putting arguments in work areas and passing the address of the work area as an argument to an intermediate C "kludge function" which unpacks the real arguments and passes them on to the target C function.

If work areas are needed in SYSLisp interface code, as when arrays must be passed to the C code, use a LispVar to hold the address of a word block acquired via GtWArray (for static arrays) or GtWrds (for dynamic blocks in the heap). Pass the array address to the C function as the pointer argument. (The LispVar function is used to access PSL "FLUID" variables in the symbol table from SYSLisp code.)

2.9.4. Oload Internals

Oload invokes the Unix "ld" loader through a C-shell script to convert the relocatable code in .o files into absolute form, then reads the absolute form into space allocated within the BPS area of PSL. The text segment goes at the low end of BPS, and the data

[5]Working with integers larger than this will require more care on the part of the programmer.

and bss segments go at the high end, following the BPS storage allocation conventions of the PSL compiler.

Since an incremental (-A) load is done, oload needs a filename path to the executable file containing the loader symbol table of the previous load. The variable SymbolFileName* tells both oload and the SaveSystem and DumpLisp functions the filename string to use. (Since oload reads the executable file, it should be publicly readable.)

When PSL is started, SymbolFileName* is automatically set to the name of the executed PSL file. This is done by importing the Unix argument string to variable UnixArgs*. UnixArgs* is a vector, and its zeroth entry is the (possibly partial) path to the PSL file which was executed. The Unix environment variable PATH contains a set of path prefixes to which partial paths are appended, until a valid filename results. "." refers to the path to the current directory, which is returned by the pwd function. (Unix system interface functions are contained in file $pv/system-extras.red.)

SymbolFileName* is set to ".oload.out" by oload, so that successive loads will accumulate a loader symbol table, and so that the C function "unexec", called by DumpSystem, will get the right symbol table in the saved PSL. (It may be useful to know that the initial value of SymbolFileName* is saved in StartupName*.)

A number of work files are created on the current directory by the oload script, with file names that begin ".oload". In particular, the .oload.out file is quite large because it spans the gap of unused space in BPS. It is a good idea to remove those files if you do not intend to repeat the load exactly.

2.9.5. I/O Control Functions

(EchoOff): undefined <u>expr</u>

> EchoOff enters raw, character-at-a-time input mode for Emode, Nmode, and similar keystroke oriented environments.

(EchoOn): undefined <u>expr</u>

> EchoOn is used to return to normal, line oriented, input mode after calling EchoOff.

The names EchoOff and EchoOn are not exactly appropriate. In addition to turning off echoing, EchoOff also turns off processing of output characters. For example, on some systems a linefeed character will normally be printed as a carriage return/linefeed pair. EchoOff turns off this processing, so that it will print as just a linefeed. EchoOff may also turn off the "special" meaning of some input characters. For example, typing the control-C character serves to interrupt the execution of a program, but the Unix version of EchoOff turns this off.

(CharsInInputBuffer): integer expr

Returns the number of characters waiting for input from the TTY, including those still in the Stdio buffer and those not yet read from Unix.

(PauseForInput n60ths): undefined expr

PauseForInput pauses until either the time limit given by N60THS (in 60ths of a second) is exceeded, or until input is available from the standard input, whichever comes first. PauseForInput is only meant to be used after calling EchoOff, otherwise it may not work correctly.

(FlushStdOutputBuffer): None Returned expr

The standard output from PSL is in Stdio line-buffered mode, and is normally flushed to the TTY whenever an end-of-line is printed or before waiting for input. In screen-oriented output environments like Emode/Nmode which use screen cursor positioning, it is necessary to explicitly flush the buffer at appropriate times.

(ChannelFlush Chnl:io-channel): None Returned expr

Flushes any channel, as FlushStdOutputBuffer does for StdOut*.

CHAPTER 3
PSL ON THE APOLLO

3.1. Purpose of This Chapter

This chapter is for beginning users of PSL on the Apollo. It begins with descriptions of how to set up various logical device definitions required by PSL and how to run PSL. The chapter continues with a description of the file structure associated with PSL, an example of use of the PSL system, and miscellaneous hints and reminders. The next section contains a discussion of interfacing to the Aegis operating system. The chapter concludes with a description of the DumpLisp utility.

3.2. Setting up Logical Names

The Apollo version of PSL uses a link in the current naming directory to reference files it needs while running. This link is called "~p" and should have a link text pointing to the psl_links subdirectory of the root PSL directory.

For example, if the PSL system was installed in the directory "/lisp", then the "~p" link would be created with

```
    $ crl ~p /lisp/psl_links -r
```

This link MUST exist for PSL to run, since PSL uses it to resolve the location of the init files.

With the link in place,

```
   $ ~p\psl
```

starts up the PSL executable. It may be desirable to add a link in your ~com directory to the psl executable, i.e.:

```
   $ crl ~com/psl ~p\psl
```

or to add the directory "~p\" to your command search path.

This executable does not contain any optional loadable modules. They can be loaded as options. Optional modules can be loaded by executing

```
   (LOAD modulename)
```

The global variable OPTIONS!* contains a list of modules currently loaded; it does not mention those in the "bare-psl" kernel.

If more complete versions of the system are desired, they can be created with SaveSystem (see Section 3.9).

3.3. File Structure on the Apollo

When PSL is installed, a system of link files is used to access the various directories. These are generally easier to type and also provide a system independent method for PSL to locate its system files. They are accessed with a link called "~p" on the user's naming directory. The links referencing the files are indicated by [~p/xxxx] below.

binfiles/?*	Directory containing binary files to rebuild kernel and each module's init files[~p/bin]
comp/?*	Directory containing general compiler sources[~p/c]
comp/apollo/?*	Apollo-specific compiler sources [~p/ac]
demo/?*	Directory containing demo programs (note they depend on all_syscalls, some may require L_Initplot to be called before starting).[~p/demo]
doc/?*	Documentation directory; Pascal_compat.txt describes some of the more useful functions in all_syscalls. [~p/d]
help/?*	Help files for use with the help utility [~p/h]
lpt/?*	Machine readable copy of the manual, suitable for printing on a line printer. [~p/lpt]

kernel/?*	Directory for sources to the kernel. [~p/k]
kernel/apollo/?*	Contains subdirectory with Apollo specific sources. [~p/ak]
tests/?*	Directory containing general PSL timing tests. [~p/t]
test/apollo/?*	Subdirectory contains specific files for the apollo version. [~p/at]
lap/?*	Directory for PSL loadable binaries (for use with "load" function) [~p/l]
support/?*t	Directory containing Apollo system specific sources (e.g., syscall packages) [~p/sup]
util/?*	Directory containing general PSL utilities sources [~p/u]
psl.map	Binder Mapfile of the PSL kernel (used for debugging).
psl.init	INIT file required for PSL system startup (loads each of the init files from the ~p/bin directory. To make startup faster, you may want to concatenate all of the init files together into one file, saving the overhead of opening 17 files).
psl	Executable file.
psl-intro	A small file to point users at PSL documentation, and how to start PSL. It should be edited for local use.
psl_links/?*	Directory containing abbreviated names for access to the various PSL directories. Its contents should be rebuilt with the psl_names script before the system is used.
psl_names	Command script for creating ~p/... directories.

3.4. Sample Session with Apollo PSL

The following is a transcript of running PSL on the Apollo.

```
$ ~p\psl
PSL 3.2, 11-Dec-83

1 Lisp> % Notice the numbered prompt.
1 Lisp> % Comments begin with "%" and do not change the prompt
1 Lisp> % number.
1 Lisp> (Setq Z '(1 2 3))  % Make an assignment for Z.
(1 2 3)
2 Lisp> (Cdr Z)            % Notice the change in prompt number.
(2 3)
3 Lisp> (De Count (L)      % Count counts the number or elements
3 Lisp>    (Cond ((Null L) 0)  % in a list L.
3 Lisp>          (T (Add1 (Count (Cdr L))))))
COUNT
4 Lisp> (Count Z)          % Call Count on Z.
3
5 Lisp> (Tr Count)  % Trace the recursive execution of "Count".
(COUNT)
6 Lisp>             % A call on "Count" now shows the value of
6 Lisp>             % "Count" and of its arguments each time
6 Lisp> (Count Z)   % it is called.
COUNT being entered
   L:   (1 2 3)
  COUNT (level 2) being entered
     L: (2 3)
    COUNT (level 3) being entered
       L:      (3)
      COUNT (level 4) being entered
         L:     NIL
      COUNT (level 4) = 0
    COUNT (level 3) = 1
  COUNT (level 2) = 2
COUNT = 3
3
7 Lisp> (Untr Count)
NIL
8 Lisp> (Count 'A)  % This generates an error causing the break
                           % loop to be entered.
***** An attempt was made to do CDR on 'A', which is not a pair
Break loop
9 Lisp break>> ?
BREAK():{Error,return-value}
----------------------------
This is a Read-Eval-Print loop, similar to the top level loop,
except that the following IDs at the top level cause functions to
be called rather than being evaluated:
```

```
?         Print this message, listing active Break IDs
T         Print stack backtrace
Q         Exit break loop back to ErrorSet
A         Abort to top level, i.e. restart PSL
C         Return last value to the ContinuableError call
R         Reevaluate ErrorForm!* and return
M         Display ErrorForm!* as the "message"
E         Invoke a simple structure editor on ErrorForm!*
                 (For more information do Help Editor.)
I         Show a trace of any interpreted functions

See the manual for details on the Backtrace, and how ErrorForm!* is
set.  The Break Loop attempts to use the same TopLoopRead!* etc, as
the calling top loop, just expanding the PromptString!*.
NIL
10 Lisp break>>         % Get a Trace-Back of the
10 Lisp break>> I       % interpreted functions.
Backtrace, including interpreter functions, from top of stack:
CDR COUNT ADD1 COND COUNT
NIL
11 Lisp break>> Q       % To exit the Break Loop.
12 Lisp>                % Load in a file, showing its execution.
12 Lisp>                % The file contains the following:
12 Lisp>                % (Setq X (Cons 'A (Cons 'B Nil)))
12 Lisp>                % (Count X)
12 Lisp>                % (Reverse X)
12 Lisp> (Dskin "small-file.sl")
(A B)
2
(B A)
NIL
13 Lisp> (Quit)
$
```

3.5. Init Files

Init files are available to make it easier for the user to customize PSL to his/her own needs. When ~p\psl is executed, if a file ~user_data/psl exists, it will be read by psl.init after the other init files are read. Currently all init files must be written in Lisp syntax. They may use Faslin or Load as needed. More details on init files are available in Part 1 of this manual.

3.6. Error and Warning Messages

Many functions detect and signal appropriate errors (see the chapter on error handling in Part 1 of this manual for details); 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.

Sometimes one sees a prompt of the form:

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

This means you have tried to define a function with the same name as a function used by the PSL system. A Y, N, YES, NO, or B response is required. B starts a break loop. After quitting the break loop, answer Y, N, Yes, or No to the query. See the definition of YesP in the input/output chapter of this manual. An affirmative response is extremely dangerous and should be given only if you are a system expert. Usually this means that your function must be given a different name.

A common warning message is

*** Function "FOO" has been redefined

If this occurs without the query above, you are redefining your own function. This happens normally if you read a file, edit it, and read it in again.

3.7. Reporting Errors and Misfeatures

Send bug MAIL to PSL-BUGS@UTAH-20 (Arpanet) or utah-cs!psl-bugs (uucp). The message will be distributed to a list of users concerned with bugs and maintenance, and a copy will be kept in <PSL>BUGS-MISSFEATURES.TXT at UTAH-20.

3.8. Aegis System Interface

3.8.1. Introduction

The Aegis operating system provides a wide variety of user-callable routines for system services such as graphics, file manipulation, pad and window usage, etc. A package called SysCalls has been developed to allow Apollo PSL to interactively use these routines. This package allows virtually any procedure documented in the Apollo System Programmer's guide to be called from PSL without requiring modifications to the PSL system.

In addition to calling existing Aegis routines, user written libraries can also be used by loading them with the shell command INLIB before executing PSL. The only restriction on user-libraries is they must use call by reference parameters instead of call by value.

3.8.2. How the Package Works

Aegis stores a symbol table for all globally accessible routines. The undocumented system function KG_$LOOKUP(SYMBOLNAME) returns the address of the routine specified in the string SYMBOLNAME. To maintain Fortran compatibility, all Aegis routines are call by reference, that is, the parameters passed to the routines are pointers to the actual data elements pushed onto the stack before calling the routine. The sources for the syscall package reside in ~p/sup/syscalls.red.

The syscall package reserves an area of storage called the CALLADDRESSBLOCK. This is where the parameters (pointers to Pascal data objects) are stored when setting up the SysCall. When the call is executed, the elements (32 bit pointers) are pushed sequentially onto the stack. For immediate data values (i.e., 16 and 32 bit integers) a space in memory is reserved called the CallArgumentBlock which is used as a place to put these parameters so that there is a memory address for them which can be used for the call by reference. The following functions call Aegis routines, automatically store the desired values into the CallArgumentBlock and create the proper pointers in the CallAddressBlock.

(SysCall RoutineName:ID, ArgCount: Integer): any　　　　expr

Once the argument block is correctly loaded, SysCall executes the call. The ROUTINENAME is the name of the Pascal system routine. A value is returned by Pascal if the routine is declared as a Pascal FUNCTION instead of a PROCEDURE. Some examples:

```
SysCall('GPR_!$SET_ATTRIBUTE_BLOCK,2);          % call to Pascal
                                                %  procedure
AtrDescptr:=SysCall('GPR_!$ATTRIBUTE_BLOCK,2);  % call to Pascal
                                                %  function
```

3.8.3. Handling Simple Arguments

Interface to the SysCalls package is generally through three routines: PutArg, GetArg, and ClearArg.

(PutArg ArgNum: integer, LispArg: any, PasType: IDlist): NIL　　　　expr

PutArg places incoming Lisp arguments into the argument block (Incoming arguments are those declared as "IN" within the Pascal definition). ARGNUM is the Pascal input argument number, determined by counting from left to right in the Pascal definition, with the first argument being 1. LISPARG is the input parameter declared in the Lisp function definition. PASTYPE is the data type the Lisp parameter should be converted to. Currently supported are (Integer16), (Integer32) and (String). Some examples:

```
PutArg(1, StreamID, '(Integer16));

PutArg(2, BufferPointer, '(Integer32));

PutArg(4, FileName, '(String));
```

(ClearArg ArgNumber: Integer): NIL <u>expr</u>

Before the SysCall procedure can be called, any arguments returned by the system procedure (declared as "OUT" in the Pascal definition) must be cleared with ClearArg so space is properly reserved. <u>ARGNUMBER</u> is the Pascal argument, as above. Note this routine should always be called to clear the returned status (Status_$t) from a system routine.

(GetArg ArgNumber: Integer, PasType:IDlist, LispType: IDlist): Any <u>expr</u>

GetArg is used for retrieving returned values from the Pascal procedure. (Note values returned from a Pascal FUNCTION are handled with the SysCall function, above). After the SysCall has completed, the <u>CALLADDRESSBLOCK</u> contains the addresses of the actual data items. To get these values back to Lisp, a scalar (local) variable is usually declared to receive the value from GetArg. <u>ARGNUMBER</u> and <u>PASTYPE</u> paramaters are used as described in PutArg, above. The <u>LISPTYPE</u> is a corresponding Lisp variable type, which can be (String), (Inum) [integers of 16 bits or less, e.g. 16 bit Pascal quantities], or (FixN) [integers of > 16 bits, e.g., pointers or Status words].

Some examples: (all destinations declared Scalar)

```
Status:=GetArg(5, '(Integer32), '(FixN));
          % 32 bit Aegis status word

Font_ID:=GetArg(2, '(Integer16), '(Inum));
          % 16 bit Pascal integer

FileName:=GetArg(2, cons('STRING,Length), '(String));
                    %  Note Pascal string is built from length
                    %  the string type
```

3.8.4. More Complicated Data Types

When the data type to be passed or received from the SysCall package is more complicated than a simple integer or string, the Lisp program making the call is expected to handle the data allocation and storage.

(PutObjPointer ArgNumber: integer LispPointer: FixN): None expr

PutObjPointer loads the <Lisp Pointer> to the object into the corresponding argument slot in CallAddressBlock. This should be done before invoking SysCall(), even for OUTgoing arguments. For example, suppose we wish to pass a list to a Pascal routine requiring a record declared with:

```
type Position_t = record
                    Xpos, Ypos: integer;
                   end;
```

From the Apollo Pascal Programmer's guide, we discover this record is stored as two 16 bit integers stored into a single fullword. Thus, the resulting Lisp code would be (assuming LispPos is a list):

```
begin scalar PosWord;
   PosWord:=GtWrds(1);  % allocate one 32 bit word and have PosWord
                        % point to it.
   PutHalfword(PosWord, 0, First LispPos);  % Place the first arg in
                                            % the first 16 bits of the
                                            % fullword.
   PutHalfword(PosWord, 1, Second LispPos);
                                            % Note the second argument
                                            % denotes the location in the
                                            % array returned by GtWrds.
   PutObjPointer(2, PosWord);               % Now put the pointer into
                                            % the call address block as
       :                                    % the second argument.
       :
       :
   SysCall('THE_!$ROUTINE_NAME,3);        % Call routine, with three
                                          % args in this case.
```

In cases where the routine returns a complicated argument, the allocation and call to PutObjPointer must still happen before the actual routine is invoked. For example (Using the same data types as above):

```
    begin Scalar PosWord;
      PosWord:=GtWrds(1);            % Allocate space for it.
      PutObjPointer(1, PosWord);     % Place the address of the array into
            :                        % the CallAddressBlock as argument 1
            :
      SysCall('THE_!$ROUTINE,3);     % Call Aegis routine
            :                        %  (three args specified).
            :
            :
      return GetHalfWord(PosWord,0) . GetHalfWord(PosWord,1);
                                     % Return the arguments as a lisp
                                     % dotted pair.  Note that GetArg was
                                     % NOT called; Pascal used the address
                                     % we specified in PutObjPointer.
```

3.8.5. Some Real Examples

These are some examples taken from the GPR (Graphics Primitives Library):

```
%
% GPR_Inq_Coordinate_Origin- returns the value of the current bitmap's
% origin as a lisp dotted pair, i.e. (orgX . orgY).
%
SysCall PROCEDURE   gpr_inq_coordinate_origin ();
begin scalar  origin,         % the coordinate origin
              status;         % returned Aegis status word.

  Origin:=GtWrds(1);          % Allocate space for returned value.
  PutObjPointer(1, Origin);   % Place pointer in CallAddressBlock as first
                              % arg.
  ClearArg(2);                % Clear the second arg (the returned status)

  SysCall('GPR_!$INQ_COORDINATE_ORIGIN, 2);     % Make the call

  Status:=GetArg(2, '(Integer32), '(FixN));     % get the status back...
  if Status eq 0             % and check to see if it's ok (=0).
    then <<                  % if it is, return the valid args.
      Origin:=GetHalfword(Origin, 0) . GetHalfword(Origin, 1);
                             % Reform the Origin(as a pointer to a fullword)
                             % into a Lisp dotted pair.
      return Origin;         % and return it.
    >> else Return ApolloError(Status);   % else, return trouble.
                             % Note: ApolloError is defined in the module
                             % ApolloCalls.b, it calls Error_$Print() to get
                             % a textual description of what went wrong.
end;
```

```
% GPR_LOAD_FONT_FILE loads a font contained in a file into an appro-
% priate area (based on the current display mode and configuration).
%
SysCall PROCEDURE   gpr_load_font_file (
                   pn,          % pathname of file
                   pnlen);      % pathname length

begin scalar  font_id,          % returned font id
                status;         % returned status

  PutArg(1,PN,       '(STRING));    % Pass args into Argument block.
  PutArg(2,PNlen, '(Integer16));
  ClearArg(3);                      % Set up slots for returned args.
  ClearArg(4);

  SysCall('GPR_!$LOAD_FONT_FILE,4);

  Status:=GetArg(4, '(Integer32), '(FixN));
  if status eq 0                    % Check the status
    then <<
      Font_ID:=GetArg(3, '(Integer16), '(Inum));% If OK, return the Font_ID
      return Font_ID;
    >> else Return ApolloError(Status);
end;


% GPR_LINE draws a line from the CP to the given position
%  and sets the CP to the given position.

SysCall PROCEDURE   gpr_line (x,y);

begin scalar status;         % returned status

  PutArg(1,x,'(Integer16));     % Place args in parameter block
  PutArg(2,y,'(Integer16));
  ClearArg(3);                  % Clear way for status.

  SysCall('GPR_!$LINE,3);

  Status:=GetArg(3, '(Integer32), '(FixN));
  if Status eq 0 then return Status     % Function returns 0 (status ok)
                                        % if all goes well.
    else return ApolloError(Status);
end;
```

Note how the status is handled; in Aegis calls it is usually returned as the last argument. Most of the time it is handled as above (Returned with ApolloError if there was an error) but in special cases (e.g., the MBX, STREAM, and SMD calls) the program may need to take specific action on certain status values. See the Apollo System Programmer's Guide for more information.

3.8.6. System Interface Package

Many of the more frequently used Aegis calls have already been defined in files called CAL_SYSCALLS, MBX_SYSCALLS, PAD_SYSCALLS, etc., corresponding to the respective packages defined for Pascal or Fortran in /sys/ins/... . These files are designed to 'look' like the Pascal include files, with the argument passing methods changed to Lisp conventions (like the examples in the previous section). In addition to these packages, there is a package defined in ~p/sup/Pascal_compat.red containing a number of definitions using the syscall package for Pascal routines built into the kernel in earlier releases. It also contains a number of routines for making life in Lisp easier; for example, the function `L_Invoke_Shell()` invokes a shell underneath PSL. By typing ^Z, control can be returned back to PSL.

Generally these functions are accessed by doing:

```
(Load All_Syscalls)
```

This loads the actual syscall package along with a collection of the most commonly used XXX_Syscalls packages and Pascal_Compat. Details on the contents of the Pascal_Compat package can be found in ~p/d/Pascal_Compat.txt, and the files contained in All_syscalls can be determined by looking in ~p/sup/All_Syscalls.build. Consult the sources to the xxx_Syscalls packages found in ~p/sup/xxx_Syscalls.red for the exact calling sequences of the individual routines. Also note the implementations of the XXX_syscalls packages are not guaranteed to be complete or accurate; in many cases routines were implemented as they were needed.

3.8.7. Demonstration Program

On the directory ~p/demo there is a small demonstration programing showing how the graphics primitives (GPR) can be used with PSL. To run the program, first enter PSL, then give the commands:

```
(load all_syscalls)   % load the rlisp parser & syscall routines

(Faslin "~p/demo/pattern.b") % load the demo program.
(L_InitPlot)                 % initialize the graphics routines.Note
                             % the window is now broken into three
                             % sections, Lisp Input, Lisp Output,
                             % and a graphics frame.
(SQpat)                      % run the demo program.  Note you may
                             % have to scroll the graphics window
                             % to see the entire result
(L_erase)                    % Erase the graphics...
(L_EndPlot)                  % Closes the graphics frame and returns
                             % to two window mode.
```

3.9. Dumplisp Utility

A utility for saving a running Lisp environment called DumpLisp is distributed with this release.

(DumpLisp Filename:String): None   expr

DumpLisp works by copying the running PSL image (including the active HEAP and BPS space) into a mapped object and then un-mapping it. When a PSL saved image is started again, it copies the saved image into your current working directory and maps this into your address space. FILENAME is the name used in the first part of the output files, as described below.

3.9.1. Using DumpLisp

The first step is to load the DumpLisp module along with any other modules you wish to have in the saved system:

```
(load DumpLisp Rlisp Compiler All_Syscalls)
```

Then execute the DumpLisp procedure, giving the filename with no extension, for example:

```
(DumpLisp "/tmp/pslsave")
```

The above creates two files, the mapped image **/tmp/pslsave.sav**, and another file containing information about the image (such as the re-entry address) called **/tmp/pslsave.mif** (MIF stands for Map Information File). Note when DumpLisp completes you are still in PSL, and further changes won't appear in the saved image since the object has already been mapped out. Note also DumpLisp fails if the .sav file already exists.

In addition to DumpLisp, the PSL system contains a function called SaveSystem allowing specific initialization code to be executed when a saved image is started. See Part 1 of the PSL manual for more details.

3.9.2. Using Saved Images

[??? A transcript of saved PSL image use is badly needed here ???]

To run a saved image, use the program **run_image**, stored in ~p\. It takes one argument, the name of the file the PSL system was saved under (again, without extension). For example, to run the above image:

```
~p\run_image /tmp/pslsave
```

This copies the saved image ("/tmp/pslsave.sav" in this case) to a file in your working directory with a .cor extension (e.g., "pslsave.cor"). It then maps in the .COR file and executes it. Note the saved image always starts up in Lisp mode, even if it was saved while running RLisp. RLisp can be re-entered by typing

```
(rlisp)
```

as usual. If it is desired for the saved image to come up in RLisp, use the SaveSystem function to write out the image (see the PSL manual for more details).

If the saved PSL system is exited by giving it an EOF (^Z on most systems), the PSL image is unmapped, and the .COR file is deleted. NOTE- if you change the working directory of the process while a saved PSL is running, it won't be able to find the .COR file when it exits. This is usually not desirable, since .COR files generally take up a lot of disk space. If the PSL image is exited by the Quit() function, a quit fault (^Q on most systems) or a fatal error, it is not unmapped but instead is still in the address space. Executing the Aegis command **las** prints out a listing of the current address space and will indicate (usually towards the end of the listing) if the PSL image is mapped in.

In this case, it can be re-entered without mapping it back in by executing the program **~p\restart**. **Restart**, like **run_image**, takes one argument, the PSL saved image name. For example, if the above image was exited with a ^Q (quit fault) or a call to Quit,

```
~p\restart /tmp/pslsave
```

would re-enter the image. Again, like **run_image**, PSL comes back in Lisp mode even if it was running RLisp. **Restart** is useful when PSL dies with a fatal error or when a run-away PSL program must be stopped with ^Q. Again like **run_image**, restart un-maps the image if PSL is exited with an end of file (^Z). **Restart** aborts with

```
reference to illegal address (from OS / MST manager)
```

if it is executed without a PSL image already in the address space.

If enough memory is not available at the address the image needs to reside at, **run_image** aborts with the error

```
        no space available (from OS / MST manager)
```

This error also occurs if **run_image** is used when a PSL image is already in the address space (use **restart** instead).

3.9.3. Other DumpLisp Details

When a PSL system is saved with DumpLisp, it is stored starting at a specific address. The image must be loaded into exactly that same address to work correctly. On a specific node, this is normally not a problem. But if the saved image is moved to another node (particularly one with a different hardware configuration, e.g., a PEB board is installed) it is possible the previously saved address may not be available when you try to load it in again.

Another important consideration is that saved PSL images usually do not work across different releases of the Aegis operating system. This is because when a normal Aegis applications program is executed, the addresses of system functions (e.g., I/O calls) are resolved at load time. Thus, dumped versions of PSL have already had their address resolved and will not be changed upon start with run-image. When a new version of Aegis is installed the addresses of these system functions will probably change and become incompatible with the ones stored in the saved image. It is also possible the availability of the image's load address may change with different releases (or configurations) of the Aegis operating system. Thus, saved PSL images should **not** be used for long-term storage of application systems.

If for some reason it is not possible to exit the PSL image using ^Z and you need to free up the address space, the **unmap** command removes the image from the address space. As above, unmap takes one argument, the saved PSL image name with no extension, e.g.:

```
        ~p\unmap /tmp/pslsave
```

Alternatively, closing the shell the image was executed in also unmaps the image.

CHAPTER 4
INDEX OF CONCEPTS

The following is an alphabetical list of concepts, with the page on which they are discussed.

CHAPTER 5
INDEX OF FUNCTIONS

The following is an alphabetical list of the PSL functions, with the page on which they are defined.

CHAPTER 6
INDEX OF GLOBALS AND SWITCHES

The following is an alphabetical list of the PSL global variables, with the page on which they are defined.